Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues

Ivica Crnkovic

Department of Computer Engineering, Mälardalen University
721 23 Västerås, Sweden
E-mail: Ivica.Crnkovic@mdh.se

Juliana Küster Filipe*

Abt. Informationssysteme, Informatik, Technische Universität Braunschweig
Postfach 3329, D-38023 Braunschweig, Germany
E-mail: J.Kuester-Filipe@tu-bs.de

Magnus Larsson

ABB Automation Products AB, LAB
721 59 Västerås, Sweden
E-mail: Magnus.Larsson@mdh.se

Kung-Kiu Lau

Department of Computer Science, University of Manchester
Manchester M13 9PL, United Kingdom
E-mail: kung-kiu@cs.man.ac.uk

Abstract

In component-based software development, object-oriented design (OOD) frameworks are increasingly recognised as better units of reuse than objects. This is because OOD frameworks are groups of interacting objects, and as such they can better reflect practical systems in which objects tend to have more than one role in more than one context. In this paper, we show how to formally specify OOD frameworks, and briefly discuss their implementation and configuration management.

Keywords: Object-oriented design frameworks, component-based software development.

1. Introduction

Object-Oriented Design (OOD) frameworks are groups of (interacting) objects. For example, in the CBD (Component-based Software Development) methodology *Catalysis* [10], a driver

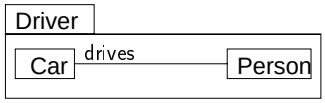


Figure 1. The Driver OOD framework.

may be represented as the OOD framework shown in Figure 1.[1] A driver is a person who drives a car, or in OOD terminology, a driver is a framework composed of a car object and a person object, linked by a 'drives' association (or attribute).

OOD frameworks are increasingly recognised as better units of reuse in software development than objects (see e.g. [12, 20]). The reason for this is that in practical systems, objects tend to have more than one role in more than one context, and OOD frameworks can capture this, whereas existing OOD methods (e.g. Fusion [6] and Syntropy [8]) cannot. The latter use classes or objects as the basic unit of design or reuse, and are based on the traditional view of an object, as shown in Figure 2, which regards an object as a closed entity with one fixed role. On the other hand, OOD frameworks allow objects that play different roles in different

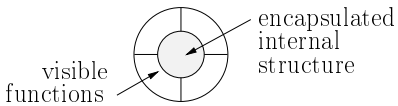


Figure 2. Traditional view of an object.

frameworks to be composed by composing OOD frameworks. In *Catalysis*, for instance, this is depicted in Figure 3.

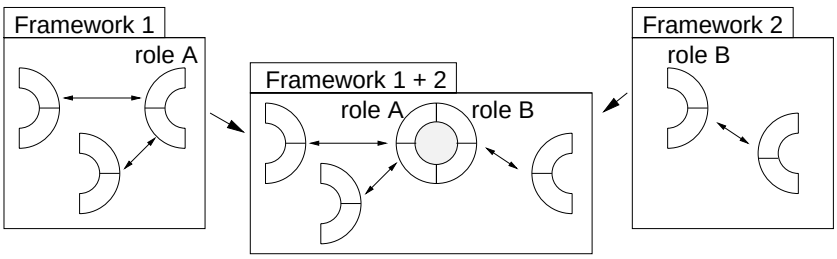


Figure 3. Objects by composing OOD frameworks.

For example, a person can play the roles of a driver and of a guest at a motel simultaneously. These roles are shown separately in the PersonAsDriver and PersonAsGuest OOD frameworks in Figure 4. If we compose these two frameworks, then we get the PersonAsDriverGuest OOD framework as shown in Figure 5. In this OOD framework, a person object plays two roles, and is a composite object of the kind depicted in Figure 3.

OOD frameworks should play a crucial role in the design and implementation of next-generation component-based software systems. In this paper, we show how to formally specify them, and briefly discuss their implementation (in COM) and configuration management.

*The second author was supported by the DFG under Eh 75/11-2 and partially by the EU under ESPRIT-IV WG 22704 ASPIRE.

[1] *Catalysis* uses the UML notation, see e.g. [21].

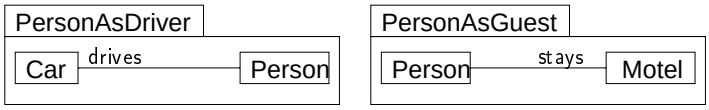


Figure 4. PersonAsDriver and PersonAsGuest OOD frameworks.

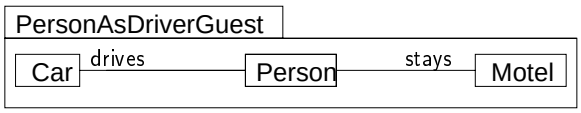


Figure 5. PersonAsDriverGuest OOD framework.

2. Formal Specification of OOD Frameworks

In this section, we describe formal specification of OOD frameworks. First we consider the *static* aspects, i.e. *without* time or state transitions, then we consider the *dynamic* aspects, i.e. *with* time and state transitions.

2.1. Static Aspects

We have considered the static aspects of OOD frameworks in [17, 18]. In this section, we briefly outline this semantics.

As we have seen in Section 1., OOD frameworks are composite objects/classes. In our approach, we define OOD frameworks and objects/classes in terms of a basic entity that we call a *specification framework*, or just a *framework*, for short.[2]

A *framework* $\mathcal{F} = \langle \Sigma, X \rangle$ is defined in the context of first-order logic with identity. It is composed of a signature Σ (containing sort symbols, function declarations and relation declarations), and a finite or recursive set X of Σ-axioms. The purpose of a framework is to axiomatise a problem domain and to reason about it. In our approach, a problem domain contains the ADT's and classes needed to define the objects of the application at hand.

A framework is thus a (first-order) theory, and we choose its intended model to be a *reachable isoinitial model*, defined as follows:[3]

Let X be a set of Σ-axioms. A Σ-structure i is an *isoinitial model* of X iff, for every other model m of X, there is one isomorphic embedding $i : \mathsf{i} \to \mathsf{m}$.

A model i is *reachable* if its elements can be represented by ground terms.

We distinguish between *closed* and *open* frameworks. The relationship between open and closed frameworks plays a crucial role in our interpretation of objects. Roughly speaking, in object-oriented programming terminology, open frameworks represent *classes*, and closed frameworks represent their instances, i.e. *objects*.

A framework $\mathcal{F} = \langle \Sigma, X \rangle$ is *closed* iff there is a reachable isoinitial model i of X.

An *open* framework $\mathcal{F}(\Omega) = \langle \Sigma, X \rangle$ does not have an isoinitial model, since its axioms leave open the meaning of some symbols Ω of the signature, that we call *open* symbols. Non-open symbols are called *defined* symbols.

Open frameworks can be *closed*, i.e made into closed frameworks, by instantiating its open symbols. We will use only open frameworks which have reachable isoinitial models for all their

[2] To avoid confusion with OOD frameworks, in this section we will use 'framework' to refer to a specification framework only, and not to an OOD framework.

[3] See [18] for a justification of this choice and [3, 16] for a discussion of isoinitial theories.

instances. Such frameworks are called *adequate*, and they can be constructed incrementally from small (adequate) closed frameworks (see [17, 18]).

Example 2..1 The Car class in Figures 4 and 5 can be defined as the following *open* framework:

OBJ-Framework $\mathcal{CAR}(.km, .option)$;
IMPORT: $\mathcal{INT}, year96$;
DECLS: $.km : [\,] \to Int$
$.option : [year96.opts]$
CONSTRS: $.km \geq 0$

where $\mathcal{INT}$ is a predefined ADT of integers, and $year96$ is an object that contains the sort $year96.opts$ of the possible options for a car in the year 96. The constraint $.km \geq 0$ is an axiom for the open symbol $.km$.

We call a framework like this an *OBJ-framework*, since it is a class of objects. To obtain objects we instantiate an OBJ-framework, i.e. by closing the OBJ-framework.

The axioms used to close $\mathcal{F}(\Omega)$ into an object represent the *state* of the object, and are called *state axioms*. State axioms can be updated, i.e. an object is a dynamic entity.

An object of class $\mathcal{CAR}$ is created, for example, by:

NEW $spider : \mathcal{CAR}$;
CLOSE: $spider.km$ BY $spider.km = 25000$;
$spider.option$ BY $spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond$.

where $spider.km = 25000$ and $spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond$ are (explicit) definitions that close the constant $spider.km$ and the predicate $spider.option(x)$ respectively.

The state of a spider object can be updated, by redefining its state axioms:

UPDATE $spider : \mathcal{CAR}$;
$spider.km = 27000$
$spider.option(x) \leftrightarrow x = Airbag \vee x = AirCond$

As we can see, the constant $spider.km$ has been changed.

An OOD framework is a composite OBJ-framework. It can be viewed as a *system* of objects, in which objects can be created (and deleted) and updated dynamically.

Example 2..2 The PersonAsDriver OOD framework in Figure 5 can be formalised as the following framework:

OOD-Framework $\mathcal{DRIVER}[\mathcal{PERSON}, \mathcal{CAR}]$;
DECLS: $.drives : [obj]$;
CONSTRS: $`X.drives(c) \to PERSON(`X) \wedge `X.age \geq 18 \wedge CAR(c)$;
$(\exists c : obj)(.drives(c))$;

where obj is a reserved sort symbol that contains the set of names of all existing objects in the system; $`X$ is a meta-symbol that stands for any object name; and the OBJ-framework $\mathcal{PERSON}$ may be something like:

OBJ-Framework $\mathcal{PERSON}$;

IMPORT ...;

DECLS: $.name : \rightarrow string;$
$.age : \rightarrow int; \ldots$

CONSTRS: $.age \geq 0$

Here the composite object is built via links between its components, which constrain object creation (and deletion) methods. We cannot create an object $n.\mathcal{DRIVER}$, if n is not a person. Furthermore, we need at least one car c.

2.2. Dynamic Aspects

In this section, we consider how to introduce time and state changes. We will combine the static formalisation outlined in the previous section with the logic MDTL presented in [13]. MDTL is an extension of the TROLL logic [11] for describing dynamic aspects of large object systems.

2.2.1. State Transitions.

In MDTL, an OBJ or OOD framework has a local logic consisting of a *home* and a *communication* logic. The *home* logic allows us to express internal state changes, whereas the *communication* logic describes framework interactions. The *home* logic of a framework is mainly a first-order temporal logic with (true) concurrency. We do not deal with concurrency explicitly in this paper, and so we will use axioms that are just first-order temporal formulae. Also, in this paper, we will not use the communication logic, since we do not deal with framework interactions.

In MDTL, in addition to attributes, an object also has *actions*,[4] which will affect its current state. Actions may be either *enabled* or *occurring* in a particular state. The *state* of an object is given by the current values of the attributes, and the current status of its actions. Thus in MDTL, a state formula is a conjunction of facts (the current values of the attributes) and actions (enabled or occurring).

If an action is *enabled*, then it may occur in the next state. When an action occurs, the state of the object changes. In the logic, $\odot a$ is used to denote the occurrence of action a, and $\rhd a$ that the action a is enabled. If an action occurs, then it must have been enabled in the previous state: $\odot a \Rightarrow Y \rhd a$. In this formula, Y is the temporal operator *Yesterday* referring to the previous state. Enabling ($\rhd$) is useful for expressing *preconditions*, and occurrence ($\odot$) for expressing *postconditions*.

In the sequel, we shall also use the temporal operators X (next state), F (sometime in the future including the present), and P (sometime in the past including the present).

The state of an OOD framework is given by the states of the current objects belonging to the framework.

We illustrate how to specify state transitions of an OOD framework in MDTL with a simple example.

Example 2..3 Consider the OOD framework for employees as depicted in Figure 6, in which a person plays the role of an employee of a company. A person as an employee has an attribute *pocket* representing the amount of money he possesses, and two actions *receive_pay* and *work*.

[4] More commonly known as *methods* in object-oriented programming.

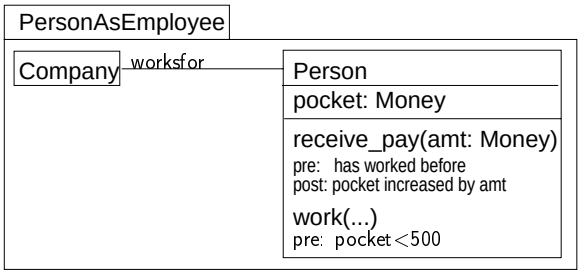


Figure 6. PersonAsEmployee OOD framework.

In this example, a person as an employee only works if he has less than £500 (precondition for $work$). A person only receives a payment if he has worked before (precondition for $receive\_pay$). If a person receives a payment, the money in his pocket increases by the amount received. Here, we express the pre- and postconditions only informally, and we omit the parameters and the postcondition of $work$, as well as the definition of the OBJ-framework for Company.

The class Person might be formalised by the following OBJ-framework:

OBJ-Framework $\mathcal{PERSON}$;

IMPORT: $\mathcal{MONEY}$

DECLS:
$$\begin{array}{rcl} .pocket & : & [\,] \to Money; \\ .receive\_pay & : & [Money]; \\ .work & : & [\ldots]; \end{array}$$

AXIOMS: ...

ST-AXIOMS:
$$\begin{array}{l} \forall_a \rhd .receive\_pay(a) \Rightarrow P \odot .work(\ldots); \\ \forall_{a,n} \odot .receive\_pay(a) \Rightarrow Y(.pocket = n) \Rightarrow .pocket = n + a; \\ \rhd .work(\ldots) \Rightarrow .pocket < £500. \end{array}$$

where the ST-AXIOMS are the *state transition axioms*.

Pre- and postconditions allow us to define the state transitions of a framework. The state transition axioms do not affect the (static) isoinitial model of the OBJ-framework, and are relevant only for the behaviour model. The first axiom states that if the action $receive\_pay$ is enabled, then sometime in the past (temporal operator P) the action $work(\ldots)$ must have occurred. The second axiom says that the occurrence of action $receive\_pay(a)$ implies that if in the previous state the value of $pocket$ was n, then its current value is $n + a$. Finally, the third axiom states that if the action $work(...)$ is enabled (it might occur in the next state) then the value of $pocket$ must be less than £500.

We can create an object joe of Person class as follows:

NEW $joe : \mathcal{PERSON}$;

CLOSE:
$$\begin{array}{rcl} joe.pocket & \text{BY} & joe.pocket = £100; \\ joe.work & \text{BY} & \rhd joe.work(\ldots); \\ joe.pay & \text{BY} & \neg \rhd joe.receive\_pay(a). \end{array}$$

When an object is created, its initial state is defined. In the initial state of joe, (attribute) $pocket$ is £100, (action) $receive\_pay(a)$ is disabled for any a, and (action) $work(\ldots)$ enabled.

In MDTL we can also express general properties of objects. For example,[5]

$$joe.pocket < £500 \Rightarrow F\ \exists_a \odot joe.receive\_pay(a)$$

means 'if *joe* has less than £500 then he will receive a payment sometime'.

2.2.2. Event Structures.

MDTL is interpreted over *labelled prime event structures* ([24]). A labelled prime event structure is thus a model for an OOD framework if it satisfies all the axioms of the framework (both the static and the state transition axioms).

A labelled prime event structure consists of a prime event structure and a labelling function. Prime event structures can be used to describe distributed computations as event occurrences together with a *causal* and a *conflict* relations between them. The causal relation implies a (partial) order among event occurrences, and the conflict relation denotes a choice. Events in conflict cannot belong to the same *run* or *life cycle*. The labelling function associates each event with a state.

Example 2..4 Consider the event structure in Figure 7. It shows a small part of a (sequential)

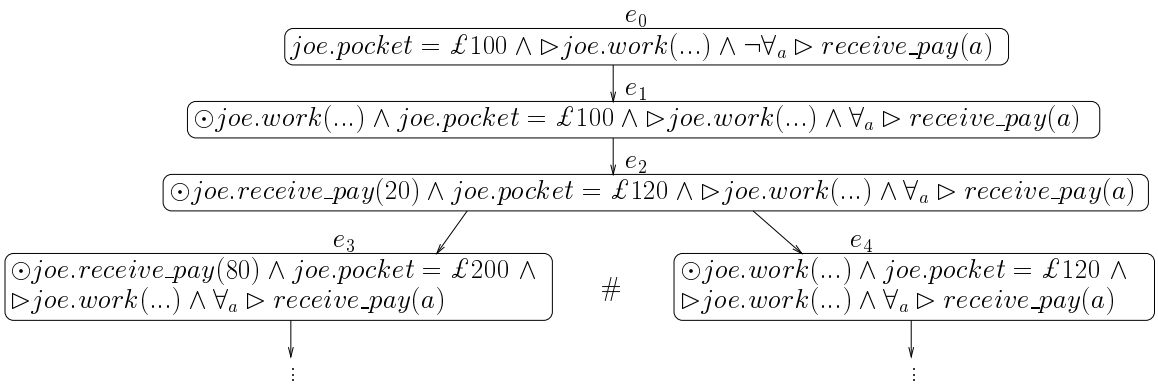


Figure 7. Event structure for *joe* as an employee.

behaviour model for *joe*.

In general, in event structures, *boxes* denote events $\{e_0, e_1, \ldots\}$, *arrows* between boxes represent event *causality*, and # denotes event *conflict*. The state of the object at a given event is written inside the box as a state formula.

For the object *joe*, the events in the event structure are labelled by the formulae of the state logic of the Person class. Event e_0 corresponds to the initial state. The occurrence of e_1 depends on the previous occurrence of e_0. With the occurrence of action $joe.receive\_pay(20)$ at event e_2, the current value of attribute *pocket* changes to £120. Events e_3 and e_4 are in conflict, which means that either one or the other occurs but not both. A conflict thus denotes a choice.

There are therefore two life cycles for *joe* in Figure 7. One consists of events $\{e_0, e_1, e_2, e_3, \ldots\}$ and the other $\{e_0, e_1, e_2, e_4, \ldots\}$. In the former, *joe* receives two payments after working. In the latter, *joe* works, then receives a payment and then works again.

Finally, it is easy to see that this event structure satisfies the state transition axioms of the OBJ-framework $\mathcal{PERSON}$.

[5] We are not saying that this property necessarily follows from the ST-axioms of the object *joe*.

In general, labelled event structures provide models for concurrent computations. Other such models include transition systems, Petri nets, traces, and synchronisation trees. Petri nets and transition systems allow an explicit representation of the (possibly repeating) states in a system, whereas trees, traces and event structures abstract away from such information, and focus instead on the behaviour in terms of patterns of occurrences of actions over time. Furthermore, event structures are a "true" concurrency model, as opposed to transition systems that model systems as non-deterministically interleaved sequential computations. A detailed survey and comparison of some of these models can be found in [24].

2.2.3. Composing Event Structures.

In order to create objects by composing OOD frameworks with state transitions in the manner depicted in Figure 3, we need to be able to compose event structures.

Example 2..5 In the previous example, a person plays the role of an employee. This partial definition of person could be combined with another view of a person, e.g., a person as a consumer. The PersonAsConsumer OOD framework in Figure 8 defines this role for a person

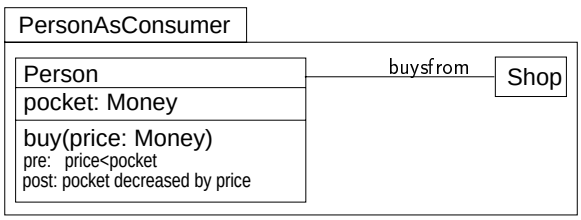


Figure 8. PersonAsConsumer OOD framework.

object. The class Person here can be defined by the same OBJ-framework $\mathcal{PERSON}$ in Example 2..3, but with the action $buy(p)$ instead of the actions $receive\_pay(a)$ and $work(\ldots)$. We omit the definition of the OBJ-framework for Shop.

A consumer has an action $buy(p)$, where p represents the price of the item bought. Pre- and postconditions for this action are the expected ones. A consumer may only buy something if he has enough money, and after buying an item the money in his pocket decreases by the amount of money spent. The state transition axioms for $buy(p)$ are thus:

$$\forall_p \rhd .buy(p) \Rightarrow .pocket > p$$
$$\forall_{p,n} \odot .buy(p) \Rightarrow Y(.pocket = n) \Rightarrow .pocket = n - p$$

Let *joe* be a person playing now the role of a consumer. In the event structure for *joe* as a consumer, a life cycle is a linear sequence of *buy* events starting from the initial state in which *joe.pocket* is initialised. In Figure 9, we show two possible life cycles with distinct initial states.

We may compose the OOD frameworks for PersonAsEmployee and PersonAsConsumer, to obtain a person with both roles together. A person now has all the actions of both roles, namely *receive_pay*, *work* and *buy*, and the attribute *pocket* in both roles. The composition is illustrated by Figure 10.

The composite PersonAsEmployeeConsumer framework contains the union of the state transition axioms of its component OOD frameworks. An event structure, i.e. a model, for a person as an employee and consumer is obtained by composing a model for person as an employee with one for person as a consumer in a special manner. Several constructions for sequential and parallel composition of event structures have been defined in the literature, e.g. [23, 19]. What we

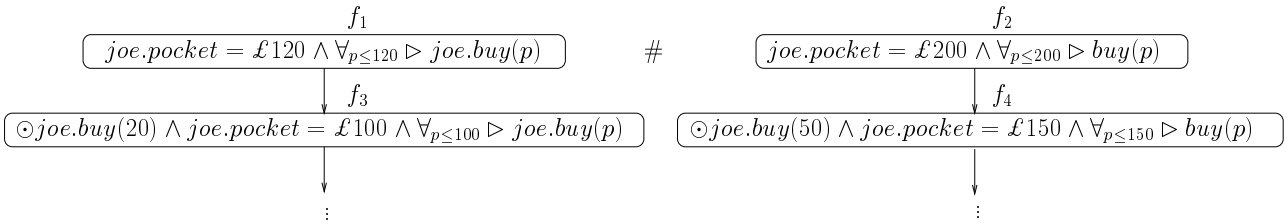


Figure 9. Event structure for *joe* as a consumer.

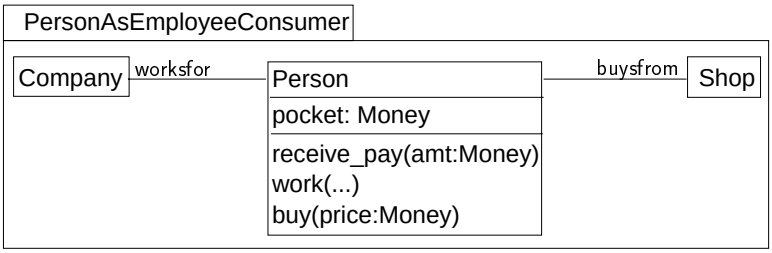


Figure 10. PersonAsEmployeeConsumer OOD framework.

need for composing roles is in fact a combination of *interleaving* of the models and *synchronisation*. Interleaving (sequential composition), because we are combining models for the same object in different roles (and objects are considered to behave sequentially), and therefore the composed model must be sequential. Synchronisation, because some attributes and/or actions for distinct roles may be identified as the same (e.g. *pocket*).

Figure 11 shows the composed model for *joe* as an employee and consumer based on the models of Figures 7 and 9 for *joe* as an employee and *joe* as a consumer respectively. In this

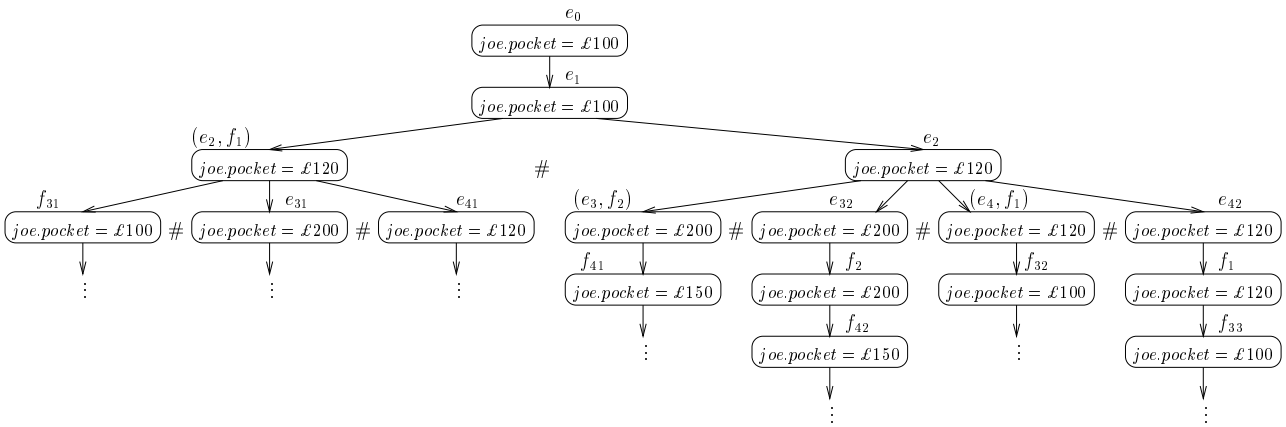


Figure 11. Event structure for *joe* as an employee and a consumer.

case, event synchronisation is done over the value of the common attribute *pocket*. That is, only those events of both models that have the same value for pocket may be synchronised, e.g., events e_2 and f_1, e_3 and f_2, and e_4 and f_1. Synchronisation is indicated by the pairs (e_2, f_1), etc.

Normally, synchronisation is done over actions, but we have a new situation here by combining roles, namely that the attribute *pocket* of a person as an employee is to be identified with the attribute *pocket* of a person as a consumer, whereas the actions *receive_pay*, *work* and *buy* are all distinct. Synchronisation is not always necessary, and some of the life cycles in

Figure 11 show just interleaving.

The construction of composite event structures sometimes leads to the duplication of events, e.g., event e_3 has been duplicated and corresponds to events e_{31} and e_{32}. The labels of these events are the same as e_3. In Figure 11, the states are just indicated by the value of the attribute *pocket* for simplicity. The label of event (e_2, f_1) is given by the conjunction of the labels e_2 and f_1, i.e., it corresponds to the formula

$$joe.pocket = £120 \wedge \odot receive\_pay(20) \wedge \triangleright joe.work \wedge \forall_a receive\_pay(a) \wedge \forall_{p \leq 120} \triangleright buy(p)$$

Parallel composition of event structures with synchronisation is useful for modelling interacting frameworks (e.g., [15]), whereas without synchronisation it models non-interacting frameworks.

3. Some Implementation Issues

Having shown how to formally specify OOD frameworks, we now turn to practical concerns. In particular, we will discuss how we might construct such frameworks in practice using currently available technology, and the issues involved in such constructions.

Of the current technologies for developing component-based software systems, COM [5] seems to lend itself most readily to the implementation of OOD frameworks. Therefore, we will briefly show how to implement OOD frameworks in COM.

The fact that OOD frameworks are composite objects/classes means that constructing these frameworks creates problems for *configuration management*. Therefore, we will consider some of these problems, in the case of COM implementations.

3.1. Implementing OOD Frameworks in COM

In this section, we show how to use COM to implement the OOD frameworks in Figures 6, 8 and 10. COM suits multiple roles because it can use multiple interfaces for each role. We will use the aggregation mechanism in COM to compose OOD frameworks. First, we implement the Person object, which corresponds to the encapsulated internal structure in Figure 2. The Person object is constructed so it supports aggregation of role objects and it has one IPerson interface (see Figure 12).

Figure 12. A COM object for the person object and the consumer role.

Secondly, the consumer and employee roles are implemented so they support being aggregated into a person object. Figure 12 shows the consumer role with one IConsumer interface. The consumer object also needs a reference to the person object to be able to work on the pocket variable. The person reference is set up when the consumer is aggregated into the person object (see Figure 13). In a similar way the employee role is implemented. Using aggregation we can

reuse the different components that we have created. Figure 13 shows how Person aggregates the two already defined COM objects. Frameworks are created at run-time by adding roles to an object.

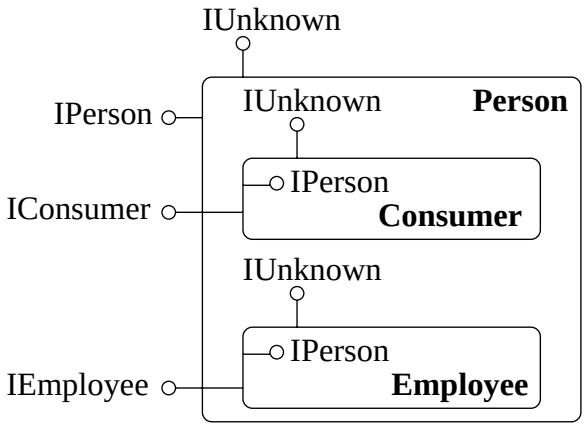


Figure 13. The Consumer and Employee roles are aggregated into the Person object.

The COM implementation of the framework concept has some limitations. The COM model defines frameworks as aggregates of the completed objects created at run-time, while a general framework model allows us to use incomplete objects (at run-time) or classes (at build-time).

3.2. Configuration Management

Using OOD frameworks instead of traditional objects yields several advantages, but it also introduces an additional level of complexity when building these frameworks. Frameworks are composite types of entities – they have an internal structure which is built from objects, or from parts of them. A framework entity also has relations to other frameworks, and can be composed from other (sub)frameworks. The definition and creation of such a composite entity introduces configuration problems. Some of them will be illustrated here for a COM implementation.

Let us consider the following cases:

- Sharing objects in several frameworks;
- Composing frameworks from objects and frameworks.

3.2.1. Sharing Objects in Several Frameworks.

Suppose framework F_1 includes objects O_1 and O_2 with a relation R_{12} between them, and framework F_2 contains objects O_1 and O_3 with a relation R_{13}. The object O_1 is shared by two frameworks:

$$F_1 = \{O_1\ O_2\ ;\ R_{12}\},\quad F_2 = \{O_1\ O_3\ ;\ R_{13}\} \tag{1}$$

Suppose we now add a new property to the object O_1, a property that is required in (an improved version of) framework F_2. This creates a new version of the object $O_{1;v2}$, ($v2$ denotes the new version) which is included into the framework F_2:

$$F_2 = \{O_{1;v2}\ O_3\ ;\ R_{13}\}$$

However, if we do not take versioning into consideration, then the framework specifications will remain the same. In this case, we can be aware of the change of the object O_1 in the context of framework F_2, but not necessarily in that of F_1. Our specification of F_1 is defined by (1), but in reality we have

$$F_1 = \{O_{1;v2}\ O_2\ ;\ R_{12}\}$$

If the role of the object $O_{1;v2}$ used in F_1 is changed, then the behaviour of F_1 will be changed unpredictably, and a system using F_1 can fail. To avoid these unpredictable situations we can introduce basic configuration management methods – a version management of objects and configuration of frameworks [7]:

- An object is identified by its name and version.
- A framework is identified by a name and a version. A new framework version is derived from object versions included in the framework.

These rules imply that new versions of frameworks will be configured when a new object version is created, as shown in our example:

$$F_{1;vi} = \{O_{1;vm}\ O_{2;vn}\ ;\ R_{12}\},\ \ F_{2;vk} = \{O_{1;vm}\ O_{3;vk}\ ;\ R_{13}\}$$

$$F_{1;vi+1} = \{O_{1;vm+1}\ O_{2;vn}\ ;\ R_{12}\},\ \ F_{2;vk+1} = \{O_{1;vm+1}\ O_{3;vk}\ ;\ R_{13}\}$$

As several frameworks can share one object, and a framework can contain several objects, the number of generated frameworks can grow explosively. It is, however, possible to limit the number of interesting configurations. Typically, in a development process, we would implement the changes on all the objects we want, collect the versions of objects we want in a baseline and derive the frameworks from the baselined object versions. In such a case, experience for similar cases [2] shows that the number of derived entities does not necessarily grow rapidly.

A shared object is not necessarily completely shared, but different parts of the object, defined by the object's roles, are used in the frameworks. In the COM implementation a complete object will be included, but a part of it will be used. In a general framework model, a class (or an object at run-time) includes only those parts which are specified in the object's role. When we define a new role for an object in a framework or re-define the existing one, we need to change a specific part of the object class. We call this specific part an object aspect. The change of an object aspect will affect only those frameworks where the aspect is included. Other frameworks, though containing the object (or part of it), are not affected by the change. In this case, it is better to keep version control on the aspect level, and relate a framework configuration to the object aspects.

If we declare an aspect as a subset of an object $A_i(O_k) \subseteq O_k$, then an object version is defined as a set of aspect versions:

$$O_{i;vk} = \{A_{j;vl}\}$$

and a framework version is defined as a set of aspect versions with relations between the aspects:

$$F_{vk} = \{\{A_{j;vl}(O_{i;vk})\}\ ;\ R_{jl}\}$$

Having control over changes on the aspect level, we can gain control over the changes on the framework level. Now we can more precisely identify the frameworks being affected by changes in object roles.

3.2.2. Composing Frameworks from Objects and Frameworks.

In the framework model it is possible to compose new frameworks from existing frameworks. A new framework is a superset of the classes and relations from the frameworks involved. If a new framework is created at run-time, as in a COM implementation, then the objects from the selected frameworks comprise the new framework. The following example illustrates the merging process of two frameworks F_1 and F_2 into $F3$:

$$F_1 = \{O_1\ O_2\ ;\ R_{12}\},\ \ F_2 = \{O_1\ O_3\ ;\ R_{13}\},\ \ F_3 = \{O_1\ O_2\ O_3\ ;\ R_{12}, R_{13}, R_{23}\}$$

The composition works fine as long as we do not need to consider the changes of objects within one framework.

Suppose we create a new object version (or a new object aspect version) in F_2 and keep the old version of the same object in F_1:

$$F_{1;vi} = \{O_{1;v1}\ O_{2;vk}\ ;\ R_{12}\},\ \ F_{2;vj} = \{O_{1;v1+1}\ O_{3;vl}\ ;\ R_{13}\}$$

In the merging process we have to recognise if different versions of the same objects are included in the frameworks being merged. If that is the case, we have two possible solutions:

- Selecting one specific version of the object (for example the latest):

$$F_{3;v1} = \{O_{1;v1+1}\ O_{2;vk}\ O_{3;v1}\ ;\ R_{12}, R_{13}, R_{23}\}$$

- Selecting both versions and enable their consistence in the new framework:

$$F_{3;v1} = \{O_{1;v1}\ O_{1;v1+1}\ O_2; vk\ O_{3;v1}\ ;\ R_{12}, R_{13}, R_{23}\}$$

For the second case there must be support for identifying object versions. This support can be provided by introducing an identification interface [14] as the standard interface of an object. There must also be support for managing different versions of the same object in the running system.

4. Conclusion

In this paper we have shown how to formally specify OOD frameworks using MDTL and event structures. In particular, we have shown a semantics for composing OOD frameworks with state transitions in the manner depicted in Figure 3.

Our work here is closely related to TROLL [11], which is used for specifying large distributed/concurrent object systems, and to [4], which formalises an algebraic semantics for object model diagrams in OMT [22]. The main difference is that they take the traditional view of objects (Figure 2), whereas we adopt the multiple-role, more reusable approach (Figure 3). Their semantics is based on initial theories, as opposed to isoinitial theories that we use.

Overall, our approach to specification is model-theoretic, whereas other approaches are mostly proof- or type-theoretic. For example, our model-theoretic characterisation of states and objects stands in contrast to the type-theoretic approach, e.g., [1]. Our model-theoretic approach also enables us to define a notion of correctness that is preserved through inheritance hierarchies, which is particularly suitable for component-based software development.

We have also presented a possible implementation of OOD frameworks using the COM technology. This implementation has some limitations, and we need to do further work to investigate how to improve this implementation.

Finally we have discussed configuration management for frameworks. We emphasise a need for using configuration management methods for managing frameworks as composite objects. The configuration management issues are complicated and need further investigation: Questions of managing relations, concurrent versions of frameworks, inclusion of change management [9], etc., must be addressed. Since aspects and objects are not entities recognised by standard configuration management tools (which recognise entities such as files, directories, etc.), new, semantic-based rules must be incorporated into such tools. For different object-oriented and component technologies, different tools have to be made. How different do they need to be, and are there possibilities to define common rules and implementation? These are questions for future investigation.

Acknowledgements

We would like to thank the referee who pointed out some minor mistakes.

References

[1] M. Abadi and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag.

[2] U. Asklund, L. Bendix, H.B. Cristensen, and B. Magnusson (1999). The unified extensional versioning model. In J. Estublier, editor, *Proc. System Configuration Managemnt SCM-9*, pages 17–33, Springer.

[3] A. Bertoni, G. Mauri, and P. Miglioli (1983). On the power of model theory in specifying abstract data types and in capturing their recursiveness. *Fundamenta Informaticae* **VI**(2):127–170.

[4] R.H. Bourdeau and B.H.C. Cheng (1995). A formal semantics for object model diagrams. *IEEE Trans. Soft. Eng.*, **21**(10):799-821.

[5] D. Box (1998). *Essential COM*. Addison-Wesely.

[6] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes (1994). *Object-Oriented Development: The Fusion Method*. Prentice-Hall.

[7] R. Conradi and B. Westfechtel (1998). Version models for software configuration management. ACM Computing Surveys, **30**(2):232–282.

[8] S. Cook and J. Daniels (1994). *Designing Object Systems*. Prentice-Hall.

[9] I. Crnkovic (1997). Experience with change oriented SCM Tools. In R. Conradi, editor, *Proc. Software Configuration Management SCM-7*, pages 222–234, Springer.

[10] D.F. D'Souza and A.C. Wills (1998). *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley.

[11] A. Grau, J. Küster Filipe, M. Kowsari, S. Eckstein, R. Pinger and H.-D. Ehrich (1998). The TROLL approach to conceptual modelling: syntax, semantics and tools. In T.W. Ling, S. Ram and M.L. Leebook, editors, *Proc. 17th Int. Conference on Conceptual Modeling, LNCS* **1507**:277-290, Springer.

[12] R. Helm, I.M. Holland, and D. Gangopadhay (1990). Contracts — Specifying behavioural compositions in OO systems. *Sigplan Notices* **25**(10) (*Proc. ECOOP/OOPSLA 90*).

[13] J. Küster Filipe (2000). Fundamentals of a module logic for distributed object systems. *J. Functional and Logic Programming* **2000**(3).

[14] M. Larsson and I. Crnkovic (1999). New challenges for configuration management. In J. Estublier, editor, *Proc. System Configuration Management SCM-9*, pages 232–243, Springer.

[15] K.-K. Lau, S. Liu, M. Ornaghi, and A. Wills (1998). Interacting frameworks in *Catalysis*. In J. Staples, M. Hinchey and S. Liu, editors, *Proc. Second IEEE Int. Conf. on Formal Engineering Methods*, pages 110-119, IEEE Computer Society Press.

[16] K.-K. Lau and M. Ornaghi (1998). Isoinitial models for logic programs: A preliminary study. In J.L. Freire-Nistal, M. Falaschi, and M. Vilares-Ferro, editors, *Proceedings of the 1998 Joint Conference on Declarative Programming*, pages 443-455, A Coruña, Spain.

[17] K.-K. Lau and M. Ornaghi (1998). On specification and correctness of OOD frameworks in computational logic. In A. Brogi and P. Hill, editors, *Proc. 1st Int. Workshop on Component-based Software Development in Computational Logic*, pages 59-75, September 1998, Pisa, Italy.

[18] K.-K. Lau and M. Ornaghi (1998). OOD frameworks in component-based software development in computational logic. In P. Flener, editor, *Proc. LOPSTR'98*, *LNCS* **1559**:101-123, Springer-Verlag.

[19] R. Loogen and U. Goltz (1991). Modelling nondeterministic concurrent processes with event structures. *Fundamenta Informaticae* **XIV**(1):39–73.

[20] R. Mauth (1996). A better foundation: development frameworks let you build an application with reusable objects. *BYTE* **21**(9):40IS 10-13.

[21] R. Pooley and P. Stevens (1999). *Using UML: Software Engineering with Objects and Components*. Addison-Wesley.

[22] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Sorenson (1991). *Object-Oriented Modeling and Design*. Prentice-Hall.

[23] F.W. Vaandrager (1989). A simple definition for parallel composition of prime event structures. Technical Report CS-R8903, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.

[24] G. Winskel and M. Nielsen (1995). Models for concurrency. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, pages 1–148. Oxford Science Publications.

Towards a Toolset for Architectural Design of Distributed Real-Time Control Systems

Jad El-khoury and Martin Törngren
The Mechatronics Lab, Department of Machine Design
Royal Institute of Technology - KTH, Stockholm, Sweden
{jad, martin}@md.kth.se

Abstract

We describe a novel tool developed to support architectural design of distributed real-time control systems. The approach enables the co-simulation of functionality (from discrete-time control to logic) together with the controlled continuous-time processes and the behavior of the computer system. In particular, modelling and simulation of distributed computer control systems is supported allowing analysis of timing and control system robustness. An emphasised feature of the tool is its multidisciplinary and integrated approach that combines the views of control and computer engineering into one view at an appropriate level of abstraction. The use of the models and the tool is illustrated through examples and the usefulness of the approach for architectural design is discussed together with avenues for further work. The current models and the tool are designed with the modelling and analysis of fault-tolerant systems in mind. Improved support in this direction is the subject of ongoing work.

1. Introduction

The mechatronics perspective provides a large number of opportunities to create improved machinery with the aid of software, electronics, sensors, actuators and control. Modern machinery, such as automobiles, trains and aircraft, are equipped with embedded distributed computer control systems, in which the software implemented functionality is steadily increasing. At the same time, a number of challenges face the developers of machinery in order to really be able to benefit from the technological advances:

- *Managing complexity* arising from the sheer amount of new functionality, as well as the non trivial dependencies between system functions and components.
- *Managing and exploiting multidisciplinarity* in order to achieve cost-efficient solutions and maintain consistency (e.g. between goals, assumptions, design and implementation) during development.
- *Verification and validation of dependability requirements* with the challenge of developing mechatronic systems that are safer and more reliable than their mechanical counterparts.

Taking the current automotive systems as an example, the computer system is typically composed of a number of nodes connected via communication networks. With the introduction of new functions in vehicular systems it is inevitable that these will share resources in terms of sensors, actuators, communication links and computer nodes with the incentive of cost reduction. For example, in a brake-by wire system, the dimensioning of the brake actuators will need to consider not only the basic “by wire braking”, but also braking demands from vehicle stability and cruise control. Functional coordination will consequently be an issue of increasing importance. The sharing of sensors will influence the choice of sensors, filtering and the information distribution. Computer system sharing influences resource management and timing. Perhaps most importantly, all types of sharing will have impact on the system dependability and maintainability.

The development approach of such future mechatronic systems, however, is still vague partly because the application area is young and partly since it requires the integrated cooperation of various disciplines from mechanical, control and computer engineering. A key element in addressing these challenges is to develop models, methods and tools to support, in particular, the so called *architectural design* stages.

1.1. The road towards architectural design

Essential architectural design decisions include determining:

- The overall system structures, including functional, software and computer structures.

- The principles and means for error detection and handling, and at what level these should be implemented.
- The policies for resource scheduling, synchronisation, communication, and function/software triggering.

Establishing a system architecture requires considering conflicting requirements such as cost, flexibility, safety and reliability. One difficulty and typical characteristic is that while some of these properties, such as reliability and performance may be quantifiable, others are difficult or impossible to quantify, and in choosing an architecture, the balancing and resulting trade-offs become more or less subjective.

Architectural design requires the use of appropriate models that describe structural and behavioural properties [9], [4]. Models together with proper analysis tools provide the possibility to perform solution space exploration early in the development process, and prior to the point in time where physical hardware (mechanics and/or electronics) is available. Models also capture knowledge and form the basis for reuse. A model and architecture based approach is clearly highly motivated given the current state of practice in industrial development of embedded control systems where huge efforts and resources are spent on testing and debugging. From an industrial perspective, Hanselmann [8] points out the need for, and the current lack of, tools that allow early analysis and verification of distributed control systems.

The next section describes the general approach taken in this project, followed by related work in section 3. Section 4 gives an overview of the model adopted in this approach. After some implementation details in section 5, our approach is illustrated in section 6 by walking through a simple example. Finally, this paper concludes with some future work proposals.

2. Approach and focus

The suggested approach allows for the analysis of various architectural design decisions, and their impacts on control functionality, through a modelling and simulation tool. This analysis is intended to be carried out early in the development process and being useful for engineers from various disciplines. The co-simulation of the control system functionality together with implementation details of the chosen computer structure, allows the user to directly study the resulting system behaviour, and thereby gives the possibility to try out different architectures. In addition, the modelling process, spanning control design and computer system implementation, is important as such in that it promotes the interdisciplinary design. The tool simulation approach obviously needs to be complemented by other types of analyses and approaches, such as control analysis for performance and stability, and static timing analysis.

The work described in this paper in particular allows the modelling and simulation of:

- *Application software* encompassing different functionalities in a wide variety of styles (e.g. discrete-time, even-triggered, data-flow, state machines etc.).
- *System software* such as communication protocols where in principle the same modelling styles are used as for the application software. The concurrency model includes scheduling and thread interactions such as inter thread communication.
- *Distributed computer systems* including networks and computer system nodes (composed of processors with network and I/O interfaces). Global scheduling is supported including scheduling of processors, local and global communication, and other resources.
- *Mechanical systems* with sensors, actuators and mechanical system dynamics.

The work forms part of a larger research effort at the Mechatronics lab [14] where a common denominator is the development of a toolset supporting model based architectural design with different types of analysis. An essential requirement for the models and the toolset is the interdisciplinary design support encompassing system, control, computer and mechanical engineers. This will require a large set of models and analysis features including models encompassing discrete-time control, finite state machines, continuous time dynamics, software tasks and resource management. Our approach is to build upon and reuse existing models and tools where appropriate, and to add models and analysis features as required, as illustrated in this paper.

3. Related Work

From a control system point of view, modelling and simulation of distributed system implementations has been used at least since the late 80s. The typical approach is to map all implementation related problems into control related effects such as time-varying feedback delays, sampling period jitter, data loss and permanent errors causing open loop drift (e.g. controller stuck at the maximum value) [12], [15], [22]. In [18] it was shown how special Simulink [10] blocks, can be developed to model these effects. Although this approach is suitable from a control design point of view it does not explicitly model the real-time computer implementation and thus lacks in supporting holistic architectural design.

Simulation tools such as the ones described in [16], [2] and [23] tackle various aspects of the design and analysis of

real-time applications. They are, however, limited in their use in the multidisciplinary approach suggested in this paper. The models used may be too abstract or domain specific to be understood by another discipline. Also, the tools' aim could be to analyse a particular aspect of the system such as a communication protocol or a specific operating system. In particular, little support exists for the co-simulation of a control application with the computer system. Exceptions to this are the tools described in [5] and [6], in which a control application can be simulated together with a task scheduler, and with potential extensions to include data sharing mechanisms. However, the usability aspects of this tool are weak from the mechatronics perspective since the control application needs to be implemented in code prior to simulation and changes to the system parameters are not straightforward.

There is relatively little work on interdisciplinary modelling. Exceptions in this regard include [21], [11] and [3]. The latter two describe work that deals with safety and reliability analysis in the context of computer control systems.

More detailed surveys of related work can be found in [13] and [20].

4. The Model

In order to achieve the goal of a multidisciplinary modelling environment, modelling aspects were borrowed from models in the various disciplines. In particular, three models are worth pointing out here. The AIDA modelling framework [13] provided insights into the control implementation requirements needed, the component models and their parameters. As a software engineering design methodology and model, the CODARTS method by Gomaa [7] highlighted the aspects of software that need to be included. Finally, the control engineering approach of using data flow diagrams was kept in the modelling of the internal task structure.

Inspiration is also drawn from an industrial case study where the task was to evaluate the fault-tolerant computer control system of the SMART satellite, to be launched by the European Space Agency in 2002. The aim of the evaluation was to analyse the high level redundancy protocols being introduced in the system, and their behavior when implemented using the Controller Area Network (CAN). The work entailed modelling the distributed fault-tolerant computer system and simulating its behaviour during stipulated failure modes such as loss of communication, node loss, and various CAN controller failures. The modelling, tool development and evaluation efforts are further described in [19].

The models described below maintain a good balance between three factors that were considered important. First, a good representative model is needed to reflect real entities in the system and which is not too mathematically abstract. At the same time, an executable model needs to be well defined and unambiguous. Finally, the model should contain a minimal number of different components with a simple and generic interface in order to simplify the development and use of the tool.

4.1. System Topology Model

At the top level, the hardware topology of the whole system is modelled. This hardware structure consists of three types of components: The surrounding environment, communication links, and the computer nodes. The node model will be described in more detail in section 4.2.

Environment. It is necessary to include the environment into the model in order to define its connection to the embedded computer system. With the types of applications considered here, the most apparent entity that needs to be modelled is the mechanical dynamics of the system including sensors and actuators. For this, well-established mathematical models exist. These models will not be discussed any further and the reader can refer to a wide range of dynamics textbooks for details. It is however necessary that the chosen environment model is integrable with the rest of the hardware model, meaning that it should be possible to actuate and sense appropriate parameters of the mechanics.

A computer node connects to the system environment at various points. This connection is performed via Hardware Units (see "Hardware Units" in Section 4.2) such as pulse width modulators (PWM), analogue to digital converters (ADC), and digital to analogue converters (DAC) that reside in the node.

Communication Link. A communication link provides data exchange facilities between computer nodes. It defines the protocols that handle the messages being sent between connected nodes. A communication link indirectly interacts with each connected node through *communication controllers* that reside in the node. The communication link performs the scheduling of messages requested from connected controllers, while the controller internally schedules its own messages. Such a setup allows for a representation of a multitude of node and link models to be connected, as well as allowing for a node to connect to one or more links, and vice versa.

As an example, consider the implementation of the CAN bus. Requests to send messages on the bus are received by the bus from the controllers. These requests are only

serviced if the bus is idle, and ongoing transmissions are not disturbed. Once the bus is idle, controllers arbitrate between each other to gain access to the bus according to the CAN protocol. The message transmission time depends on the bus bit rate and the message size including any protocol overheads. Note that the arbitration within a CAN controller is performed independently at the node level, and that this local scheduling is not part of the CAN protocol itself.

4.2. Node model

A node consists of the following types of components:

- One or more tasks from which the system software is built. The task model is described in section 4.3.
- A task scheduler.
- Zero or more operating system Service Providers (SP) such as inter-task communication, task synchronisation and semaphores.
- Zero or more hardware units such as communication controllers, timers, ADCs and DACs
- A processor.

The application functionality to be developed by the user is composed of application tasks, with services provided by the other components comprising the operating system. Software layers, which interface the application software to the system hardware and operating system, can be easily modelled and designed with this approach. For example, system tasks, belonging to the operating system, can be developed to implement software drivers or high level network protocols.

Scheduler. A single task scheduler exists for each node in the system. The scheduler's role is to, when triggered, simply choose and activate a single task that is to run on the node processor at that time. The scheduler may be triggered by any of the service providers installed on that node, or by a timer reaching certain predefined points in time. A *task list* component also exists which holds certain information on each task such as its ID, current status, priority and any user-specific parameters. The scheduler only needs to interface to the task list in its decision making, and different scheduler models require different information about the tasks and hence, the task list model should be consistent with that of the scheduler.

This model allows for the modelling of a wide range of schedulers such as event/time triggered, static/dynamic, and off-line/on-line schedulers. Developing a new scheduler requires the implementation of the function that decides on the next running task, as well as the design of the data structures needed for each task in the task list. Using a particular scheduler simply requires the inclusion of the scheduler and its accompanying task list into the node, and any off-line customisation is done by the user through the task list.

As an illustrating example, consider the design of a fixed priority preemptive scheduler. The task list stores, for each task, the user specified static priority as well as its current status. A task status can be one of *ready*, *running* or *blocked*. The scheduler in this case is triggered every time a task changes its status in order to evaluate if a more legitimate ready task needs to run on the processor. A scheduler triggering may be caused by, for example, an interrupt or a service provider.

Service Provider. Examples of service providers are inter-task communication and task synchronisation. Although they vary in their functionality, these components have very similar features and interactions to the rest of the system. Essentially, a *service provider* (SP) responds to a *service request* from a task to perform certain activities. This activity may cause the calling task (or any other task, in general) to change status due to the internal state of the SP.

The mechanism of making requests by a task and the response to these requests is fixed across all services. What varies is the interpretation of the requests and the way they are handled. Hence, developing new services simply requires the definition of the internal states, and the functionality to handle the various types of possible requests. All SPs have access to the task list and are able to trigger the processor scheduler.

As an example, consider an inter-task communication service implemented as a first-in/first-out, block-on-full service. When a task requests to send a message, it simply sends the data to the specific SP. Normally, the SP places the data in the FIFO buffer. However, if the buffer is full, the SP changes the task status to blocked, and triggers the scheduler. When the buffer is available again, the SP changes the task status to ready and triggers the scheduler.

Hardware Unit. Hardware units may be fully embedded in the computer node (such as a floating point processor) and hence only interfacing with the processor, or they could lie on the border (such as an ADC) and provide an interface between the processor and the surrounding environment.

From the task perspective, the interface to a hardware unit is similar to that of a service provider in that a request is made for a service which the unit provides. Hence, it is important to match the requests to the correct service. However, a hardware unit differs from a service provider in that it has no direct access to the internal data structures of the OS such as the scheduler or task list. Instead, the unit may cause processor interrupts that tasks in the system need to handle appropriately. This model naturally facilitates the masking of these units by developing unit drivers

(consisting of system tasks) that encapsulate the hardware units and handle the generated interrupts.

As a simple example, consider a hardware unit implementing a CAN communication controller. A task requesting to send a message over CAN makes a request for service by directly accessing the hardware registers. The unit in turn communicates with the associated CAN communication link, and upon receiving a message from the link, produces a hardware interrupt (if so configured). The CAN controller performs the local scheduling of simultaneous transmission requests, e.g. FIFO or priority based.

4.3. The task model

A task is modelled as a single sequence of *elementary functions* (EF). Each EF is assumed to take a specified non-zero amount of time to execute. We can draw a simple task as shown in Figure 1a, illustrating the precedence relationship between the elementary functions. The EF can be either user-specific (octagonal block representation) or an operating system service request (rectangular block).

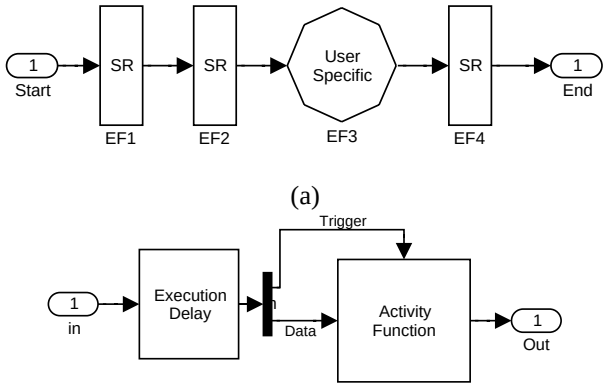


Figure 1. (a) The task model consisting of a sequence of elementary functions (EF). (b) The internal model of an EF.

When first triggered, the task is made ready to run on the processor. During its lifetime, and depending on the system activities and the scheduler being used, a task runs on the processor at different time slots. The directed link between two EFs within a task indicates passing control from the source to the destination EF, triggering the destination EF to begin its execution. The currently activated EF terminates once the task executes for a time period that is equivalent to the EF execution time, since the EF was first activated. When the control is passed to the last block, the task is terminated.

This simple model can be extended in order to provide looping and branching of the elementary functions, as shown in Figure 2. Once triggered, the Branch block (B) produces an output trigger in one, and only one of its outputs, based on internal logic. The Merge block (M) produces a trigger on the output as soon as any of its inputs is triggered. These mechanisms do not consume any processor time, and are only intended to be used for high level modes of operations of the application.

Tasks may be initially triggered as soon as the node is booted, or they may be configured to be triggered by an interrupt. The first is typical for many application tasks, while the latter can be used to model system interrupt handlers. It is also necessary to have at least a single task (the system idle task) in the system that is initiated during boot time, and that may never terminate.

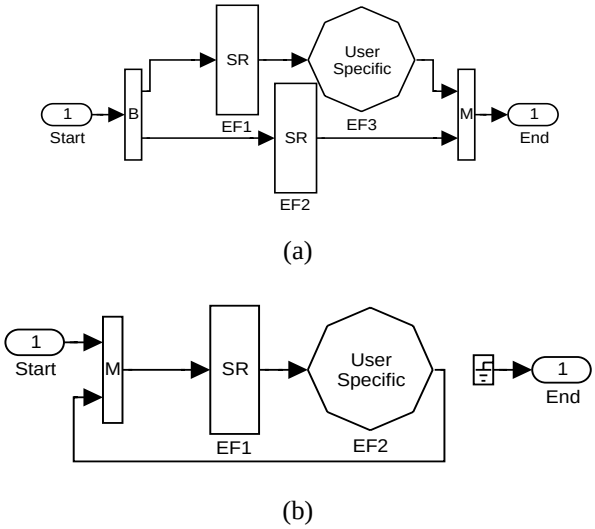


Figure 2. Examples of more complex task structures such as (a) branching or (b) looping.

Elementary Function (EF). The internal model of an EF is shown in Figure 1b. The input to an EF (either user-specific or an operating system service request) consists of the elementary function trigger and its data. When an EF is first triggered, the input data is captured and when its specified execution time elapses, an output trigger is produced together with output data. By definition, an elementary function may be preempted at any instance in its execution. Non-preemptive EFs can be implemented as a subset of normal EFs, by implicitly surrounding each elementary function with service requests that disable and enable preemption.

4.4. Model attributes

Table 1 lists the explicit attributes that the user needs to specify for each of the components in the model. In addition, components contain implicit properties that can be automatically derived from their context. For example, a task implicitly identifies the list of elementary functions that are contained within it. Also, each component contains a unique identifier that distinguishes it from other components.

Table 1. Explicit component attributes.

Component	Attributes
Communication link	Link Speed Communication Protocol
Scheduler	Scheduling Algorithm
Service Provider/ Hardware Unit	Service Response Policies Internal Buffer Sizes
Elementary Function	Execution Time

4.5. Software structure

As well as being executable, the models described above are representative enough that the collection of task models from each of the nodes in the system can serve as a basis for a detailed software model, which could be translated to a more traditional and familiar form, if desired. Sufficient level of detail is available to generate pseudo-code for each task in the system, and configuration information indicating the services needed for each node.

5. Simulator Overview

In this section, we will give a brief outline of the simulator implementation of the above models. More detailed description requires extensive explanations of various Matlab/Simulink aspects and is the subject of a technical report to be produced in the near future. The Matlab/Simulink toolbox [10] has been chosen to implement the simulator for the following reasons:

- Working in the control engineer environment allows the control engineer to specify, validate and interact with the computer engineer in a familiar environment that reduces the chances of ambiguity and confusion.
- A rich API is available for the extension of the tool in a variety of languages such as C and Java. Combined with the broad range of existing blocks, the tool development is simplified, particularly for monitoring and testing.
- Simulink allows for the integration of custom code into its models. Hence, it is possible to model the control application together with any encapsulating software. In addition, Simulink supports hierarchical models.
- Simulink supports modelling and simulation of hybrid systems.
- Many commercial development tools such as rapid prototyping tools, working with Simulink, can give a well integrated working environment.

On the other hand, there is a strong gap between the Simulink modelling level and the real-time system implementation, further motivating this work.

The handling of hybrid systems required the simulator to have an event-triggered architecture, where all the activities in the system are event-triggered, and the Simulink triggering capabilities are used extensively. Note that this still allows for the modelling of purely time-triggered architectures, where time is viewed as an event. A combination of C-coded S-functions and Simulink blocks were used in the implementation.

Each of the model components discussed in section 4, is actually represented in the simulator as a Simulink block. The drag-and-drop approach of blocks from libraries is used to build the models. Blocks are then customised through a graphical user interface. Also, there are no restrictions on the types of standard Simulink blocks that can be used with the simulator models, except for those imposed by Simulink itself. This ensures a well integrated environment with Simulink and the user does not need to learn a new tool.

As can be seen from the definitions in section 4, there is extensive data exchange between the components. Taking the traditional Simulink approach of connecting blocks to exchange data is not favourable since this will certainly complicate the model for the simplest cases. Even worse, confusion will occur between data exchange of the application itself and that needed for the implementation of the underlying components. The approach taken in the simulator is to hide all data exchanges that do not form part of the representation model of the system. For example, although data exchange is necessary between a scheduler and each of the tasks on its processor, these links are not explicit in the model presentation since they do not contribute to the understanding of the model. The user need not be concerned with these hidden links since each component automatically reconfigures itself based on the presence of other blocks in the system. Also, the interface between these types of components within a node is well defined and fixed, allowing for the independent development of subtypes and variations in the internals of each of the components.

The simulator permits the user to monitor any variable in the system, as illustrated by the examples in section 6.

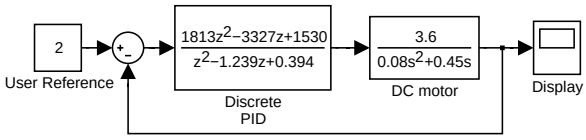
Parameters that may be of interest for timing analysis include a task's status, the times when particular events or activities within a task occur, or the time when a service request is serviced.

6. Illustrative Examples

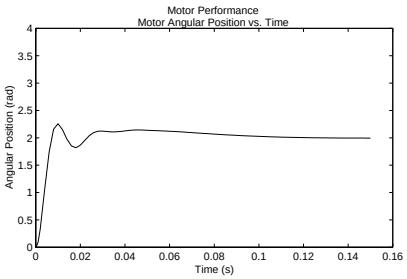
The aim of this simple example is to demonstrate how the tool can be used in architectural design for the evaluation of the computer implementation effects on the control performance. We will proceed from a pure functional design, to a single node and a distributed system implementation. In addition, the usability aspects of the tool are illustrated.

6.1. Controller Design

Consider the problem of controlling the angular position of a DC motor in order to reach a set point specified by the user. The control algorithm is to be implemented on a computer system. After modelling the mechanics of the motor, the control engineer uses established techniques in designing a discrete controller that produces acceptable performance. A Simulink model of the resulting system is represented in Figure 3a.



(a)



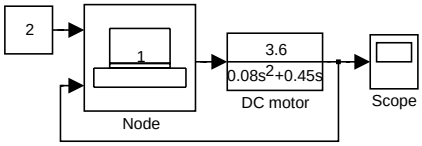
(b)

Figure 3. (a) A pure functional model of a DC-motor controller together with a model of the motor dynamics. (b) Closed loop performance of the system in Figure 3a for a step response.

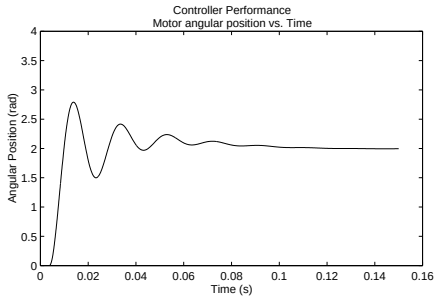
In this example, a continuous time PID controller was developed and then translated into a corresponding discrete-time controller. A sampling period of 2ms was chosen according to standard rules of thumb [24]. The angular position of the motor and the user reference value are measured by the controller. The controller outputs and controls the motor voltage. This information becomes the software specification to be implemented. At this stage of development, important requirements such as controller jitter and delays are often overlooked, since they are dependant on implementation details and their values can only be deduced once the system is implemented. As shown in [5] and [22], these parameters are critical to ensure a certain controller performance. One approach to determine these parameters is to iterate between the control design and the software implementation until satisfactory results are achieved. Another cheaper alternative is to model and evaluate the computer implementation effects early in the design stage, and to try to take appropriate design measures before the implementation is carried out.

6.2. A Single Node architecture

Assume that it has been decided that the whole controller application should be implemented on a single computer node. Having identified what needs to be measured from the system environment, it is deduced that two ADCs and one DAC are needed. A complete model of our hardware structure is given in Figure 4a.



(a)

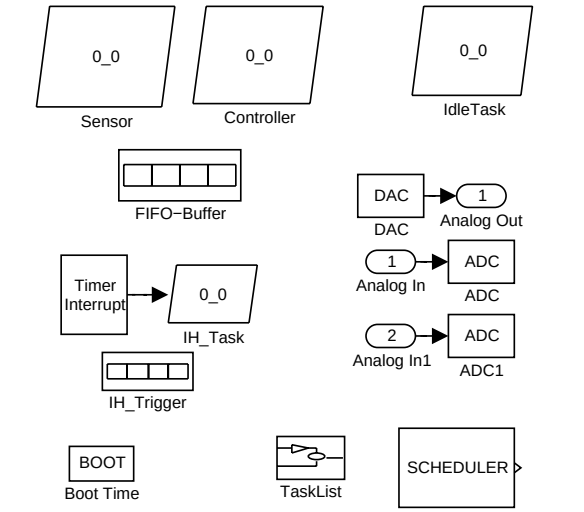


(b)

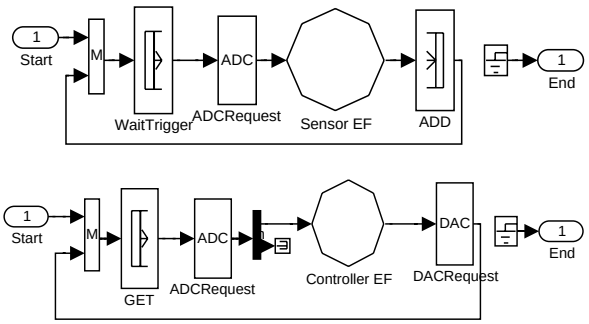
Figure 4. (a) The hardware model, showing the computer hardware and system environment. (b) Closed loop system performance where the computer-induced delays cause oscillatory behavior.

The node internal structure is shown in Figure 5a. The application is to be divided into two tasks. The first, called the Sensor task, performs the sampling of the sensor values, while the second, called the Controller task, executes the control algorithm. Data sharing between the two tasks is to be done via a FIFO, block-on-empty, inter-task communication buffer (FIFO_Buffer). The tasks' internal

structures are shown in Figure 5b. The sensor task is triggered periodically (WaitTrigger) by a system timer with a period of 2 milliseconds; the trigger is produced by an interrupt handler task (IH_Task) and communicated to the sensor task via a buffer (IH_Trigger). The controller task is triggered by the arrival of messages (the sampled sensor values) from the sensor task.



(a)



(b)

Figure 5. (a) Internal node configuration. (b) Internal task structure for the Sensor and Controller tasks.

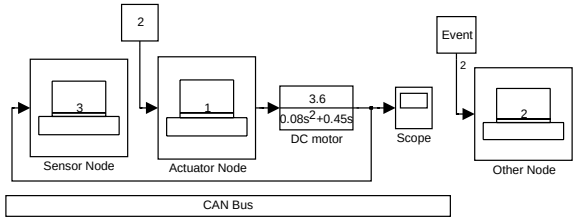
To build this model, the developer simply drags and drops the blocks from the Simulink library. The parameters of each block are then customised through a GUI. Partitioning is performed by copying the blocks from the original control model (e.g. "discrete PID" block in Figure 3a.) into the appropriate EF (e.g. "Controller EF" in Figure 5b). The execution time of each EF is then specified. Having chosen a fixed priority preemptive scheduler for this node, the user can specify the priority of each of the application tasks in the system. The task list is designed such that the priority of system tasks are automatically generated by the task list component and may not be modified by the user.

Once the model is built, a simulation is performed and output devices such as scopes can be placed in various parts of the system to monitor any data required. Figure 4b shows the motor angular position. Note the difference between this controller performance (such as overshoot, rise time and settling time) and that achieved by the pure functional model in Figure 3b.

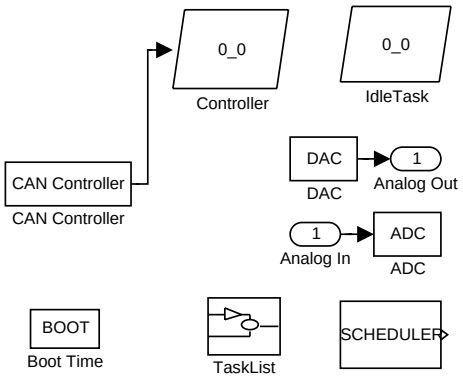
By instrumenting the model, various jitter and delay parameters were measured. For example, the motor actuation occurred 1.4ms after the sensor reading. This delay caused the change in the controller performance detected in Figure 4.

6.3. A distributed architecture

Now, assume that it is required to distribute this control application over two computer nodes. One node performs the sensor sampling, while the other node performs the motor actuation. CAN is chosen as the communication protocol between the nodes. Hence, a CAN bus is placed in the hardware model, with a CAN controller in each of the computer nodes. This architecture is shown in Figure 6a where the "Other Node" is added to introduce interference and blocking to the CAN communication.



(a)

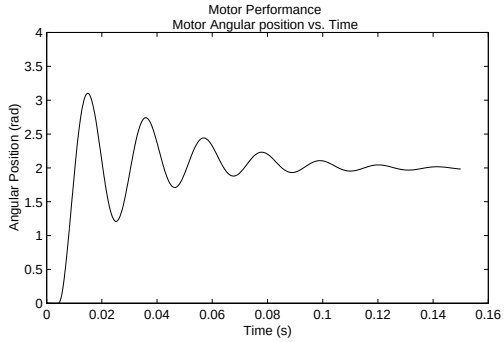


(b)

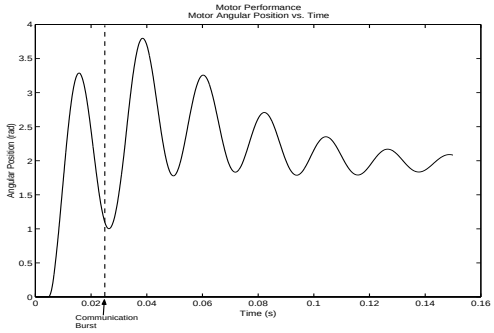
Figure 6. (a) Distributed hardware model. (b) Internal node structure for the "Actuator Node".

The internal node and task structures are essentially similar to the previous example with the difference that the Sensor and Controller tasks are placed in the Sensor and

Actuator nodes respectively. Also, the communication of the sensor samples is now performed over a communication link. Figure 6b shows the internal node structure for the "Actuator Node". The Controller task is triggered by the CAN Controller block upon receiving a message (the sensor value) from the CAN bus. In this example, the CAN bus bit rate is configured to 100kbit/s, and the sensor value is sent on the bus with an ID of three and a size of three bytes. For each CAN frame, the worst-case bit stuffing and frame overhead are assumed [17].



(a)



(b)

Figure 7. Closed loop performance for the (a) distributed architecture (b) distributed architecture, with interference at time = 25 ms.

This architecture gives the controller performance shown in Figure 7a. As expected, this is worse than that of the single node architecture since the transmission time of messages over a bus is longer than inter-task communication within a single node.

Studying the effects of interference and blocking.

Assuming the presence of other functions in the system, it may be desired to study how these affect our control system. Here, we assume that another application resides on a third node in the system and it periodically sends low priority messages (ID = 4) on the CAN bus as shown in Figure 6a. Also, at time = 25 ms, the node sends a single high priority message, (ID = 2) eight bytes in size, on the bus. How does this affect the controller application? By incorporating the third node in our model and running a simulation, the new controller performance can be investigated (Figure 7b). The arrival times of each message in the CAN controllers is shown in Figure 8.

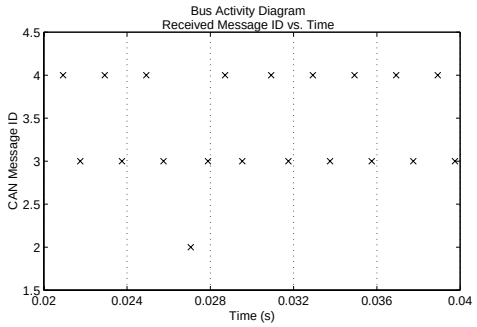


Figure 8. The arrival time of messages received by the CAN Controllers, (high priority interference message: ID=2, sensor readings: ID=3, other periodic messages: ID=4).

Here, we see that the controller performance has degraded even further, with a transient glitch, due to the delays in communicating between the two nodes, induced by the new function. Such information can be used by the system designer for example to produce limiting requirements on the communication bandwidth of other functions in the system, and/or by modifying the control system to be more robust towards (varying) time-delays.

Further Evaluation. Early in the design stages, the developer may wish to evaluate the effects on the application of varying certain parameters such as the task scheduler, inter-task communication mechanisms, or the communication protocols used. The tool allows the simple exchange of such parameters by simply replacing a particular component with another and reconfiguring it appropriately, with minimal effort from the user.

7. Future Work and Conclusion

This paper has described the current status of the models and toolset being developed at the Mechatronics Lab, as part of the AIDA project [1]. The analysis of computer induced effects (due to design faults such as timing problems and computer hardware faults) on the system behaviour, constitutes a kind of failure mode and effects analysis. We expect the simulation tool to be especially useful for evaluating different error detection as well as error handling techniques, spanning the computer and control system levels. As mentioned earlier, this work is to be extended in various directions in the future. Most

importantly, it is desired to validate and evaluate this work through further industrial case studies, such as the evaluation of architectures for x-by-wire systems in other related projects at the Mechatronics lab. Currently, no emphasis has been placed on simulation efficiency; this may require work in the future.

Parallel to the work described here, an analysis tool, aiming at the timing analysis of distributed control systems, is also under development at the lab. In the future, the two tools are to be integrated, providing two complementary approaches for system analysis. This may necessitate the extension of the current models in order to overcome any unforeseen shortcomings in the models.

Certain tool implementation limitations also need to be overcome. For example, in the current implementation, the scheduler preemption overhead is not an explicit parameter of the scheduler and need to be specified for each task in the system. From the usability perspective, it will be desired to expand the current libraries (for example, with a collection of communication protocols) and extend the tool to support features such as redundancy mechanisms and clock synchronisation. Also, it is desired to further the development of hardware entity models, such as simple processors and smart sensors, allowing the modelling of functionality implemented directly in hardware.

8. Acknowledgement

This work is partially funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research. The authors are also grateful for the valuable suggestions and reviews of the RTC group members (Ola Redell, Martin Sanfridson and DeJiu Chen) during the work described in this paper.

9. References

[1] http://www.md.kth.se/~ola/aida/index.html

[2] Audsley N., *STRESS: A Simulator for Hard Real-Time Systems*, RTRG 106, Dept. of Computer Science, University of York (1991).

[3] Bass, J.M., Brown, A.R., Hajji, M.S., Marriott, D.G., Croll, P.R.; Fleming, P.J, Automating the development of distributed control software, *IEEE Parallel & Distributed Technology: Systems & Applications Conference*, Winter 1994, pp 9-19.

[4] Bass, L. and Kazman, R., *Architecture-Based Development*, Technical Report, Carnegie Mellon University, CMU/SEI-99-TR-007, ESC-TR-99-007.

[5] Eker J. and Cervin A., A Matlab toolbox for real-time and control systems co-design, *Proc. of 6th Int. Real-Time Computing Systems and Applications Conference*, 1999, pp 320-327.

[6] Palopoli L., Abeni L., Buttazzo G. Real-Time control system analysis: an integrated approach. *Proceedings of Real-Time Systems Symposium*, 2000, pp 131-140.

[7] Gomaa H., *Software design methods for concurrent and real-time systems*. Addison-Wesley publishing company, 1993, ISBN 0-201-52577-1.

[8] Hanselmann H., Development Speed-Up for Electronic Control Systems. *Convergence-International Congress on Transportation Electronics*, October, 1998.

[9] Maier, M. W., System Architecture: An Emergent Discipline?, *IEEE Aerospace Applications Conf.*, Vol. 3, pp. 231-246, 1996.

[10] http://www.mathworks.com

[11] Papadopoulos Y., McDermid J.A., Sasse R., and Heiner G., Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in Conditions of Failure. *Reliability Engineering and System Safety*, 71(3):229-247, Elsevier Science, 2001.

[12] Ray, A. and Halevi, Y., Integrated Communication and Control Systems: Part II - Design Considerations. *ASME Journal of Dynamic Systems, Measurements and Control*, Vol 110, Dec. 1998, pp 374-381.

[13] Redell, O., *Modelling of Distributed Real-Time Control Systems, An Approach for Design and Early Analysis*, Licentiate Thesis, Department of Machine Design, KTH, 1998, TRITA-MMK 1998:9, ISSN 1400-1179, ISRN KTH/MMK--98/9--SE, Stockholm, Sweden.

[14] http://www.md.kth.se/RTC

[15] Shin, K.G. and Kim, H., Derivation and application of hard deadlines for real-time control systems. *IEEE Trans. on Systems, Man and Cybernetics*, Vol. 22/no. 6, Nov-Dec. 1992, pp 1403 -1413

[16] Storch, M.F., DRTSS: a simulation framework for complex real-time systems, *Proc. of Real-Time Technology and Applications Symposium*, 1996, pp 160-169.

[17] Tindell, K.W., Hansson, H., and Wellings, A.J., Analysing real-time communications: controller area network (CAN) *Proceedings of Real-Time Systems Symposium*, 1994, pp 259 -263

[18] Törngren, M., *Modelling and design of distributed real-time control applications*. Doctoral thesis, Dept. of Machine Design, KTH, ISSN1400-1179, ISRN KTH/MMK--95/7--SE.

[19] Törngren, M. and Fredriksson, P., *SMART-1. CAN and Redundancy logic simulation of the SMART SU*. Swedish Space Corporation, Report S80-1-SRAPP-1.

[20] Törngren, M., Elkhoury, J., Sanfridson, M. and Redell, O., *Modelling and Simulation of Embedded Computer Control Systems: Problem Formulation*. Technical report, Department of Machine Design, KTH, TRITA-MMK 2001:3, ISSN1400-1179, ISRN KTH/MMK/R--01/3--SE.

[21] Vestal, S., Integrating control and software views in a CACE/ CASE toolset. *IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, 1994, pp: 353 -358

[22] Wittenmark, B., Nilsson, J. and Törngren, M., Timing Problems in Real-time Control Systems. P*roceedings of the 1995 American Control Conference*, Seattle, WA, USA.

[23] Zhu, J., Design and simulation of hard real-time applications, *Proc. of 27th Annual Simulation Symposium*, 1994, pp 217 -225.

[24] Åström, K. J. and Wittenmark B., *Computer-Controlled Systems, Theory and Design*. Second edition, Prentice-Hall International, inc, 1990, ISBN 0-13-172784-2.

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **36,** 66–80 (1996)
ARTICLE NO. 0090

An Overview of RealTimeTalk, a Design Framework for Real-Time Systems

CHRISTER ERIKSSON,* JUKKA MÄKI-TURJA,* KJELL POST,* MIKAEL GUSTAFSSON,* JAN GUSTAFSSON,* KRISTIAN SANDSTRÖM,* AND ELLUS BRORSSON†,[1]

**Department of Computer Engineering, Mälardalen University, P.O. Box 883, S-721 73 Västerås, Sweden; and †Department of Computer Engineering, Dalarna University College, P.O. Box 10044, S-781 10 Borlänge, Sweden*

RealTimeTalk (RTT) is a design framework for developing distributed real-time applications with both hard and soft requirements. The framework supports design via hierarchical decomposition. We believe that object-orientation is the best way to go about structuring a problem, hence the RTT language is based on Smalltalk with an analysis frontend to infer type information for run-time safety, and to yield more precise estimations of execution times. Unlike most real-time systems, RTT does not force the designer to embed constructs for timing requirements, communication, and synchronization in the code. Rather, such information is specified on a higher level of abstraction using graphical tools. This not only keeps the code "clean" but also simplifies timing analysis and resource allocation. A comparison with other real-time systems concludes the paper.

1. INTRODUCTION

A current trend today is the replacement of mechanical and/or electromechanical control systems, as found in cars and nuclear power plants, with computer-based systems. This move is explained primarily by the reduced production costs and increased functionality and flexibility of software systems. Still, a computer based solution must be at least as dependable as the replaced solution.

Another observable trend is the ever-increasing complexity of computer based systems, a consequence of today's demands on functionality and distribution. In the early days, one computer performed one function, then gradually more and more functions were added to each computer. Today, we see applications requiring a function to be mapped over several computers. Performance and safety issues, as well as geographic reasons, are incentives behind such a distribution of control. These systems are often referred to as real-time systems with the meaning that "the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced" [21].

[1] E-mail: {cen,jma,kpt,mgn,jgn,ksm}@mdh.se; jbr@t.hfb.se.

In the real-time research community, one often distinguishes between *hard* and *soft* temporal requirements. If a deadline is violated in a hard real-time computation the logical result is useless. In a soft real-time computation, the logical result could have a meaning even though a deadline is not met. Many real-time applications of today have both hard and soft temporal requirements. For example, in the computer system of a car, a hard requirement involves controlling the brakes, whereas the automatic climate control is an example of a soft requirement.

Real-time applications often have two types of requirements on control: *periodic* and *aperiodic*. Periodic control means that an activity is computed repeatedly with a predefined period time, e.g., implementing a cruise control for a car leads to a periodic activity. Aperiodic control means that the application generates events that the computer system must respond to within a certain time, e.g., the brake system in a car. Periodic and aperiodic activities can be either soft or hard.

An important aspect in the design of real-time systems is the integration of methods, architectures, and tools to avoid inconsistency between design and implementation. It is not unusual to find people using a particular method in the design phase of some new system, then working with another tool for the implementation, violating specifications set during the design phase. A classical example of this dilemma is structured analysis and design which aids the specification of the system without considering the run-time environment. With the proper integration of design specification and implementation tools, specifications made in the design phase could be enforced throughout the entire life-cycle of the product.

This paper describes RealTimeTalk (RTT), a framework for the development of applications with both hard and soft real-time requirements. The organization of the paper is as follows. Section 2 highlights the features of RTT; Section 3 presents the programming-in-the-large aspects of RTT, followed by a presentation of the RTT language in Section 4. Section 5 describes the run-time environment; Section 6 describes the prototype system; Section 7 con-

0743-7315/96 $18.00

tains a short description of related work. We conclude the paper in Section 8.

2. INTRODUCTION TO RTT

The main goal of RTT is to simplify the design and implementation of predictable real-time systems. We strongly believe that object-oriented methods are of vital importance in this process. Object-orientation gives advantages such as reusability, rapid prototyping, incremental development, modularity, extensibility, and promotes the use of frameworks. However, notions for synchronization and distribution are often weak.

A number of desirable properties for the development of real-time systems can be identified (cf. [6]). For instance, frameworks which support hierarchical decomposition fosters good design and helps the designer to cope with complexity. Component based design promotes reusability and safety. The transformation between analysis, design, and implementation should be reversible in order to maintain the consistency between the different models when the system is maintained and extended. Within the framework, it should be possible to both specify and verify the functional and temporal aspects of the system. Ideally, a common description language for different levels of abstraction should be used. This language should support incremental development, prototyping and simulation, error-handling mechanisms, and the possibility of encapsulating low level languages, for reasons of efficiency and code re-use.

The *design framework* for RTT is divided into parts for programming in the large and programming in the small. Programming in the large corresponds to application design on a higher level and contains tools to map a design to a resource structure, whereas programming in the small concerns implementation of classes.

The syntax of the RTT programming language is based on Smalltalk [8], a true object-oriented language with simple syntax and semantics. Recently, Smalltalk has begun to regain some of the ground it lost after C++'s appearance. We believe this trend is mainly due to Smalltalk's simplistic nature and its support for rapid prototyping, making it very easy to focus on the problem and to quickly build high quality applications without having to worry about, e.g., memory management—a nontrivial problem in most C++ applications.

However, the nondeterministic nature of garbage collection and method invocation, together with the weak type system makes Smalltalk less suited for real-time applications. The language also has weak support for communication and synchronization in a distributed environment. In our adaptation of Smalltalk, we have modified these drawbacks to make RTT type safe, predictable, and usable in a distributed environment.

The RTT design framework also separates the specification of hard and soft real-time into two parts. It also includes the interface between these two parts. The software model for the design of the hard real-time part of an application is based on a set of design objects where hierarchical decomposition is a key concept. The soft real-time part is currently open for any reactive model that conforms to the interface of the hard real-time part and is adaptable to the RTT run-time system.

In the hard real-time part, we have separated the functional and temporal behavior of the system. The functional requirements are specified with the RTT design objects and are checked by prototyping and testing. The temporal constraints are also specified within the RTT design objects and is statically verified by the RTT pre-run-time scheduler. The maximum calculated execution time (MAXTC) is calculated for each schedulable entity (task) by the RTT compiler. The timing information is then fed to the scheduler which tries to find a feasible schedule for the system. If such a schedule is found the temporal behavior will be guaranteed during run-time.

The run-time environment consists of a set of nodes which are connected by a predictable broadcast bus. The RTT software platform runs on each node and is divided into two parts: an operating system and a communication system. The software platform supports execution of time-triggered hard real-time tasks and event-triggered soft real-time tasks. Furthermore, the communication system supports both hard and soft real-time messages.

3. PROGRAMMING IN THE LARGE

3.1. Introduction

When designing object-oriented systems today, one often ends up with a monolithic structure where an ocean of objects collaborate to achieve a desired function. Needless to say, maintenance of such nonhierarchical implementations are difficult at best and requires understanding of the system on all levels. It also leads to problems for the customer, who has to verify the functionality of the actual implementation against the requirements specification. Of course, many systems are described on different levels of abstraction during the design, but these abstractions are seldom explicit in the implementation. Another issue is how to handle synchronization and communication in an object-oriented system. Most programming models integrate synchronization and communication with the functionality. For example, when using a real-time operating system, tasks typically have synchronization calls interleaved with the rest of the code, making it difficult to verify and maintain the code.

Furthermore, when implementing time constraint services with strict deadlines in a conventional system, time constraints are often mapped to period times or priorities of individual tasks. This approach is often usable when constructing simple single node systems, but when implementing time constraint functions which include several tasks (and where the tasks might execute on different nodes) these solutions are not adequate. To implement these functions, the time constraints must be specified for

the complete function in an explicit way to make the system predictable.

Implementing a time constrained service can be done either by the *time-triggered* or *event-triggered* approach. The time-triggered execution model is defined as follows: the system observes the state of the environment at specific points in time. Thereafter the system decides, based on an analysis of the state, which actions must be taken. Afterwards, new values are emitted to the environment at a predefined point in time. By event-triggered, we mean that an event occurrence propagates into the control system at an arbitrary point in time. For example, an interreupt normally initiates an action, such as resuming a task waiting for the event.

When implementing hard real-time functions it is very important to be able to verify the function easily. With the time-triggered approach this is more straightforward, because the number of states to test is much smaller than for the event-triggered approach [15]. The reason for this is that the events in the environment can only propagate into the system at predefined points in time, or more specific, in predefined time intervals. In the event-triggered model, events can propagate into the system at arbitrary points in time and thus the temporal behavior will be more difficult to verify compared to the time-triggered approach. As a consequence, the time-triggered approach is superior to the event-triggered approach when implementing control loops and monitoring functions. On the other hand, when implementing functions that are inherently event-triggered, the transformation to a time-triggered model is often nontrivial and the transformed solution does not reflect the structure of the function. Thus, when implementing such functions the event-triggered approach is more suited. Furthermore there are powerful, commercially available tools for modelling event-triggered client-server applications, for instance user interfaces. We conclude that there is a need for both of these approaches, because one could not choose an approach without considering the controlled application.

RTT is well suited for time-triggered hard real-time applications, and periodic or event-triggered soft real-time applications.

3.2. Application Example

The following example, in which we imagine a railroad network supervised by a distributed computer system serves to illustrate various aspects of the RTT software model.

The railroad network is divided into segments (Fig. 1). Segments can be connected in either end to form loops and junctions. Each segment consist of a straight sequence of zones, each zone corresponding to a fixed amount of rail. Trains in a segment are managed by a segment computer. Trains are considered dumb in that respect that the segment computer dictates their speed. Segment computers need to negotiate on in- and outgoing trains and are therefore linked to each other. A central computer (henceforth referred to as the traffic area controller, TAC) collects information from each segment computer to construct an overall plan. This plan is produced continuously to make efficient use of the railroad network. The segment computers attempts to follow this plan, but will of course deviate from it, should there be a chance of collision. It should be stressed that segment computers are in no way dependent on the TAC for safety issues.

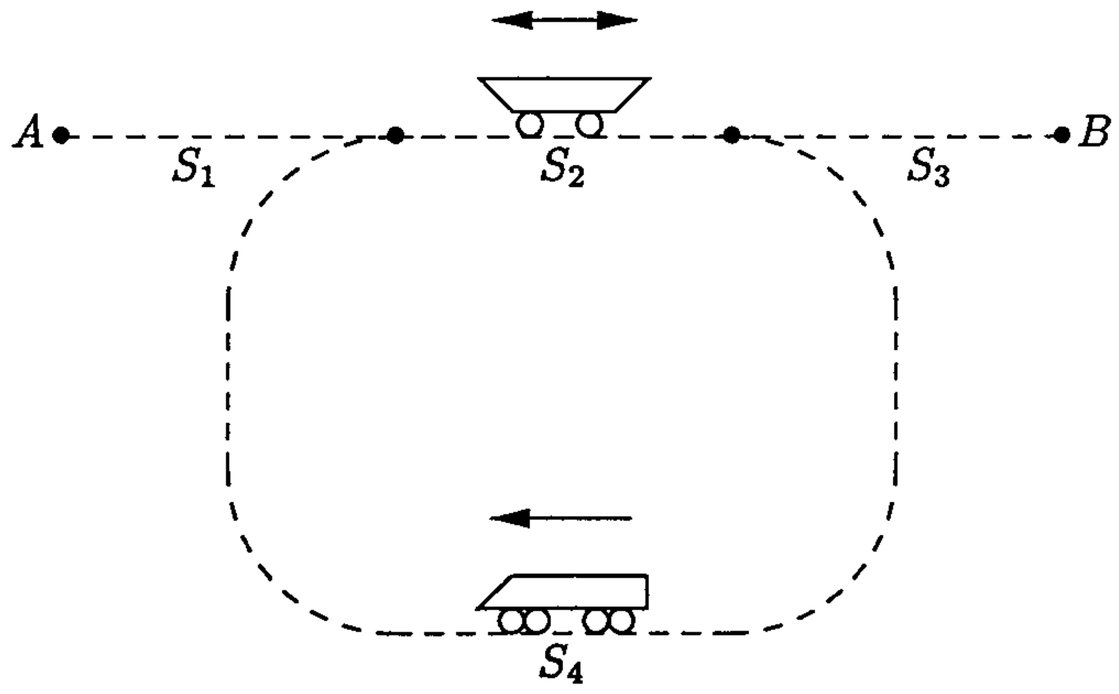


FIG. 1. Railroad network. Each S_i denotes a segment; dashes correspond to zones.

A plan is based on a "snapshot" of the system and describes desired speed settings for each train until the next plan is constructed. After the plan is formed (which may take some time), a new snapshot is taken to verify that the plan is still consistent. If so, the plan is downloaded to the segment computers, otherwise it is discarded. Should the central computer face a deadlock, the system stops and an operator have to resolve the situation.

This example will be used in the remainder of this paper to illustrate various parts of the RTT framework.

3.3. The Hard Real-Time Model

A key concept in the design of hard real-time applications is the use of different levels of abstractions. In RTT, design is supported by hierarchical decomposition and the use of a predefined set of design objects. These design objects are *modes*, *mode transitions*, *use-cases*, and *tasks*. Figure 2 illustrates a generic application structure.

3.3.1. Modes and Mode Transitions. When developing a system one often realizes that the modelled system has several distinct states, or *modes*. For a locomotive, these modes could be notOperating, manuallyOperating, and autopilot. The fact that there exist distinct modes does not mean that the functionality of these modes are disjoint. For instance, it is reasonable to assume that the modes manualOperation and autopilot have some common functionality. When the different modes of a system have been identified one has to model the transitions between them. In the train example, two of the mode transitions could be ManualToStopped and AutoToManual.

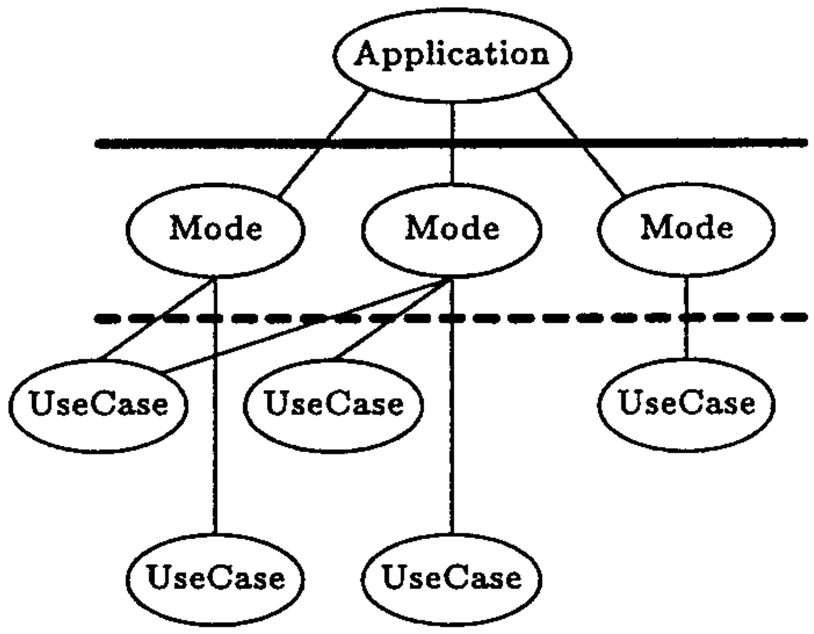


FIG. 2. The hierarchical structure of a generic application.

A mode defines the activities that must take part during a particular state of the application. A mode transition corresponds to actions taken when we have a change of mode in the system. Thus a mode is mapped to a continuous schedule whereas a mode transition is mapped to a one-shot schedule.

The problem with implementing modes and mode transitions is that most systems does not have an explicit notation for modes. Instead, modes and mode transitions are usually embedded in the code. If the model supports distinct modes one can easily see which functions that run in each mode and thereby allocate resources in an efficient way. Another advantage is that each mode could be designed separately. In the RTT model, a precise notation and semantics for modes and mode transitions are supported by the design objects Mode and ModeTransition. The transitions between the different modes are described as a high level state machine. For instance, in Fig. 3, we see such a graph for the train application.

In RTT each activity in a mode and mode transition is modelled by a use-case.

3.3.2. Use-Cases. A use-case can be seen as an engine that controls a number of cooperating objects to perform a certain task. A use-case models an activity as a periodic function. An aperiodic activity is translated to a periodic activity, for example, with the theory provided by Mok [18].

When implementing a use-case that consists of a several collaborating computational entities, it is important to have an explicit notation for both synchronization and communication, as well as being able to describe both sequential and parallel executions. With synchronization we mean the order in which the different entities should be executed. Furthermore, it should also be possible to describe that some resource is shared and that the access of this resource is protected. In many systems these mechanisms are, as mentioned before, embedded in the implementation which make them hard to understand and maintain.

With communication we mean the information that is exchanged between the different entities. This information should also be explicit for the same reasons as for synchronization. RTT models both *synchronization* and *communication* with a precise syntax and semantics, at a high level of abstraction.

A use-case is defined by a *precedence graph*, an *interaction graph*, and the *period time* of the computation. A precedence graph is a directed acyclic graph and defines the execution order between the entities. The interaction graph specifies the communication between the entities and the use of shared resources. The period time specifies the time between two consecutive activations of the entities specified in the precedence graph. These entities are called tasks.

3.3.3. Tasks. A task encapsulates an object and provides the thread of control and communication for the object. As just mentioned, the model does not mandate synchronization and communication constructions in the code. Rather, we have equipped the tasks with in- and out-ports to decouple the code from communication details. The synchronization is decoupled from the code by explicitly specifying it in a precedence graph. Let us first describe the operation of a task, then its temporal attributes.

Assume an object *S* with a method `m:` and an object *C* who wants to send the message `m: arg` to *S* (Fig. 4). *C* could potentially reside on a different node in the system. As seen in Fig. 4, the object *S* and its method are encapsulated by a task. The role of the task is to (1) map the in-port of the task to the arguments of the method; (2) invoke the method `m` in the object; and (3) map the result of the method's execution to the out-ports.

The temporal attributes of the task are *release time*, *deadline*, *MAXTC* (maximum calculated execution time), and *MINTC* (minimum calculated execution time). These temporal attributes can be divided into two groups. The

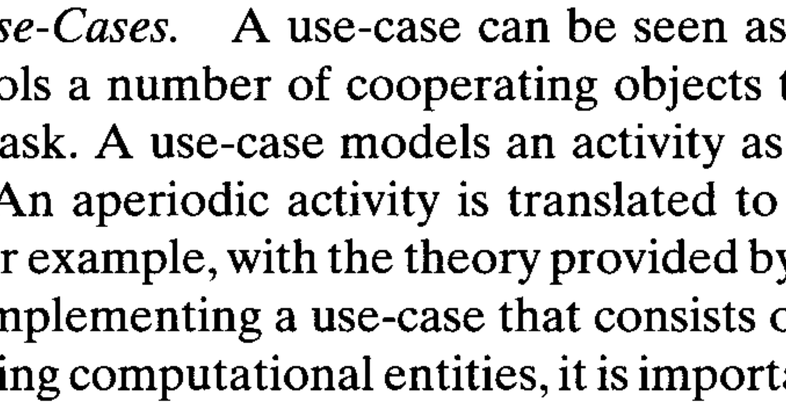


FIG. 3. A high-level state machine for the train example.

Task

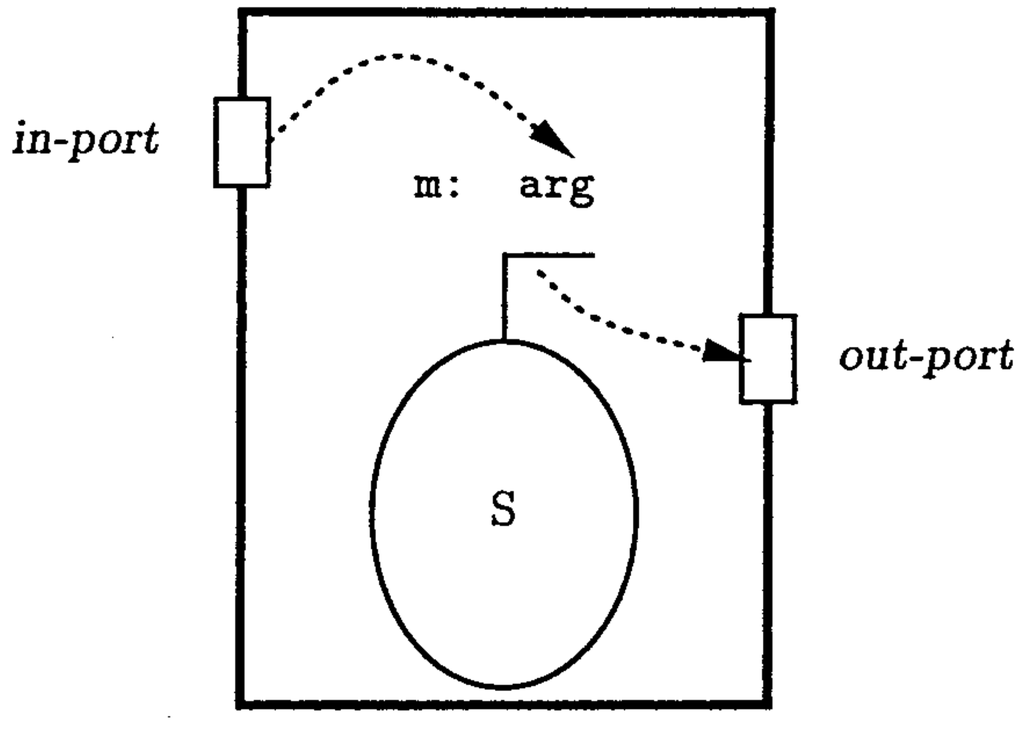


FIG. 4. The structure of the task that encapsulates the object *S*.

first group, involving MAXTC and MINTC are static, meaning that these parameters do not depend on the use-case the task is part of. Release time and deadline, on the other hand, are context dependent; i.e., they depend on the temporal requirements of the use-case in which they are defined. The release time and deadline are set relative to the period time of the specific use-case. The following relation defines constraints on the release time and deadline:

$$(\text{deadline} \leq \text{period time}) \wedge (\text{release time} \geq 0) \wedge (\text{deadline} \geq \text{release time} + \text{MAXTC}).$$

A complete specification of a task include the object that it encapsulates, the activation method of the encapsulated object, the node the task will execute on, the in- and out-ports, and the four temporal attributes just mentioned. In RTT, a port is either a primitive object—such as a boolean, integer, character, float or string—or a compound object, having primitive objects as instance variables.

3.3.4. Precedence Graphs and Interaction Graphs. For some activities, it is very important that input and output with the environment is synchronized. For example, in the train example it is imperative that each snapshot of the trains' position is taken within a certain time interval by each segment computer, or otherwise an inconsistent view of the state may result. It is also desirable to be able to describe the execution order of the tasks.

In order to specify synchronization between tasks on a higher level of abstraction, we have introduced precedence graphs. A precedence graph is a directed acyclic graph which specifies the precedence relationship between tasks. For example, if task A precedes task B in the graph, task A must terminate its execution before task B can start.

To specify communication between tasks, the *interaction graph* is introduced. The interaction graph is a binary relationship specifying pairs of tasks that can communicate with each other, i.e., where the producer's out-port is connected to the consumer's in-port. The interaction graph also specifies the use of shared resources.

These two graphs are often merged into one graph. For example, in the MARS system [16] the precedence relationship also includes data transfer. In our opinion, this requirement is too strong and makes it difficult to describe things like feedback loops. The reason behind our separation of synchronization and communication is that these two issues really are orthogonal concepts; i.e., communication may or may not be synchronized. Figure 5 shows a precedence graph for the use-case in the train example that takes snapshots of the state of the environment. S_n is the task that encapsulates the segment object at segment computer n. The idea is that each segment computer is supposed to take a snapshot of the trains' position at roughly the same time. When the snapshot has been taken the information is sent to the TAC task which encapsulates the traffic area controller object.

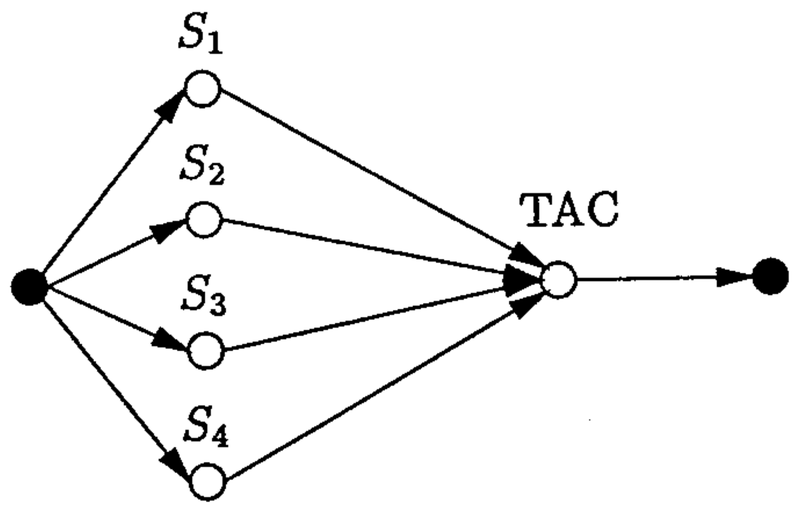


FIG. 5. The precedence graph for the use-case snapshot.

The corresponding interaction graph is shown in Fig. 6. In this example, the communication is synchronized; i.e., each precedence relation has an associated interaction relation.

The precedence graph, the period time of the use-case, and the tasks' temporal attributes are related, as seen in the following example.

EXAMPLE 3.1. The time between two consecutive activations of the tasks in the precedence graph is specified by the period time of the use-case. Assume that a period time of 500 ms has been derived from the requirements specification to obtain the needed observation frequency. The precision on each observation requires that the observation of each segment object is done within 100 ms. The precision requirement can be specified by the release time and deadline attribute of each task. In this case, the release time could be specified as 0 ms for the segment tasks ($S1$ ... $S4$). The deadlines for each segment task then has to be specified to 100 ms to fulfill the precision requirement.

In many applications, sensors are read with a much higher frequency than the control loop executes to enable signal processing [25]. Therefore it must be possible to specify the communication between a task in the control loop's use-case and the producing signal's processing task. This is specified in the same way as for tasks communicating within one use-case, i.e., by connecting the producing task's out-port to the consumer's in-port. There is no difference between these two cases from the modelling point of view but the translation to a resource structure will be different. This translation will be described in Section 6.5.1.

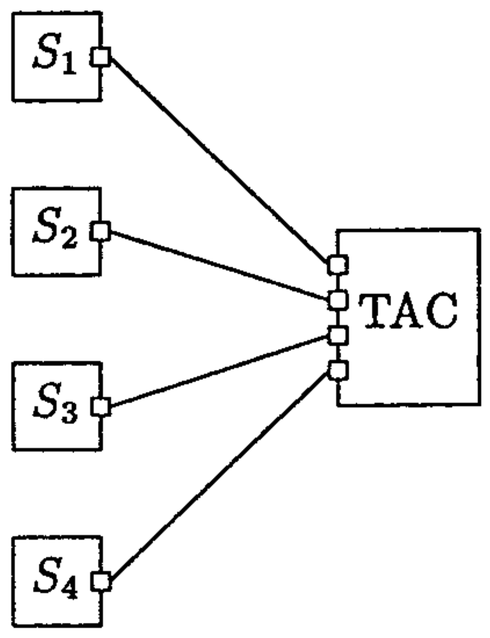


FIG. 6. The interaction graph for the use-case snapshot.

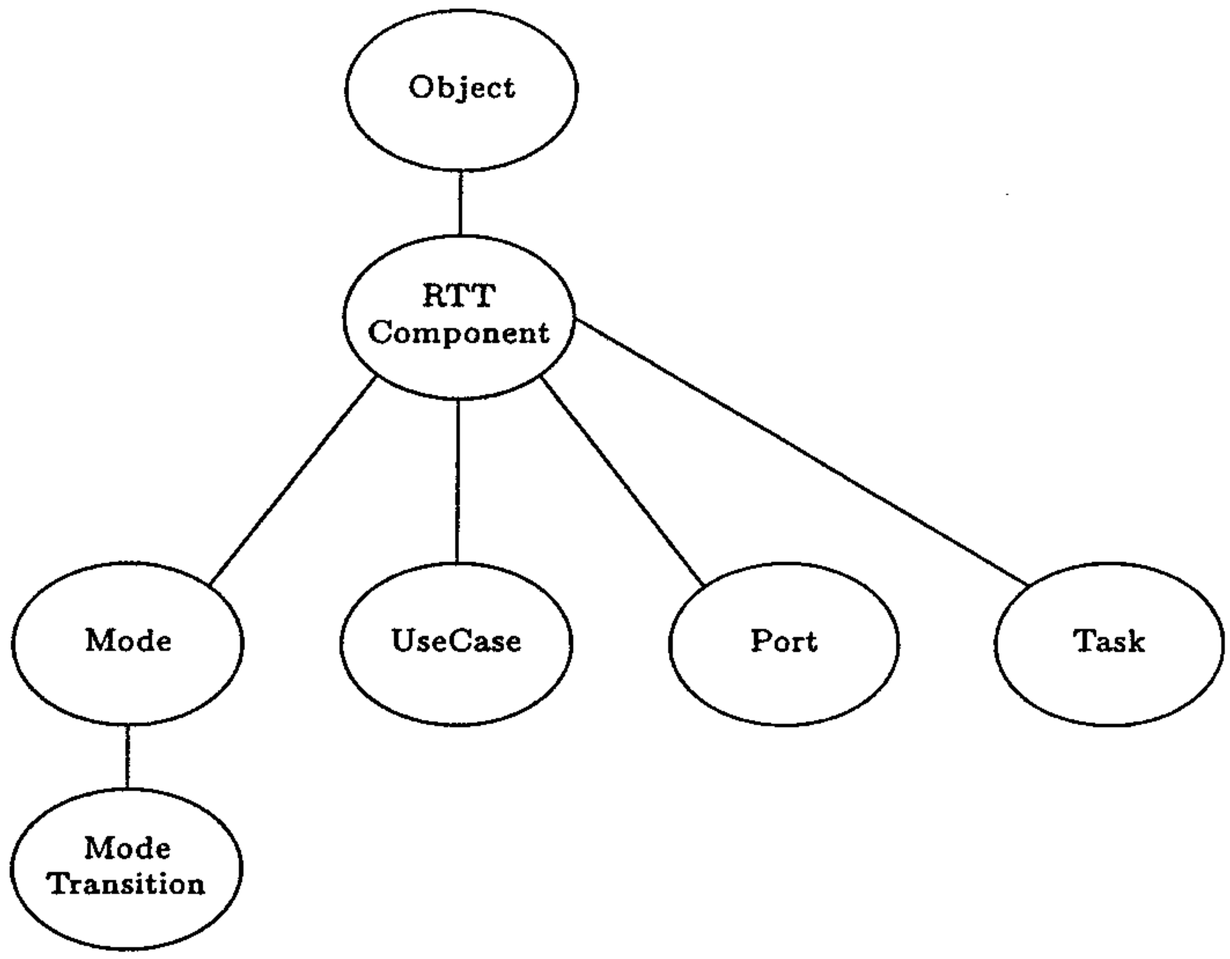


FIG. 7. Class hierarchy.

3.3.5. Supporting Class Hierarchy. The RTT class hierarchy supports objects such as modes, mode transitions, use-cases, and ports. An application engineer uses these classes when developing his system. Furthermore these objects are mapped to the resource structure to reflect the design; i.e., one can easily follow the high-level specification all the way down to implementation. An example of such a hierarchy can be seen in Fig. 7.

3.4. The Soft Real-Time Model

The reason for introducing a soft real-time model is that most applications of a reasonable size have both hard and soft functionality. The question is: why not implement the soft functions as hard? There are several reasons:

- If resources for a soft function is allocated pre-runtime, the period time and the maximum execution time must be considered. If a short response time is required for an event, the period time must be relatively short. This, however, gives a poor utilization if the event occurs seldom. On the other hand, if one would like to increase the utilization the period time must be increased but then the responsiveness will decrease. If the function instead is implemented in the soft real-time part the assumption is that the system in the average case will have an acceptable response time. However, one should keep in mind that no guarantee is provided.
- Sometimes it is very difficult to model a inherently event-triggered function in a time-triggered model.
- If the hard and soft functions are implemented in separate parts, the effort that must be spent on verification and validation of timing requirements will decrease. Many hard real-time functions are also safety critical and if the safety critical functions are separated from the rest of the system, the verification and validation of the safety requirements will be simpler [17].

Another reason for having both hard and soft capabilities is that this can be used to give the hard part an increased flexibility. For instance, making on-line modifications to a system, e.g., installing a new schedule and modified code. When the soft part has loaded the alterations to the system, the hard part would be informed and a change to the new schedule could be made at an appropriate time. This functionality would be very expensive to implement with just the hard part because, as mentioned earlier, the resources to accomplish this have to be pre-allocated and this functionality will be used seldom.

In the case, where the hard and soft part has to cooperate, i.e., exchange information and/or control, there must be clear definitions how to perform this cooperation without jeopardizing the temporal requirements for the hard real-time part and the consistency of data for both the hard and the soft parts. The information that is passed between the soft and hard parts is, as defined earlier, residing in objects. The responsibility of the interface could either be delegated to each object that holds the information, or to special interface objects, acting as a fire wall between the two parts. The benefit with mapping the interface direct to objects is that the object-oriented paradigm is not suppressed. The disadvantage of this approach is that failures in the soft part more easily could intrude on the hard real-time part. If special interface objects are used instead, the object-oriented approach could not be fully utilized because shared information has to be spread out on several objects. The advantage is that the separation of the two parts are distinct.

Independent of which model is adopted, data consistency must be maintained in both directions. How to achieve this has been investigated by, for example, Thijssen *et al.* [19].

Depending on the application to develop both approaches could be considered. If there are high demands

on safety the fire wall approach should be used. On the other hand if the application does not have high demands on safety but instead have a high structural complexity the interface on the object level is preferred to minimize the suppression of the object-oriented paradigm. In RTT, both these approaches are available so the developer could choose the appropriate one.

4. PROGRAMMING IN THE SMALL—THE RTT LANGUAGE

4.1. Introduction

The RTT language is primarily targeted toward the code for the hard real-time part and we will discuss the language from this point of view. Naturally one could use RTT also for the soft real-time part, without the restrictions discussed in this section.

A feature brought over from the Smalltalk community is the notion of frameworks, i.e., predefined class hierarchies with support for different application areas. This will speed up the development of applications and promote high quality code and reuse.

An underlying design philosophy in RTT is to let the designer produce a prototype within a Smalltalk development environment, and with limited concern for temporal aspects. The prototype can be implemented and modified quickly. This prototyping environment helps the programmer to focus on a reasonable functional solution to the problem at hand. Later, when the designer is satisfied, the code is fed to the RTT compiler which provides information about execution times for each task to the RTT scheduler, which in turn tries to find a feasible schedule.

4.2. Syntax and Semantics

Although the syntax of RTT is the same as of Smalltalk, the semantics of some programming constructs have been changed to make applications predictable:

1. Recursion is not allowed because of the problem of determining the recursive depth of a data dependent recursion [9].

2. Loops have to be bounded. Constructs in Smalltalk like

```
[...] whileTrue: [...].
```

with no upper bound on the number of iterations have in RTT been replaced with

```
[...] whileTrue: [...] maxIterations: m.
```

These and other similar constructs are defined in RTT and may not be changed by the application programmer. For a detailed description, please see [9].

3. Dynamic creation or changes of classes and methods are not allowed at run-time. This is necessary to guarantee time determinism of applications.

4. Data structures have to be limited in size; e.g., linked lists of indefinite length are ruled out.

4.3. The Effect of Polymorphism

RTT shares Smalltalk's dynamic typing; i.e., types are associated with values rather than variables. Dynamic typing is generally considered more flexible than static typing and also relives the programmer from having to declare the types of variables; this is a benefit, especially during the prototyping phase.

However, because of the lack of typing information at compile time, the compiler must make pessimistic assumptions regarding which method will be invoked for a certain message sent at run-time. This leads to an over-estimation in the calculation of the MAXTCs. A more serious problem is the risk of getting "message not understood" at run-time; i.e., the receiving object does not implement a method for the message. This is of course disastrous in a hard real-time system.

One ad hoc way of dealing with this problem is to systematically rename methods so that it will always be clear by looking at the program text which method will be invoked for a particular message. This, however, is against the very idea of polymorphism. Another alternative is to obtain information about which class the receiver of the message is an instance of. This can be arranged by letting the programmer provide type declarations. However, we would like to be able to use as many Smalltalk programs as possible "as they are," keeping manual conversion chores to a minimum. We also believe that type declarations are a burden to the programmer, especially during the prototyping phase.

Instead, we are currently in the process of designing a type inference system for RTT [10]. The purpose of this system is to annotate each variable occurrence in the program with a type. We define a type as a set of class names $\{C_1, ..., C_k\}$ that represents the classes that a variable occurrence can be an instance of at run-time.

With type inference, RTT can be used "typeless" in the prototyping phase. Later, when the product is ready to run, type inference is used to detect type errors and optimize the execution (Fig. 8).

With type information at hand, the compiler can

- *Reduce over-estimations of MAXTC calculations.* When a receiver is known to be an instance of a set of classes, the MAXTC calculation is limited to these classes.
- *Guarantee statically type-safe programs.* Once types are inferred, the compiler can verify statically that all messages will be understood at run-time and that arguments

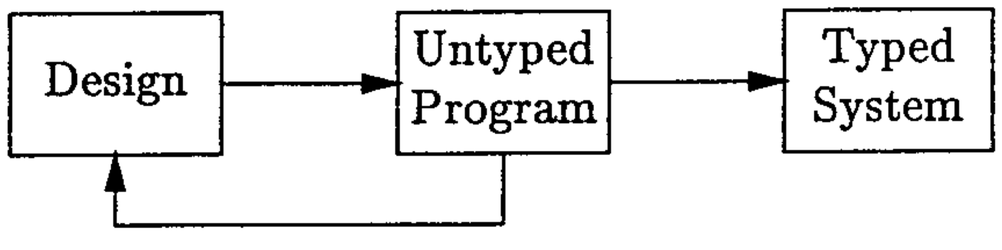


FIG. 8. Type inference in RTT program development.

on safety the fire wall approach should be used. On the other hand if the application does not have high demands on safety but instead have a high structural complexity the interface on the object level is preferred to minimize the suppression of the object-oriented paradigm. In RTT, both these approaches are available so the developer could choose the appropriate one.

4. PROGRAMMING IN THE SMALL—THE RTT LANGUAGE

4.1. Introduction

The RTT language is primarily targeted toward the code for the hard real-time part and we will discuss the language from this point of view. Naturally one could use RTT also for the soft real-time part, without the restrictions discussed in this section.

A feature brought over from the Smalltalk community is the notion of frameworks, i.e., predefined class hierarchies with support for different application areas. This will speed up the development of applications and promote high quality code and reuse.

An underlying design philosophy in RTT is to let the designer produce a prototype within a Smalltalk development environment, and with limited concern for temporal aspects. The prototype can be implemented and modified quickly. This prototyping environment helps the programmer to focus on a reasonable functional solution to the problem at hand. Later, when the designer is satisfied, the code is fed to the RTT compiler which provides information about execution times for each task to the RTT scheduler, which in turn tries to find a feasible schedule.

4.2. Syntax and Semantics

Although the syntax of RTT is the same as of Smalltalk, the semantics of some programming constructs have been changed to make applications predictable:

1. Recursion is not allowed because of the problem of determining the recursive depth of a data dependent recursion [9].

2. Loops have to be bounded. Constructs in Smalltalk like

```
[...] whileTrue: [...].
```

with no upper bound on the number of iterations have in RTT been replaced with

```
[...] whileTrue: [...] maxIterations: m.
```

These and other similar constructs are defined in RTT and may not be changed by the application programmer. For a detailed description, please see [9].

3. Dynamic creation or changes of classes and methods are not allowed at run-time. This is necessary to guarantee time determinism of applications.

4. Data structures have to be limited in size; e.g., linked lists of indefinite length are ruled out.

4.3. The Effect of Polymorphism

RTT shares Smalltalk's dynamic typing; i.e., types are associated with values rather than variables. Dynamic typing is generally considered more flexible than static typing and also relives the programmer from having to declare the types of variables; this is a benefit, especially during the prototyping phase.

However, because of the lack of typing information at compile time, the compiler must make pessimistic assumptions regarding which method will be invoked for a certain message sent at run-time. This leads to an over-estimation in the calculation of the MAXTCs. A more serious problem is the risk of getting "message not understood" at run-time; i.e., the receiving object does not implement a method for the message. This is of course disastrous in a hard real-time system.

One ad hoc way of dealing with this problem is to systematically rename methods so that it will always be clear by looking at the program text which method will be invoked for a particular message. This, however, is against the very idea of polymorphism. Another alternative is to obtain information about which class the receiver of the message is an instance of. This can be arranged by letting the programmer provide type declarations. However, we would like to be able to use as many Smalltalk programs as possible "as they are," keeping manual conversion chores to a minimum. We also believe that type declarations are a burden to the programmer, especially during the prototyping phase.

Instead, we are currently in the process of designing a type inference system for RTT [10]. The purpose of this system is to annotate each variable occurrence in the program with a type. We define a type as a set of class names $\{C_1, ..., C_k\}$ that represents the classes that a variable occurrence can be an instance of at run-time.

With type inference, RTT can be used "typeless" in the prototyping phase. Later, when the product is ready to run, type inference is used to detect type errors and optimize the execution (Fig. 8).

With type information at hand, the compiler can

- *Reduce over-estimations of MAXTC calculations.* When a receiver is known to be an instance of a set of classes, the MAXTC calculation is limited to these classes.
- *Guarantee statically type-safe programs.* Once types are inferred, the compiler can verify statically that all messages will be understood at run-time and that arguments

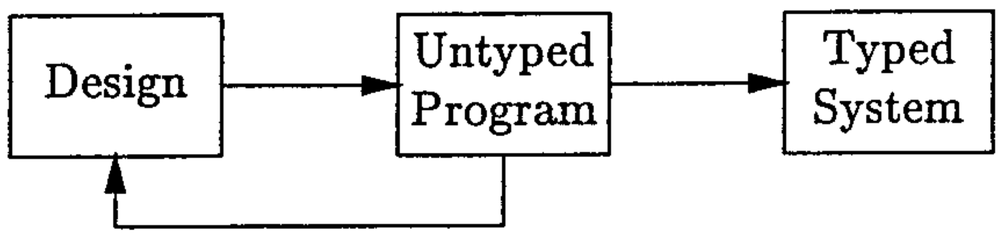


FIG. 8. Type inference in RTT program development.

to system builtins have the proper type. Of course, the compiler may be overly pessimistic and reject a program that would avoid an offending construct at execution.

• *Eliminate run-time type checking.* When programs are type-safe, there is no need for builtin primitives to check the type of arguments before using them. This will reduce run-time overhead.

• *Produce more efficient code.* When a receiver is known to be an instance of one class only, method lookup can be replaced with a function call. This in turn enables other optimizations, e.g., inlining. Efficiency can sometimes also be improved even if the receiver could be an instance of more than one class [10].

• *Characterize polymorphic recursion.* In object-oriented languages, recursion has a deeper meaning, compared to other languages, since a function (a method) is identified not only by the name of the function, but also by the receiving object. Suppose an instance I sends a message m. Then we can differentiate between three kinds of recursion, in ascending order of generality:

—Instance recursion: when the execution of m invokes the method for m in I again (perhaps transitively).

—Class recursion: when the execution of m invokes a method for m in an instance of the same class as I.

—Polymorphic recursion: when the execution of m invokes a method for m in some other instance (perhaps I). Thus, polymorphic recursion may not be recursive at all!

With type information, polymorphic recursion can sometimes be classified as non-recursive and therefore be allowed in RTT.

5. PRINCIPLES OF THE RUN-TIME ENVIRONMENT

5.1. Introduction

The run-time environment provides a platform for running RTT applications and consists of a set of nodes that are interconnected via a communication network.

The run-time environment contains both strictly hardware contained functions as well as software functions. However, many functions is a result of cooperation between hardware and software. In addition, a lot of the functions could either be implemented directly in hardware or software. Therefore, we do not separate the run-time environment into a hardware and a software part.

Most hardware components used in run-time environments today are designed to be used in systems with the goal of maximizing the average performance. As observed by Stankovic [22] these components have shortcomings when used in hard real-time hardware architectures. Therefore when designing efficient hard real-time run-time environments a lot of effort has to be spent on evaluating the predictability and the dynamics of existing components, design of new components, and study how the components fit together.

5.2. Functionality of the Run-Time Environment

The platform has to support execution of hard real-time modes according to a pre-run-time generated schedule. This schedule could consist of a number of subschedules, e.g., one subschedule for each node and one for the communication network. These subschedules have to run synchronized to behave as one system schedule. This requires that the clocks on each node are synchronized with a known accuracy. To minimize the damage in case of a timing fault, the deadlines of the use-cases in the current mode must be supervised.

Since RTT applications not only consist of hard real-time functionality the run-time environment has to provide a base for running the soft part of an RTT application. The soft part of an RTT application is as mentioned before scheduled on-line and should fulfill the timing requirements on a best effort basis. This feature should by no means jeopardize the temporal behavior of the hard real-time part of an RTT application.

The RTT framework put no demands on the topology of the communication network. For the hard real-time part of an RTT application, the communication services must support bounded end to end message transfer times. In analogy with the execution of an application there must exist communication services for soft real-time messages which does not interfere with hard real-time messages.

In object-oriented systems, as mentioned earlier, there are a number of basic features that could give variance in program execution and thus would give a poor utilization of the hardware resource. The two features which contributes the most to poor utilization are garbage collection and method dispatching. To be able to run real-time object-oriented applications, the run-time environment must have implementations of these features that are both predictable and have low variance in execution time.

5.3. Structure of the Run-Time Environment

If the run-time environment is divided into parts and each part has a strict functionality, for example, application, communication, method dispatcher, garbage collection, I/O, operating system, debugging, and monitoring parts, many benefits are provided. The following benefits was identified by Stankovic [22]

• It will be simpler to map an application to the resource structure because the application part could be isolated from unpredictable interrupts generated by the nondeterministic environment.

• The system would be more manageable due to the separation. It should be noted that each part must have its own resources to make the separation strict.

To be able to utilize any part efficiently there must be a small variation in execution time for each basic operation. For example the data moved in a move instruction could be accessible in cache memory or in the ordinary RAM.

This will give a big variation and when calculating the execution time for a real-time program the MAXTC has to be considered which will lead to poor utilization. A basic operation could also for example be a context switch.

To simplify the communication system it is preferable to use a broadcast bus topology. In such a topology, no relaying nodes have to be used in communication between nodes in the same network. Using such an approach, only one bus slot is used when sending one frame. This will simplify the tools that allocates bus bandwidth to the application.

In Section 6, we will present a prototype of the run-time environment.

6. AN RTT PROTOTYPE SYSTEM

6.1. Introduction

In this section we will briefly present some of the tools and solutions to the proposed models discussed earlier.

First, we will present a prototype of the run-time environment, thereafter a few words about the RTT compiler followed by a short description of the MAXTC Tool. Thereafter, we will present the principles of the configuration compiler, i.e., the resource mapping tool.

6.2. The RTT Prototype Run-Time Environment

6.2.1. Introduction. The run-time environment consists of a number of nodes. Each node is structurally partitioned into three separate units, one *application unit*, one *communication unit*, and one *time handling unit*. The two first units have their own resources such as CPU and memory. These two units communicate through a dual port memory. The time handling unit is implemented directly in hardware and thus also has its own resources.

The memory in the communication and application units are protected by a memory protection unit. This unit is developed especially for this architecture and will ensure that the tasks can only access memory to which they have been reserved rights. The reason for developing this unit is that commercial available memory management units have unpredictable timing behavior [22].

The communication and application unit also includes a real-time garbage collector [13].

6.2.2. The Application Unit. The application unit executes the application tasks and is responsible for the application I/O. The task execution platform is provided by the Rubus real-time operating system [1]. It contains guaranteed services (hard real-time) and best effort services (soft real-time). Rubus is divided into two executives, the hard real-time and the soft real-time executive.

- The hard real-time executive is based on the time-triggered execution paradigm and is dispatching tasks according to a pre-run-time generated schedule. It is also handles deadline supervision and takes care of deadline violations. The deadline control is made in cooperation with the time handling unit.
- The soft real-time executive is based on the event-triggered execution paradigm where a priority based scheduling policy is used. The hard real-time executive will always preempt the soft real-time executive when it is time to dispatch a hard real-time task.

6.2.3. The Communication Unit. The communication unit in each node is connected to a broadcast bus, in this case a CAN bus (ISO/DIS 11519, unit 1). This unit is responsible for the network communication and handles all messages sent to and from the application unit. It also handles group membership protocols [5].

The RTT communication protocol is implemented as an application layer on top of the data link layer of the CAN protocol. This protocol makes a distinction between hard and soft real-time messages. To fulfill timing requirements, hard real-time messages are scheduled pre run-time.

The protocol is based on the TDMA paradigm; i.e., the network bandwidth is divided into time slots. There is a maximum number of frames that can be sent in one bus slot. A frame corresponds to a CAN message. The nodes have a consistent view of the bus slots since there exists an approximate global time base.

TDMA based protocols usually allocate one bus slot per node. In its bus slot, the node can either send a frame or leave the bus slot unused. This leads to an inefficient use of network bandwidth. In the RTT communication protocol, several nodes can transmit frames in a bus slot according to the pre-run-time generated schedule. This is possible due to the collision avoidance arbitration mechanism provided by the CAN protocol.

Each node knows when all hard real-time frames have been sent every node listen to the traffic on the bus and has knowledge about the schedule. When all the hard real-time frames in a bus slot has been sent, soft real-time frames will be transmitted until the end of the bus slot. This makes it possible to get a high utilization of the network bandwidth.

6.2.4. The Time Handling Unit. The time handling unit includes separate timers for the communication and application units. The timers for each unit handles dispatching and deadline supervision of hard tasks and dispatching of soft tasks. Furthermore, this unit provides an approximate global time that could be read by the other two units. To be able to provide such a global time, a clock synchronization algorithm is implemented in the RTT communication protocol.

6.3. Compiler

The present version of the RTT compiler generates C-code which has to be compiled and linked with C-tools to generate a run-time system, as described in [9]. In this

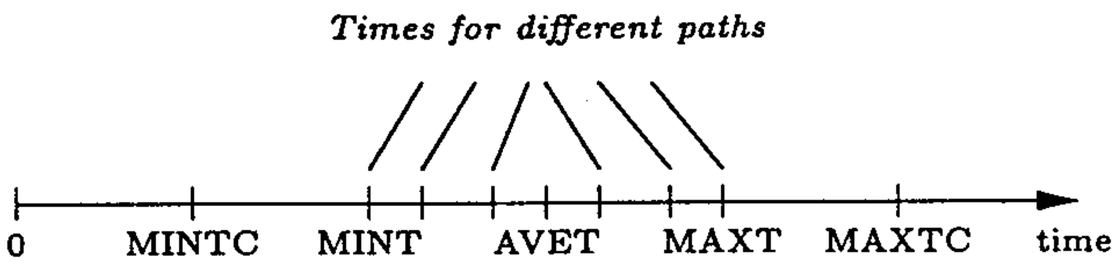


FIG. 9. Execution times for a program.

version, dynamic types are used, which may cause some problems, as mentioned in Section 4.3. The next version of the RTT compiler will attempt to solve these problems by using type inference. This compiler will also generate optimized assembly code, yielding a better utilization of resources.

6.4. The MAXTC Tool

The execution times for a program may vary depending on the path taken in the control graph. In Fig. 9, the following time measures are shown:

- MINT = real minimum execution time
- MAXT = real maximum execution time
- MINTC = calculated minimum execution time
- MAXTC = calculated maximum execution time
- AVET = $\Sigma_{i=1}^{n} T_i/n$ = pathwise average execution time of a program.
- OF = MAXTC/MAXT = overreservation factor.
- D = MAXT/MINT = dynamic factor.

The MAXTC calculation tool [9] calculates MAXTC. Naturally, the overreservation factor (OF) has to be as close to 1 as possible to avoid reservation of too large CPU resources in the run-time schedule. The tool also calculates MINTC. This number can for instance be used to assess the dynamic factor of a program. It also gives an indication of the spare capacity left for soft functionality. (Remember that hard real-time tasks are preallocated, while soft real-time tasks uses unallocated CPU-time and spare time from hard real-time tasks.)

Currently, there is a prototype of the tool which cooperates with the current dynamic type version of the RTT compiler. To be able to solve some problems with this implementation, the main one being high overreservation, the next version of the MAXTC tool will exploit the type information inferred by the compiler.

The method dispatch (or method invocation scheme) used in Smalltalk makes it difficult to calculate the execution time for a system before run-time. The reason for this is the linear search for methods which starts at the class of the receiver and goes up through the inheritance tree. When execution time is calculated, the time required for method dispatching must be predictable for every message. For this reason, a method dispatch table is used at run-time. The method dispatch table is built at compile time using an algorithm called modified two-way coloring (MTWC) by Huang and Chen [14]. With type information at hand most of the dynamically bound message sends could be statically bound. The reason for this is that the use of polymorphism is not used in the extent that one could believe [4].

6.5. The RTT Configuration Compiler

6.5.1. Introduction. In this section, we will present how the hard real-time part of an application is translated into a runnable application. As mentioned before, the hard real-time part is based on the time-triggered approach and scheduled prior to run-time. The translation process includes allocation of both CPU capacity for each node and bus bandwidth. The output from the translation process is a schedule for each mode and node, plus message space for the messages sent over the network. This translation is done by a tool called the RTT configuration compiler.

The configuration compiler requires an *architecture specification* and a *configuration specification*. The architecture specification lists the number of nodes and gives the characteristics of the nodes and the bus. The configuration specification describes the application as given by the designer, i.e., modes, use-cases, tasks.

The configuration compiler consists of two cooperating tools (Fig. 10): the *RTT communication handling tool* and the *RTT pre-run-time scheduler*.

The communication handling tool automatically inserts necessary system tasks, communication buffers, and mutual exclusion relationships to make it possible for the pre-run-time schedulers to find a schedule. Many pre-run-time schedulers only supports communication between tasks

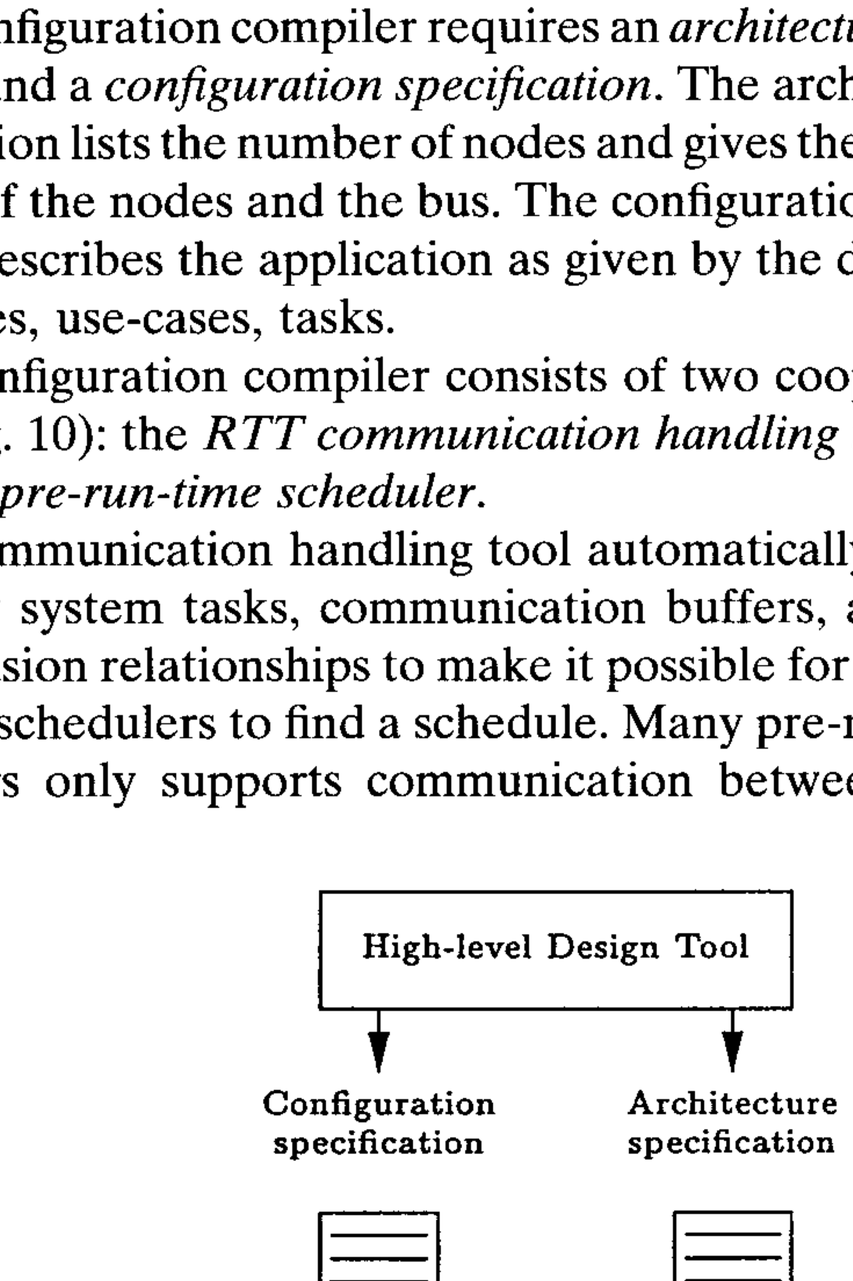


FIG. 10. The configuration compiler. A high-level graphical tool is used to produce the specifications for the compiler.

that have a precedence relationship [20, 16]. These schedulers cannot handle communication between task running with different period times, a desired property mentioned earlier.

In this section, we show how communication between tasks residing in different precedence graphs can be supported by using mutual exclusion relationships.

6.5.2. Architecture and Configuration Specification. To be able to transform an application to a resource structure, an architecture specification must be provided:

- The number of nodes in the system.
- The time between two consecutive points where the kernel may
  - —explicitly start the execution of a task;
  - —preempt a task;
  - —check if a task meets its deadline.
- The length in time of a bus slot. A message that is sent in one bus slot is available for the receiver in the next bus slot.
- The number of hard real-time frames that can be allocated to a bus slot by the scheduler.
- The accuracy of the clock synchronization.

The configuration specification includes synchronization and communication requirements for each mode of the application. This specification is generated from the programming tool which the user uses when designing an application. The structure of the configuration specification is very similar to the structure of an application; i.e., it includes modes, use-cases and tasks.

6.5.3. Communication Handling Tool. To support the communication requirements, system tasks are automatically installed by the communication handling tool. For example, assume two tasks communicating with each other such that the producer is a predecessor to the consumer in the precedence graph. Assume furthermore that the tasks are allocated on the same node. The communication tools will then install a system task as a successor to the producer and as a predecessor to the consumer. The role of the system task is to copy the contents from the producer's out-port to the consumer's in-port.

The reason for installing system tasks is that the communication will be transparent from the sender's point of view. So, for example, if we change the architecture to support redundant buses, we only have to change the system tasks.

Communication between two tasks can be performed between any two tasks that are defined in

- The same precedence chain and where the consumer is a successor to the producer. Multicast and broadcast are defined as a finite number of producer and consumer relations.
- The same precedence graph which have no precedence relationship.
- Different precedence graphs.
- The same precedence chain and where communication from a successor to a predecessor is required. This feature is useful to have when, for example, a control algorithm is implemented.

The communication can, of course, take part between tasks that are running on either the same node or on different nodes. It can either be buffered or unbuffered. When tasks share resources which do not allow concurrent access, the tasks must be defined to be mutually exclusive.

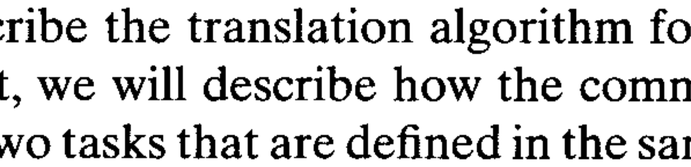
When a transformation is made, some system tasks are installed. A system task is from the run-time system's perspective a task which has special privileges. For example, a send task has the right to read from a task's out-ports. To be able to handle the communication requirements we have three types of system tasks:

- sendTask; this task reads a message from a predefined out-port of a task. Thereafter it passes the information to the communication subsystem. The information on where to read the information is passed to the system task as an argument when it is activated. The feature of passing information to the task is supported by the real-time kernel in use, Rubus [1].
- receiveTask; this task reads a message from the communication subsystem and stores the information at a predefined memory location. Normally, the memory location is a task's in-port.
- localSendTask; this task reads a message from a predefined out-port of a task and thereafter writes the message to a pre-defined memory location.

The translations for all the different cases can not be described here due to the limited space. Rather, we will describe the translation algorithm for two different cases. First, we will describe how the communication is handled by two tasks that are defined in the same precedence graph. Second, we will describe the communication between tasks that runs with different period times on different nodes. In the second example, we will also see how the mutual exclusion relationship can be used. The complete description for the different cases can be found in [7].

EXAMPLE 6.1. The producer and consumer is defined in the same precedence chain and the consumer is a successor to the producer in the precedence graph. The producer and consumer are allocated on different nodes (Fig. 11).

a) Specified precedence graph:



b) Transformed precedence graph:

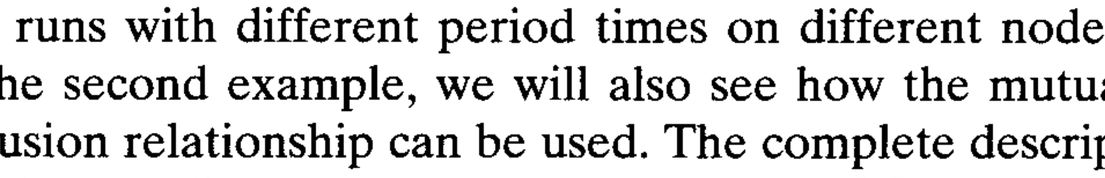


FIG. 11. The transformation of a precedence graph when the producer (P) and the consumer (C) are allocated on different nodes. The notation $C(n)$ means that the task C is allocated on node n.

Transformation:

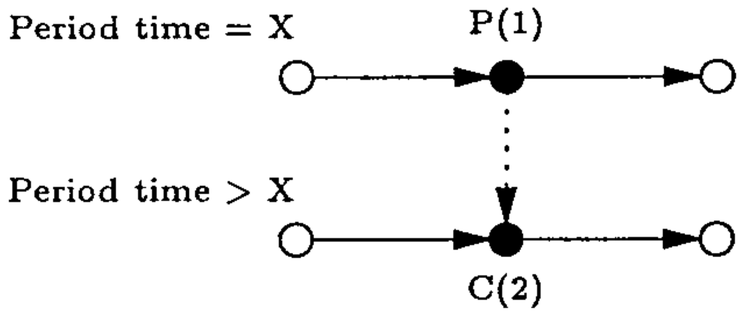
• Install a communication buffer to hold the temporary message on the producer's node.
• Install a send task as a predecessor to the consumer. The send task copies the contents from the producers out-port to the installed communication buffer.
• Install a communication buffer on the receiver's node. This buffer holds the contents of the received message.
• Install a receive task on the consumer's node as a successor to the abovementioned send task, and as a predecessor to the consumer. The role of the receive task is to copy the received message from the communication buffer to the consumer's in-port.

EXAMPLE 6.2. The producer and consumer are defined in different precedence graphs and allocated on different nodes. The period time of the producer is less than the period time of the consumer (Fig. 12).

Transformation:

• Install a variable to hold a temporary message on the producer's node.
• Install a local send task as a successor to the producer. The send task copies the contents from the producer's out-port to the installed variable.
• Install a communication buffer to hold the temporary message on the producer's node.
• Install a communication buffer to hold the message which was received by the consumer's node.
• Install a receive task on the consumer's node as a predecessor to the consumer. This task copies the message from the installed communication buffer to the consumer's in-port.
• Install a send task on the producer's node as a predecessor to the installed receive task. This task copies the message from the temporary variable to the installed communication buffer.

a) Specified precedence graphs:



b) Transformed precedence graphs:

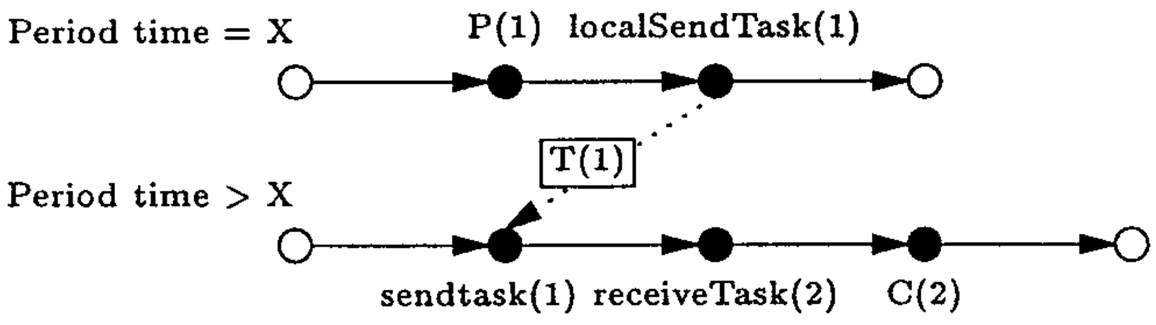


FIG. 12. The transformation when the producer's period time is less than the consumer's. Dotted lines indicate dataflow.

• Define a mutual exclusion relationship between the local send task and the send task.

The motivation to install a send task as a predecessor to the consumer at the producer's node is to minimize the traffic on the network. In Fig. 12, the box $T(1)$ depicts the temporary variable T allocated on node 1. The reason for defining a mutual exclusion relation between the local send tasks and the send task is to get a consistent message.

6.5.4. The RTT Scheduler. The scheduler tries to find a feasible schedule for the specification, using a heuristic search. The scheduler supports precedence and mutual exclusion relationships, but requires that tasks are preallocated to nodes. The mutual exclusion relationship is a binary relationship between two tasks (or two groups of tasks) which means that if one task is executing, the other task is blocked until the executing task terminates. This relationship is used when, for example, two tasks defined in different precedence graphs wants to access the same resource. A resource could for example be a shared object that is used to transfer information between two tasks. The scheduler can be configured to support either preemption or nonpreemption. The scheduler also supports bus allocation. The bus is scheduled by treating the bus as a processor and each frame is mapped to a specific type of task, named sendTask.

7. RELATED WORK

7.1. Introduction

This section compares RealTimeTalk with the following system architectures: DROL [23] (an object-oriented programming language for distributed real-time systems); MARS [16] (maintainable real-time systems); CHAOS [3] (concurrent hierarchical adaptable object system); ARTS [24] (a distributed real-time operating system) and DEDOS [12, 11] (dependable distributed operating system).

MARS and CHAOS describe a complete system architecture covering all steps from analysis to a running system. DROL includes an object model and programming model for distributed real-time systems. ARTS and DEDOS are focused on distributed operating system architectures. It should also be mentioned that no hardware details are covered and that all these systems, with the exception of MARS, are based on object-orientation; i.e., their language is based on C++ or an extension thereof. The MARS language, Modula/R, is a real-time extension of Modula/2.

7.2. DROL

In DROL, the synchronization and program logic is separated by meta-level objects. In RTT, this separation is done with precedence graphs. The advantages of our separation policy has been discussed before.

DROL can be used for applications with soft real-time

requirements while RTT is geared toward applications with both hard and soft real-time requirements.

The assumption that the system has no global time seems a bit strange and it is not clear how a server at another node can detect if a deadline will be missed or not. Additionally, it seems difficult to get an upper bound on delays throughout the system.

The time polymorphic invocation is a novel approach and it will be interesting to see if it is usable in real applications.

7.3. MARS

The major differences between MARS and RTT is the model of execution; in MARS, it is based on the process model, while RTT uses an object-oriented one. Furthermore, the MARS system supports only hard real-time, while RTT supports both soft and hard real-time.

There are, however, many similarities between the MARS system and RTT. For example, both systems separates the communication and synchronization from the code. In MARS, the precedence relationship between tasks are stronger because the precedence relationship includes both synchronization and communication. In RTT, these concepts have been separated to support communication between tasks residing in different precedence graphs and to allow implementation of feedback controllers in a straightforward way.

It should also be noted, that within the MARS system, redundant hardware aspects, dependability analysis, and testing is covered. These areas have not yet been covered in RTT.

7.4. CHAOS

A fundamental difference between CHAOS and RTT is CHAOS' support of adaptability. Adaptability means that the system can be changed during run-time to adapt to changes in the environment not known in advance. In comparison, an RTT system can only adapt to changes that are known in advance.

In the next generation of dynamic and adaptable systems, some parts of the system must be designed by preallocating all needed resources as in RTT. Furthermore, high-level features, as for instance planning, must be supported by an adaptable system. It seems that the dynamic behavior introduced in CHAOS will make it very hard to guarantee hard real-time requirements.

7.5. ARTS

The ARTS system is an event driven system, while RTT is a time driven system. The ARTS system can be seen as an architecture for testing different scheduling theories. The real-time object model is an interesting concept because it supports temporal encapsulation. Another difference, compared to RTT, is that the synchronization in ARTS is embedded in the code and thus inheritance anomalies could occur [2].

7.6. DEDOS

The DEDOS system is very interesting due to the support of both hard and soft real-time requirements. One of the big differences between DEDOS and RTT is the supported language; DEDOS is based on an extension of C++ called DEAL, while RTT is based on Smalltalk.

The communication style between the soft and hard real-time parts in DEDOS have also been used in RTT. However, in RTT, there is also support for sending events from the hard real-time part to soft real-time part. RTT also seems to be more focused on the analysis and design phase of a system. The explicit use of precedence graphs will hopefully make the system more easy to understand and maintain. In DEDOS, the precedence graphs are constructed from the program by a tool. When integrating the functions and synchronization within the code, inheritance anomalies could occur [2].

It should also be mentioned that DEDOS also includes support for fault-tolerance in both software and hardware.

8. CONCLUSION

We have presented the RealTimeTalk (RTT) system and how it supports important real-time areas such as synchronization, communication, distribution, timing requirements, modelling and programming. The RTT system basically consists of the following parts:

- Programming in the large.
  - —A model for designing both hard and soft real-time functionality.
  - —Interaction support between hard and soft real-time parts.
  - —Support for object-orientation as a mean for designing applications.
- A real-time language.
  - —A language based on Smalltalk with support for prototyping of applications.
  - —Real-time constructs to guarantee a predictable temporal behavior of an application.
- A run-time environment.
  - —A distributed hardware architecture.
  - —An operating system with support for both hard and soft real-time tasks.
  - —A communication system with support for both hard and soft real-time messages.
- Tools for mapping a specification to a resource structure.
  - —A compiler, with future support for type inference.
  - —A MAXTC tool, for calculating the maximum execution time for tasks.
  - —A configuration compiler, which synthesize the communication and synchronization requirements of an application.

The RTT approach has also been compared to other important real-time systems in the research community.

ACKNOWLEDGMENTS

This work was supported by the National Board for Industrial and Technical Development (Project 93-3180), the Västmanlands County Administrative Board, and fundings from Mälardalen University and the Royal Institute of Technology. We gratefully acknowledge help from Arcticus Systems AB on the run-time system.

REFERENCES

1. Arcticus Systems AB, Rubus OS Real-Time Operating System Tutorial. Technical report, Arcticus Systems AB Datavägen 9A, 175 62 Järfälla, Sweden, 1996.
2. L. Bergmans and M. Aksit, Composing synchronisation and real-time constraints. *The Object-Oriented Real-Time Workshop*, in conjunction with the 7th IEEE Symposium on Parallel and Distributed Processing, San Antonio, Texas, Oct. 1995, pp. 108–115.
3. T. E. Bihari and P. Gopinath, Object-oriented real-time systems: Concepts and examples. *IEEE Computer* (Dec. 1992).
4. E. Brorsson, C. Eriksson, and J. Gustafsson, RealTimeTalk: An object-oriented language for hard real-time systems. *Intl Workshop on Real-Time Programming WRTP'92, IFAC/IFIP*, Bruges, Belgium, June 1992.
5. C. Eriksson, M. Gustafsson, and H. Thane, A communication protocol for soft and hard real-time systems. *Euromicro Workshop on Real-Time 96*, 1996.
6. C. Eriksson, R. Hassel, K. Sandström, and L. Myrehed, A graphical design environment for the development of object-oriented hard real-time systems. *TOOLS Europe 95*, Paris, Mar. 1995.
7. C. Eriksson and K. Sandström, The translation of an application configuration to a runnable application by utilising a pre run-time scheduler. Technical Report CUS95RR02, Dept. of Computer Engineering, University of Mälardalen, Sweden, 1995.
8. A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Addison–Wesley, Reading, MA, 1989.
9. J. Gustafsson, Calculation of execution times in object-oriented real-time software—A study focused on RealTimeTalk. Licentiate thesis, 1994.
10. J. Gustafsson, K. Post, J. Mäki-Turja, and E. Brorsson, Benefits of type inference for an object-oriented real-time language. *Object-Oriented Real-Time Workshop*, San Antonio, Texas, Oct. 1995.
11. D. K. Hammer, P. Lemmens, E. Luit, O. S. van Roosmalen, P. van der Stok, and J. Verhoosel, DEDOS: A distributed environment for object-oriented real-time systems. *IEEE J. Parallel Distrib. Technol.* **2,** 4 (1994).
12. D. K. Hammer and O. S. van Roosmalen, An object-oriented model for the construction of dependable distributed systems. *2nd International Workshop on Object Orientation in Operating Systems*, 1992.
13. R. Hassel and K. Sandström, Garbage collection i realtid. Master's thesis, Univ. of Mälardalen, Sweden, 1993. [In Swedish]
14. S. Huang and D. Chen, Efficient algorithms for method dispatch in object-oriented programming systems. *J. Object-Oriented Comput.* (Sept. 1992).
15. H. Kopetz, Event-triggered versus time-triggered real-time systems. Lecture Notes in Computer Science, Vol. 563, pp. 87–101. Springer-Verlag, Berlin/New York, 1991.
16. H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger, Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro* **9,** 1 (Feb. 1989), 25–40.
17. N. Leveson, *Safeware, System Safety and Computers*. A Wesley, Reading, MA, 1995.
18. A. K. Mok, Fundamental design problems of distributed syst the hard-real-time environment. Ph.D. thesis, MIT, 1983.
19. P. Thijssen, P.D.V. van der Stok, and L. Somers, Formal veri and simulation of a real-time concurrency protocol. *Inter Workshop on Responsive Computer Systems*, 1992.
20. K. Ramamritham, Allocation and scheduling of complex tasks. *10th Int. Conf. on Distributed Computing Systems*, 1 108–115.
21. J. A. Stancovic, Misconceptions about real-time computing *Computer* **21** (Oct. 1988), 10–19.
22. J. A. Stankovic, The Spring architecture. *Euromicro Works Real-Time*, June 1990, pp. 104–113.
23. K. Takashio and M. Tokoro, An object-oriented language for uted real-time systems. *OOPSLA '92*, Vancouver, 1992, pp. 1
24. H. Tokuda and C. W. Mercer, ARTS: A distributed real-time *Oper. Systems Rev.* **23,** 3 (1989).
25. M. Törngren, Modelling and design of distributed real-time applications. Ph.D. thesis, Dept. of Machine Design, Royal of Technology, Stockholm, Sweden, 1995.

CHRISTER ERIKSSON received a B.Sc. in mathematics fro sala University in 1988 and a licentiate degree from the Royal of Technology, Stockholm, Sweden in 1994. He worked for ABB A tion AB from 1984 until 1988, primarily on the run-time system f Master. He is currently a lecturer at Mälardalen University. His interests are design of real-time systems, object-oriented progra distributed architectures for real-time systems, and real-time o systems.

JUKKA MÄKI-TURJA received a B.Sc. in computer scien Mälardalen University in 1993. Since 1994, he has been a Ph.D. at the University. His primary research interests are in langua compilers, especially program analysis. He is currently involve RealTimeTalk project, working with the RTT language and its ty ence system.

KJELL POST received an M.Sc. in computer engineeri Linköpings Universitet in 1987 and a Ph.D. in computer science University of California, Santa Cruz in 1994. He is currently a pro Mälardalen University. His research interests are implementat analysis issues for programming languages.

MIKAEL GUSTAFSSON received an M.Sc. in electrical eng from the Royal Institute of Technology, Stockholm, Sweden Between 1984 and 1991, he was employed by ABB Automatio a designer of man-machine communication software used in t Master process control product family. Since 1991, he has be Mälardalen University, where he is currently a lecturer. His research interests covers computer architectures, operating syste data communication protocols for hard real-time systems.

JAN GUSTAFSSON is a senior lecturer at Mälardalen Un where he currently is the Head of the Department of Comput neering. He received a B.Sc. in mathematics, astronomy, and c science in 1974 from Uppsala University and a licentiate degre chine elements, computer controlled mechanics in 1994 from th Institute of Technology in Stockholm. From 1975 to 1985, he ABB (then ASEA), working with development of real-time i control systems. Since 1985, he has been with Mälardalen Un where his main topics are programming methodology, object- programming, and real-time systems.

KRISTIAN SANDSTRÖM received a B.Sc. in engineering f lardalen University, Västerås, in 1995. Since then, he has been in the Department of Computer Engineering at Mälardalen U

as a research engineer. His primary research interests are software models for real-time systems, design of real-time systems, object-oriented programming, and scheduling in real-time systems.

ELLUS BRORSSON receive a M.Sc. in mechanical engineering from the Royal Institute of Technology, Stockholm, Sweden in 1989. Since then, he has been with the Mechatronics group in the Department Machine Design at the Royal Institute of Technology, where he has be working as a teaching fellow and attending the Ph.D program. Recent he became a lecturer in the Computer Engineering Department at D larna University. His primary research interest is real-time systems a languages for real-time.

Received November 1, 1995; revised February 1, 1996; accepted April 25, 1996

Timed Automata with Asynchronous Processes: Schedulability and Decidability

Elena Fersman, Paul Pettersson and Wang Yi*
Uppsala University
Department of Computer Systems
P.O. Box 325, S-751 05 Uppsala, Sweden
{elenaf,paupet,yi}@docs.uu.se

Abstract. We adopt a model of timed automata extended with asynchronous processes i.e. tasks triggered by events. A task is an executable program characterized by its worst execution time and deadline, and possibly other parameters such as priorities etc. for scheduling. The main idea is to associate each location of an automaton with a task (or a set of tasks). A transition leading to a location denotes an event triggering the tasks and the clock constraint on the transition specifies the possible arrival times of the event. This yields a model for real-time systems expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive. An automaton is schedulable if there exists a (preemptive or non-preemptive) scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable in the sense that all associated tasks can be computed within their deadlines.

Our main result is that the schedulability checking problem is decidable. The problem has been conjectured to be undecidable due to the nature of preemptive scheduling. To our knowledge, this is the first decidability result for preemptive scheduling in dense-time models. The proof is based on a class of suspension automata, that is timed automata with subtraction in which clocks may be updated by subtraction operations. We show that if each clock is bounded with a maximal constant and subtraction operations are performed on clocks only in the bounded zone, the reachability problem is decidable. The crucial observation is that subtraction preserves region equivalence in the bounded clock zone, and the schedulability checking problem can be encoded as a reachability problem for such automata. Based on the proof, we have developed a symbolic technique, which has been implemented as a prototype tool for schedulability analysis.

1 Introduction

Real time systems are often designed as a set of tasks imposed with hard time constraints such as deadlines. The tasks may be triggered periodically or sporadically (non-periodically) by either time or events. Hard time constraints mean that whenever a task is triggered (released), it must be computed within the given deadline. One of the most important issues in the development of real time systems is *schedulability analysis* prior to the implementation. It is to check, based on models of the environment (controlled systems) and control software under development, whether the tasks can be guaranteed to meet their deadlines with the available computing resources such as processor time.

* Contact author.

In the area of real time scheduling, researchers have developed various methods [But97] e.g. rate monotonic scheduling, which are widely applied in the analysis of periodic tasks in time-driven systems. To deal with *non-periodic* tasks in event–driven systems, the standard method is to consider non-periodic tasks as periodic using the estimated *minimal* inter-arrival times as *task periods*. Clearly, the analysis based on such a task model would be pessimistic in many cases, e.g. a task set which is schedulable may be considered as non-schedulable as the inter-arrival times of the tasks may vary over time, that are not necessary minimal. To achieve more precise analysis, we need task models that allow more precise and relaxed timing constraints.

In recent years, in the area of formal methods, there have been several advances in formal modeling and analysis of real time systems based the theory of timed automata due to the pioneering work of Alur and Dill [AD94]. Notably, a number of verification tools have been developed (e.g. Kronos and UPPAAL [DY95,BLL+96,LPY97]) in the framework of timed automata, that have been successfully applied in industrial case studies (e.g. [BGK+96,LP97,LPY98]). Timed automata have proved expressive enough to model many real-life examples, in particular, for event-driven systems. The advantage with timed automata in modeling systems is that one may specify more relaxed timing constraints on events (i.e. discrete transitions) than the traditional approach in which events are often considered to be periodic. Moreover, timed automata also allow for modelling other behavioral aspects of systems such as synchronization and concurrency. However, it is not clear how timed automata can be used for schedulability analysis because there is no support for specifying resource requirements and hard time constraints on computations.

Following the work of [EWY98], we propose to extend timed automata with asynchronous processes i.e. tasks triggered by events. A task is an executable program characterized by its worst case execution time and deadline, and possibly other parameters such as priorities etc for scheduling. The main idea is to associate each location of an automaton with a task (or a set of tasks in the general case). Intuitively a transition leading to a location in the automaton denotes an event triggering the task and the guard (clock constraints) on the transition specifies the possible arrival times of the event. Semantically, an automaton may perform two types of transitions. Delay transitions correspond to the execution of running tasks (with highest priority) and idling for the other waiting tasks. Discrete transitions correspond to the arrival of new task instances. Whenever a task is triggered, it will be put in the scheduling queue for execution (i.e. the ready queue in operating systems). We assume that the tasks will be executed according to a given scheduling strategy e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first). Thus during the execution of an automaton, there may be a number of processes (released tasks) running logically in parallel.

For example, consider the automaton shown in Figure 1. It has three locations l_0, l_1, l_2, and two tasks P and Q (triggered by a and b) with computing time and relative deadline in brackets $(2, 10)$, and $(4, 8)$ respectively. The automaton models a system starting in its initial location may move to location l_1 by event a at any time, which triggers the task P. In location l_1, as long as the constraints $x \geq 10$ and $y \leq 40$ are satisfied and event a occurs, a copy (instance) of task P will be created and put in the scheduling queue. However, in location l_1, it can not create more than 5 instances of task P because the constraint $y \leq 40$ will be violated after 40 time units. In fact, every copy will be computed before the next instance arrives and the scheduling queue may contain at most one task instance and no task instance will miss its deadline in location l_1. In location l_1, the system is also able to accept event b, trigger the task Q and then switch to location l_2. In l_2, because there is no constraints labelled on the

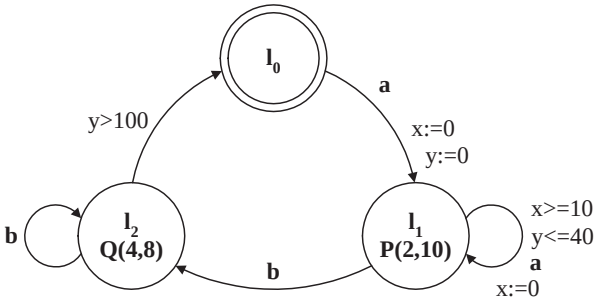


Fig. 1. Timed automaton with asynchronous processes.

b-transition, it may accept an unbounded number of b's, and create an unbounded number of copies of task Q in 0 time. in zero time because there is no constraint on the b-transition. This is the so-called zeno behavior. However, after more than two copies of Q, the queue will be non schedulable[1]. This means that the system is non-schedulable. Thus, zeno-behaviour will correspond to non schedulability in our setting, which is a natural property of the model.

We shall formalize the notion of schedulability in terms of reachable states. A state of an extended automaton will be a triple (l, u, q) consisting of a location l, a clock assignment u and a task queue q. The task queue contains pairs of remaining computing times and relative deadlines for all released tasks. Naturally, a state (l, u, q) is schedulable if q is schedulable in the sense there there exists a scheduling strategy with which all tasks in q can be computed within their deadlines. An automaton is schedulable if all reachable states of the automaton are schedulable. Note that the notion of schedulablity above is relative to the scheduling strategy. A task queue which is not schedulable with one scheduling strategy, may be schedulable with another strategy. In [EWY98], we have shown that under the assumption that the tasks are non-preemptive, the schedulability checking problem can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. The result essentially means that given an automaton it is possible to check whether the automaton is schedulable with any *non-preemptive* scheduling strategy. For *preemptive scheduling* strategies, it has been conjectured that the schedulability checking problem is undecidable because in *preemptive scheduling* we must use stop-watches to accumulate computing times for tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable.

Surprisingly the above conjecture is wrong. In this paper, we establish that the schedulability checking problem for extended timed automata is decidable for preemptive scheduling. Note that the preemptive earliest deadline first algorithm (EDF) is optimal in the sense if EDF can not schedule a task set, no other algorithms can. Thus our result applies to not only preemptive scheduling, but any scheduling strategy. That is, for a given extended timed automata, it is checkable if there exists a scheduling strategy (preemtive or non preemtive) with which the automaton is schedulable. The main idea in the proof is to model scheduling strategies with variants of timed automata (not stop-watch automata), and then encode the schedulability analysis

[1] According to the optimal scheduling strategy EDF, no scheduling strategy will be able to schedule the queue $[Q(4,8), Q(4,8), Q(4,8)]$ so that the deadline for the last instance of Q can be met.

problem as a reachability problem. We identify a variant of timed automata (a class of suspension automata): *timed automata with subtraction* in which clocks may be updated by subtraction. We show that if each clock is bounded with a maximal constant and subtraction operations are performed on clocks only in the bounded zone, the reachability problem is decidable[2]. The crucial observation is that the schedulability checking problem can be translated to a reachability problem for bounded timed automata with subtraction.

The rest of this paper is organized as follows: Section 2 presents the syntax and semantics of timed automata extended with tasks. Section 3 describes scheduling problems related to the extended model. Section 4 is devoted to the main proof that the schedulability checking problem for preemptive scheduling is decidable. Section 5 concludes the paper with summarized results and future work, as well as a brief summary and comparison with related work.

2 Timed Automata with Tasks

Recall that a timed automaton is a standard finite-state automaton extended with a finite collection of real-valued clocks. The transitions of a timed automaton are labelled with a *guard* (constraints on clocks), an *action*, and a *clock reset* (a subset of clocks to be reset to zero). We may view a timed automaton as an abstract model of a running system. The model describes the possible events (alphabets accepted by the automaton) that may occur during the operation of the system and the occurrence of the events must obey the timing constraints (given by the clock constraints).

However it is not clear how events or action symbols accepted by a timed automaton should be handled or computed. In fact, there is no way in timed automata to specify resource requirements and hard time constraints on computations. We propose to extend timed automata with asynchronous processes i.e. tasks triggered by events asynchronously. The main idea is to associate each location of a timed automaton with an executable program (or a set of programs in the general case) called a *task type* or simply a task.

2.1 Syntax

Let $\mathcal{P}$ ranged over by P, Q, R, denote a finite set of task types. A task type may have different instances that are copies of the same program with different inputs. We further assume that the *worst case execution times* and *hard deadlines* of tasks in $\mathcal{P}$ are known [3]. Thus, each task P is characterized as a pair of natural numbers denoted $P(C, D)$ with $C \leq D$, where C is the worst case execution time of P and D is the relative deadline for P. The deadline D is a relative deadline meaning that when task P is released, it should finish within D time units. We shall use $C(P)$ and $D(P)$ to denote the worst case execution time and relative deadline of P respectively.

As in timed automata, assume a finite set of alphabets Act for actions and a finite set of real-valued variables $\mathcal{C}$ for clocks. We use a, b etc to range over Act and x_1, x_2

[2] Because the subtraction preserves region equivalence in the bounded clock zone, and therefore the state space of a bounded automaton with subtraction can be partitioned according to region equivalence

[3] Note that tasks may have other parameters such as fixed priority for scheduling and other resource requirements e.g. on memory consumption. For simplicity, in this paper, we only consider computing time and deadline.

etc. to range over $\mathcal{C}$. We use $\mathcal{B}(\mathcal{C})$ ranged over by g to denote the set of conjunctive formulas of atomic constraints in the form: $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \mathcal{C}$ are clocks, $\sim \in \{\leq, <, \geq, >\}$, and C, D are natural numbers. The elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints*.

Definition 1. *A timed automaton extended with tasks, over actions Act, clocks $\mathcal{C}$ and tasks $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M\rangle$ where*

- *$\langle N, l_0, E, I\rangle$ is a timed automaton where*
  - *N is a finite set of locations ranged over by l, m, n,*
  - *$l_0 \in N$ is the initial location, and*
  - *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times Act \times 2^{\mathcal{C}} \times N$ is the set of edges.*
  - *$I : N \mapsto \mathcal{B}(\mathcal{C})$ is a function assigning each location with a clock constraint (a location invariant).*
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations with tasks* [4].

Intuitively, a discrete transition in an automaton denotes an event triggering a task and the guard (clock constraints) on the transition specifies all the possible arrival times of the event (or the associated task). Whenever a task is triggered, it will be put in the scheduling (or task) queue for execution (corresponding to the ready queue in operating systems).

Clearly extended timed automata is at least as expressive as timed automata; for example, if M is the empty mapping, we will have ordinary timed automata. It is a rather general and expressive model. For example, it may model time-triggered periodic tasks as a simple automaton as shown in Figure 2(a) where P is a periodic task with computing time 2, deadline 8 and period 20. More generally it may model systems containing both periodic and sporadic tasks as shown in Figure 2(b) which is a system consisting of 4 tasks as annotation on locations, where P_1 and P_2 are periodic with periods 20 and 40 respectively (specified by the constraints: x=20 and x=40), and Q_1 and Q_2 are sporadic or event driven (by event a and b respectively).

In general, there may be a number of processes (released tasks) running logically in parallel. For example, an instance of task Q_2 may be released before the preceding instance of task P_1 has been computed because there is no constraint on when b_2 will arrive. This means that the scheduling queue may contains at least P_1 and Q_2. In fact, instances of all four task types may appear in the queue at the same time.

To have a more general model, we may allow data variables shared between automata and tasks. For example. a boolean can be used to denote the completion of a task. The arrivals of events may be effected by the completion of certain tasks. This will not add any technical difficulty. However for simplicity, we will not consider synchronization between an automaton and the associated tasks in this paper. The only requirement on the completion of a task is given by the deadline. When a task is finished does not effect the control behavior specified in the automaton.

To handle concurrency and synchronization, parallel composition of extended timed automata may be introduced in the same way as for ordinary timed automata (e.g. see [LPY95]) using the notion of synchronization function [HK89]. For example, consider the parallel composition $A||B$ of A and B over the same set of actions Act. The set

[4] Note that M is a partial function meaning that some of the locations may have no task. Note also that we may also associate a location with a set of tasks instead of a single one. It will not introduce technical difficulties.

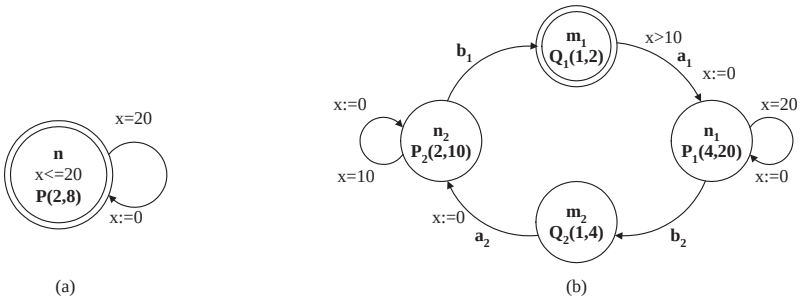


Fig. 2. Modeling Periodic and Sporadic Tasks.

of nodes of $A||B$ is simply the product of A's and B's nodes, the set of clocks is the (disjoint) union of A's and B's clocks, the edges are based on synchronizable A's and B's edges with enabling conditions conjuncted and reset-sets unioned. Note that due to the notion of synchronization function [HK89], the action set of the parallel composition will be Act and thus the task assignment function for $A||B$ is the same as for A and B.

2.2 Operational Semantics

Semantically, an extended timed automaton may perform two types of transitions just as standard timed automata. But the difference is that delay transitions correspond to the execution of running tasks with highest priority (or earliest deadline) and idling for the other tasks waiting to run. Discrete transitions corresponds to the arrival of new task instances.

We represent the values of clocks as functions (called clock assignments) from $\mathcal{C}$ to the non–negative reals $\mathcal{R}_{\geq 0}$. We denote by $\mathcal{V}$ the set of clock assignments for $\mathcal{C}$. Naturally, a semantic state of an automaton is a triple (l, u, q) where l is the current control location, u denotes the current values of clocks, and q is the current task queue. We assume that the task queue takes the form: $[P_1(c_0, d_0), P_2(c_1, d_1)...P_n(c_n, d_n)]$ where $P_i(c_i, d_i)$ denotes a released instance of task type P_i with remaining computing time c_i and relative deadline d_i.

Assume that there are a number of processors running the released task instances according to a certain scheduling strategy Sch e.g. FPS (fixed priority scheduling) or EDF (earliest deadline first) which sorts the task queue whenever new tasks arrives according to task parameters e.g. deadlines. In general, we assume that a scheduling strategy is a sorting function which may change the ordering of the queue elements only. Thus an action transition will result in a sorted queue including the newly released tasks by the transition. A delay transition with c time units is to execute the task in the first positon of the queue with c time units. Thus the delay transition will decrease the computing time of the first task with c. If its computation time becomes 0, the task should be removed from the queue (shrinking). We adopt the structural equivalence over queues respecting $[P_1(0, d), P_2(c_1, d_1)...P_n(c_n, d_n)] = [P_2(c_1, d_1)...P_n(c_n, d_n)]$. Moreover, after a delay transition with c time units, the deadlines of all tasks in the queue will be decreased by c (since time has progressed by c). To summarize the above intuition, we introduce the following functions on task queues:

- Sch is a sorting function for task queues (or lists), that may change the ordering of the queue elements only. For example, $\mathsf{EDF}([P(3.1,10),Q(4,5.3)]) = [Q(4,5.3),P(3.1,10)])$. We call such sorting functions scheduling strategies that may be preemptive or non-preemptive [5].
- Run is a function which given a real number t and a task queue q returns the resulted task queue after t time units of execution according to available computing resources. For simplicity, we assume that only one processor available [6]. Then the meaning of $\mathsf{Run}(q,t)$ should be obvious and it can be defined inductively. For example, let $q = [Q(4,5),P(3,10)]$. Then $\mathsf{Run}(q,6) = [P(1,4)]$ in which the first task is finished and the second has been executed for 2 time units.

Further, for a real number $t \in \mathcal{R}_{\geq 0}$, we use $u+t$ to denote the clock assignment which maps each clock x to the value $u(x)+t$, $u \models g$ to denote that the clock assignment u satisfies the constraint g and $u[r \mapsto 0]$ for $r \subseteq \mathcal{C}$, to denote the clock assignment which maps each clock in r to 0 and agrees with u for the other clocks (i.e. $\mathcal{C}\backslash r$).

Now we are ready to present the operational semantics for extended timed automata by transition rules:

Definition 2. *Given a scheduling strategy* Sch [7]*, the semantics of an extended timed automaton* $\langle N, l_0, E, I, M\rangle$ *with initial state* (l_0,u_0,q_0) *is a transition system defined by the following rules:*

- $(l,u,q) \xrightarrow{a}_{\mathsf{Sch}} (m, u[r\mapsto 0], \mathsf{Sch}(M(m) :: q))$ *if* $l \xrightarrow{g,a,r} m$ *and* $u \models g$
- $(l,u,q) \xrightarrow{t}_{\mathsf{Sch}} (l, u+t, \mathsf{Run}(q,t))$ *if* $(u+t) \models I(l)$

where $M(m) :: q$ *denotes the queue with* $M(m)$ *inserted in* q*.*

We shall omit Sch from the transition relation whenever it is understood from the context.

Consider the automaton in Figure 2(b). Assume that preemptive earliest deadline (EDF) is used to schedule the task queue i.e. Sch is sorting the queue in EDF order, (and Run is as defined earlier). Then the automaton with initial state $(m_1, [x = 0], [Q_1(1,2)])$ may demonstrate the following sequence of typical transitions:

$$
\begin{aligned}
(l_0,[x=0],[Q_1(1,2)]) &\xrightarrow{0.5} (m_1,[x=0.5],[Q_1(0.5,1.5)])\\
&\xrightarrow{10} (m_1,[x=10.5],[Q_1(0,0)]) = (m_1,[x=10.5],[])\\
&\xrightarrow{a_1} (n_1,[x=0],[P_1(4,20)])\\
&\xrightarrow{0.5} (n_1,[x=0.5],[P_1(3.5,19.5)])\\
&\xrightarrow{b2} (m_2,[x=0.5],[Q_2(1,4),P_1(3.5,19.5)])\\
&\xrightarrow{0.3} (m_2,[x=0.8],[Q_2(0.7,3.7),P_1(3.5,19.2)])\\
&\xrightarrow{a_2} (n_2,[x=0],[Q_2(0.7,3.7),P_2(2,10),P_1(3.5,19.2)])\\
&\xrightarrow{b_1} (m_1,[x=0],[Q_2(0.7,3.7),Q_1(1,2),P_2(2,10),P_1(3.5,19.2)])\\
&\ldots
\end{aligned}
$$

[5] As in scheduling theory, we adopt the standard assumptions on scheduling strategies: A non-preemptive strategy will never change the position of the first element of a queue. A preemtive strategy may change the ordering of task types only, but never change the ordering of task instances of the same type.

[6] However, the semantics may be extended to multi-processor setting. We only need to modify the function Run according to available resources (i.e. number of processors in this case).

[7] Note that we fixed Run to be the function that represents a one-processor system.

Note that the queue is always ordered by EDF with increasing deadlines and though the queue is growing, in location m_1, the generation of new task instances will be slowed down by the constraint $x > 10$ labeled on the edge m_1 to n_1 and the queue will be reduced by the following transition:

$$(m_1, [x = 0], [Q_2(0.7, 3.7), Q_1(1, 2), P_2(2, 10), P_1(3.5, 19.2)]) \xrightarrow{10} (n_1, [x = 10], [])$$

The automaton may also perform other sequences of transitions triggering copies of tasks. For example, it may stay in location n_1 or n_2 where instances of the periodic tasks P_1 and P_2 may be released and executed.

3 Schedulability Analysis

In this section we study verification problems related to the model presented in previous section. First, we have the same notion of reachability as for timed automata.

Definition 3. *We shall write* $(l, u, q) \longrightarrow (l', u', q')$ *if* $(l, u, q) \xrightarrow{a} (l', u', q')$ *for an action* a *or* $(l, u, q) \xrightarrow{t} (l', u', q')$ *for a delay* t*. For an automaton with initial state* (l_0, u_0, q_0)*,* (l, u, q) *is reachable iff* $(l_0, u_0, q_0)(\longrightarrow)^*(l, u, q)$*.*

Note that the reachable state space of an extended timed automaton is infinite because of not only the real-valued clocks, but also the unbounded size of the task queue. However, for certain analysis, e.g. safety properties (that are not related to the task queue), we may only be interested in the reachability of locations. A nice property of our extension is that the location reachability problem can be checked by the same technique as for timed automata [DOTY95,BLL+96,ACH+95,YPD94]. So we may view the original timed automaton (without task assignment) as an abstraction of its extended version, which preserves location reachability. The existing model checking tools such as [DOTY95,BLL+96] can be applied directly to verify the abstract models.

But if properties related to the task queue are of interests, we need to develop a new verification technique. One of the most interesting properties of extended automata related to the task queue is schedulability.

According to the transition rules, the task queue is growing with action transitions and shrinking with delay transitions. Multiple copies (instances) of the same task type may appear in the queue [8]. To illustrate the changing size of the task queue, we consider again the automaton shown in Figure 1 in Section 1. In location l_1, it can never generate more than 5 instances of P due to the constraint $y \leq 40$, namely the number of copies may be bounded by the clock constraints [9]. This is illustrated by the following transitions:

[8] In practice, copies of the same task type may have different input to compute.

[9] In fact, in location l_1, the size of the queue will be bounded by 1 because every released instance will be computed before the next release that will not arrive within 10 time units due to the constraint $x \geq 10$.

$$
\begin{aligned}
(l_0, [x=0, y=0], []) &\xrightarrow{3} (l_0, [x=3, y=3], []) \\
&\xrightarrow{a} (l_1, [x=0, y=0], [P(2,10)]) \\
&\xrightarrow{2.5} (l_1, [x=2.5, y=2.5], [P(0,7.5)]) = (l_1, [x=2.5, y=2.5], []) \\
&\xrightarrow{7.5} (l_1, [x=10, y=10], []) \\
&\xrightarrow{a} (l_1, [x=0, y=10], [P(2,10)]) \\
&\ldots \\
&\xrightarrow{a} (l_1, [x=0, y=40], [P(2,10)]) \\
&\xrightarrow{3.4} (l_1, [x=3.4, y=43.4], [P(0,6.6)]) = (l_1, [x=3.4, y=43.4], [])
\end{aligned}
$$

In fact, the queue size in location l_1 is bounded by 1. But in location l_2, an infinite number of copies of Q may be released in zero time because there is no constraint on the b-transition, which demonstrates the so-called zeno behavior:

$$
\begin{aligned}
(l_1, [x=3.4, y=43.4], []) &\xrightarrow{b} (l_2, [x=3.4, y=43.4], [Q(4,8)]) \\
&\xrightarrow{b} (l_2, [x=3.4, y=43.4], [Q(4,8), Q(4,8)]) \\
&\xrightarrow{b} (l_2, [x=3.4, y=43.4], [Q(4,8), Q(4,8), Q(4,8)]) \\
&\ldots
\end{aligned}
$$

But note that after more than two copies of Q, the queue will be non schedulable[10]. We notice that zeno-behaviour will correspond to non schedulability in our setting which is a nice property of the model. We shall see that non-schedulability can be checked by reachability analysis. However zeno-freeness does not necessarily implies schedulability.

Now we formalize the notion of schedulability.

Definition 4. *(Schedulability) A state (l, u, q) where $q = [P_1(c_1, d_1) \ldots P_n(c_n, d_n)]$ is a failure denoted (l, u, Error) if there exists i such that $c_i \geq 0$ and $d_i < 0$, that is, a task failed in meeting its deadline. Naturally an automaton A with initial state (l_0, u_0, q_0) is non-schedulable with Sch iff $(l_0, u_0, q_0)(\longrightarrow_{\mathsf{Sch}})^*(l, u, \mathsf{Error})$ for some l and u. Otherwise, we say that A is schedulable with Sch.*

More generally, we say that A is schedulable iff there exists a scheduling strategy Sch with which A is schedulable.

The schedulability of a state may be checked by the standard schedulability test. We say that (l, u, q) is schedulable with Sch if $\mathsf{Sch}(q) = [P_1(c_1, d_1) \ldots P_n(c_n, d_n)]$ and $(\sum_{i \leq k} c_i) \leq d_k$ for all $k \leq n$. Alternatively, an automaton is schedulable with Sch if all its reachable states are schedulable with Sch.

Note that checking schedulability of a state is a trivial task according to the definition. But checking the relative schedulability of an automaton with respects to a given scheduling strategy is not easy, and checking the general schedulability (equivalent to finding a scheduling strategy to schedule the automaton) is even more difficult.

Fortunately the queues of all schedulable states of an automaton are bounded. First note that a task instance that has been started can not be preempted by another instance of the same task type. This means that there is only one instance of each task type in the queue whose computing time can be a real number and it can be arbitrarily

[10] According to the optimal scheduling strategy EDF, no scheduling strategy will be able to schedule the queue $[Q(4,8), Q(4,8), Q(4,8)]$ to meet the deadline for the last instance of Q.

small. Thus the number of instances of each task type $P \in \mathcal{P}$, in a schedulable queue is bounded by $\lceil D(P)/C(P) \rceil$ and the size of schedulable queues is bounded by

$$\sum_{P \in \mathcal{P}} \lceil D(P)/C(P) \rceil$$

We will code schedulability checking problems as reachability problems. First, we consider the case of non-preemptive scheduling to introduce the problems we are aiming at in this paper. We have the following positive result.

Theorem 1. *The problem of checking schedulability relative to non-preemptive scheduling strategy for extended timed automata is decidable.*

Proof. A detailed proof is given in [EWY98]. We scketch the proof idea here. It is to code the given scheduling strategy as a timed automaton (called the scheduler) denoted $E(\mathsf{Sch})$ which uses clocks to remember computing times and relative deadlines for released tasks. Note that for non-preemptive scheduling, only one instance is running, which means that only one clock, denoted c is needed to remember the computing time for the running task and one clock, denoted d_i for each released task instance P_i to remember the deadline.

The scheduler automaton is constructed as follows: Whenever a task instance P_i is released by an event $\mathsf{release}_i$, d_i is reset to 0. Whenever a task is started to run[11], c is reset to 0. Whenever the constraint $d_i = 0$ is satisfied, and P_i is not running, an error-state (non-schedulable) should be reached.

We also need to transform the original automaton A to $E(A)$ to synchronize with the scheduler that P_i is released whenever a location, say l to which P_i is associated, is reached. This is done simply by replacing actions labeled on transitions leading to l with $\mathsf{release}_i$.

Finally we construct the product automaton $E(\mathsf{Sch})||E(A)$ in which both $E(\mathsf{Sch})$ and $E(A)$ can only synchronize on identical action symbols namely $\mathsf{release}_i$'s. It can be proved that if an error-state of the product automaton is reachable, the original extended timed automaton is non-schedulable.

For *preemptive scheduling* strategies, it has been conjectured that the schedulability checking problem is undecidable. The reason is that if we use the same ideas as for non-preemptive scheduling to encode a preemptive scheduling strategy, we must use stop-watches (or integrators) to add up computing times for suspended tasks. It appears that the computation model behind preemptive scheduling is stop-watch automata for which it is known that the reachability problem is undecidable. Surprisingly this conjecture is wrong.

Theorem 2. *The problem of checking schedulability relative to preemptive scheduling strategy for extended timed automata is decidable.*

The rest of this paper will be devoted to the proof of this theorem. It follows from Lemma 3, 4, and 5 established in the following section.

Before we go further, we state a more general result follows from the above theorem.

[11] Initially or the running task P_i is finished i.e. the constraint $c = C(P)$ is satisfied.

Theorem 3. *The problem of checking schedulability for extended timed automata is decidable.*

From scheduling theory[But97], we know that the preemptive version of Earliest Deadline First scheduling (EDF) is optimal in the sense that if a task queue is non schedulable with EDF, it can not be schedulable with any other scheduling strategy (preemptive or non preemptive). Thus, the general schedulability checking problem is equivalent to the relative schedulability checking with respects to EDF.

4 Decidability and Proofs

We shall code the schedulability checking problem as a reachability problem. Note that in the case of non preemptive scheduling, we are able to do so using timed automaton only. For preemptive scheduling, we need a more expressive model.

4.1 Timed Automata with Subtraction

Definition 5. *A timed automaton with subtraction is a timed automaton in which clocks may be updated by subtraction in the form $x := x - C$ in addition to reset of the form: $x := 0$, where C is a natural number.*

Note that this is the so called suspension automata [MV94] and also called updatable automata in [BDFP00]. It is known that the reachability problem for this class of automata is undecidable. However, for the following class of suspension automata, location reachability is decidable.

Definition 6. *(Bounded Timed Automata with Subtraction) A timed automaton is bounded iff for all its reachable states (l, u, q), there is a maximal constant C_x for each clock x such that*

1. *$u(x) \geq 0$ for all clocks x, i.e. clock values should not be negative* [12]*, and*
2. *$u(x) \leq C_x$ for all l' such that $l \xrightarrow{gar} l'$ and $(x := x - C) \in r$ for some C.*

Note that in general, it may be difficult to compute the maximal constants from the syntax of an automaton. But we shall see that we can compute the constants for our encoding of scheduling problems.

Because subtraction operations on clocks are performed only within a bounded area. It preserves the region equivalence. We adopt the standard definition due to Alur and Dill [AD94].

Definition 7. *(Region Equivalence denoted $\sim$) For a clock $x \in \mathcal{C}$, let C_x be a constant (the ceiling of clock x). For a real number t, let $\{t\}$ denote the fractional part of t, and $\lfloor t \rfloor$ denote its integer part. For clock assignments $u, v \in \mathcal{V}$, u, v are region-equivalent denote $u \sim v$ iff*

1. *for each clock x, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or $(u(x) > C_x$ and $v(x) > C_x)$, and*

[12] This condition may be relaxed (e.g. the clock values may be negative, but bounded with a lower bound). But this is precisely what we need for the main proof.

2. *for all clocks* x, y *if* $u(x) \leq C_x$ *and* $u(y) \leq C_y$ *then*
   (a) $(\{u(x)\} = 0$ *iff* $\{v(x)\} = 0$ *and*
   (b) $(\{u(x)\} \leq \{u(y)\}$ *iff* $\{v(x)\} \leq \{v(y)\})$

It is known that region equivalence is preserved by the delay (addition) and reset. In the following, we establish that region equivalence is also preserved by subtraction for clocks that are bounded as defined in Definition 6. For a clock assignment u, let $u(x - C)$ denote the assignment: $u(x - C)(x) = u(x) - C$ and $u(x - C)(y) = u(y)$ for $y \neq x$. The following results states that $\sim$ is a bisimulation with respect to the operations: addition, reset and the conditional subtraction.

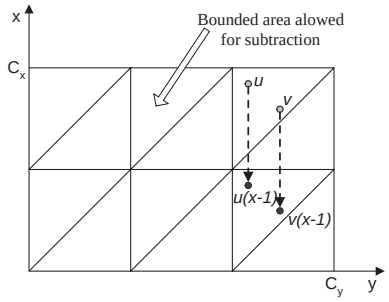


Fig. 3. Region equivalence preserved by subtraction when clocks are bounded.

Lemma 1. *Let* $u, v \in \mathcal{V}$. *Then* $u \sim v$ *implies*

1. $u + t \sim v + t$ *for a positive real number* t, *and*
2. $u[x \mapsto 0] \sim v[x \mapsto 0]$ *for a clock* x *and*
3. $u(x - C) \sim v(x - C)$ *for all natural numbers* C *such that* $C \leq u(x) \leq C_x$. *for a natural number* C.

Proof. The proof for the first two items can be found in the literature e.g. [LW97].

We check the third according to the definition of regioin equivalence. It is illustrated in Figure 3 that the equivalence is preserved by subtraction in the bounded area. Assume that $u \sim v$, and for each clock x we also have $C \leq u(x) \leq C_x$. Note that because $u \sim v$, and $C \leq u(x)$, we have $C \leq v(x)$. Then we have $u(x - C)(y) \geq 0$ and $v(x - C)(y) \geq 0$ for any clock y. Now we have two cases to check:

1. If $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$, obviously we have $\lfloor u(x) - C \rfloor = \lfloor v(x) - C \rfloor$ for a natural number C and then $\lfloor u(x - C) \rfloor = \lfloor v(x - C) \rfloor$ by definition.
2. Note that the subtraction operation on clocks does not change the fractional parts of clock values. Therefore for all clocks y and z, we have:
   (a) $(\{u(x - C)(y)\} = 0$ iff $\{v(x - C)(y)\} = 0$ and
   (b) $(\{u(x - C)(y)\} \leq \{u(x - C)(z)\}$ iff $\{v(x - C)(y)\} \leq \{v(x - C)(z)\})$

In fact, region equivalence over clock assignments induces a bisimulation over reachable states of automata, which can be used to partition the whole state space as a finite number of equivalence classes.

Lemma 2. *Assume a bounded timed automaton with subtraction, a location l and clock assignments u and v. Then $u \sim v$ implies that*

1. *whenever* $(l,u) \longrightarrow (l',u')$ *then* $(l,v) \longrightarrow (l',v')$ *for for some* v' *s.t.* $u' \sim v'$ *and*
2. *whenever* $(l,v) \longrightarrow (l',v')$ *then* $(l,u) \longrightarrow (l',u')$ *for for some* u' *s.t.* $u' \sim v'$.

Proof. It follows from Lemma 1.

Note that the above lemma essentially states that if $u \sim v$ then (l,u) and (l,v) are bisimular, which implies the following result.

Lemma 3. *The location reachability problem for bounded timed automata with subtraction, whose clocks are bounded with known maximal constants is decidable.*

Proof. Because each clock of the automaton is bounded by a maximal constant, it follows from lemma 2 that for each location l, there is a finite number of equivalence classes of states which are equivalent in the sense that they will reach the same equivalence classes of states. Because the number of locations of an automaton is finite, the whole state space of an automaton can be partitioned into finite number of such equivalence classes.

4.2 Encoding of Schedulability as Reachability

Assume an automaton A extended with tasks, and a *preemptive scheduling* strategy Sch. The aim is to check if A is schedulable with Sch. As for the case of *non-preemptive scheduling* (Theorem 1), we construct $E(A)$ and $E(\mathsf{Sch})$, and check a pre-defined error-state in the product automaton of the two. The construction is illustrated in figure 4.

$E(A)$ is constructed as a timed automaton which is exactly the same as for the non-preemptive case (Theorem 1) and $E(\mathsf{Sch})$ will be constructed as a timed automaton with subtraction.

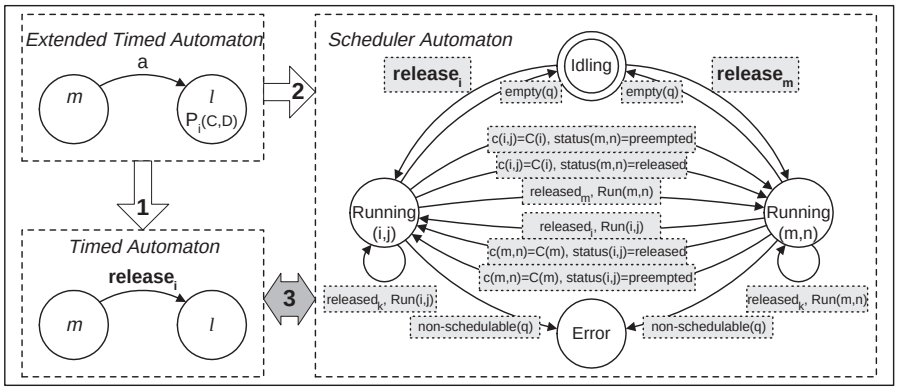


Fig. 4. Encoding of schelulability problem.

We introduce some notation. We use $C(i)$ and $D(i)$ to stand for the worst case execution time and relative deadline respectively for each task type P_i. We use P_{ij} to denote the jth instance of task type P_i.

For each task instance P_{ij}, we have the following state variables:

$\mathsf{status}(i,j)$ initialized to free. Let $\mathsf{status}(i,j) = \mathsf{running}$ stand for that P_{ij} is executing on the processor, $\mathsf{preempted}$ for that P_{ij} is started but not running, and $\mathsf{released}$ for that P_{ij} is released, but not started yet. We use $\mathsf{status}(i,j) = \mathsf{free}$ to denote that P_{ij} is not released yet or position (i,j) of the task queue is free.

Note that according to the definition of scheduling strategy, for all i, there should be only one j such that $\mathsf{status}(i,j) = \mathsf{preempted}$ (only one instance of the same task type is started), and for all i,j, there should be only one pair (k,l) such that $\mathsf{status}(k,l) = \mathsf{running}$ (only one is running for a one-processor system).

We need two clocks for each task instance:

1. $c(i,j)$ (a computing clock) is used to remember the accumulated computing time since P_{ij} was started (when $\mathsf{Run}(i,j)$ became true) [13], and subtracted with $C(k)$ when the running task, say P_{kl}, is finished if it was preempted after it was started.
2. $d(i,j)$ (a deadline clock) is used to remember the deadline and reset to 0 when P_{ij} is released.

We use a triple $\langle c(i,j), d(i,j), \mathsf{status}(i,j)\rangle$ to represent each task instance, and the task queue will contain such triples. We use q to denote the task queue. Note that the maximal number of instances of P_i appearing in a schedulable queue is $\lceil D(i)/C(i)\rceil$. We have a bound on the size of queue as claimed earlier, which is $\sum_{P_i \in \mathcal{P}} \lceil D(i)/C(i)\rceil$. We shall say that queue is empty denoted $\mathsf{empty}(q)$ if $\mathsf{status}(i,j) = \mathsf{free}$ for all i,j.

For a given scheduling strategy Sch, we use the predicate $\mathsf{Run}(m,n)$ to denote that task instance P_{mn} is scheduled to run according to Sch. For a given Sch, it can be coded as a constraint over the state variables. For example, for EDF, $\mathsf{Run}(m,n)$ is the conjunction of the following constraints:

1. $d(k,l) \leq D(k)$ for all k,l such that $\mathsf{status}(k,l) \neq \mathsf{free}$: no deadline is violated yet
2. $\mathsf{status}(m,n) \neq \mathsf{free}$: P_{mn} is released or preempted
3. $D(m) - d(m,n) \leq D(i) - d(i,j)$ for all (i,j): P_{mn} has the shortest deadline

$E(\mathsf{Sch})$ contains three type of locations: Idling, $\mathsf{Running}$ and Error with $\mathsf{Running}$ being parameterized with (i,j) representing the running task instance.

1. Idling denotes that the task queue is empty.
2. $\mathsf{Running}(i,j)$ denotes that task instance P_{ij} is running, that is, $\mathsf{status}(i,j) = \mathsf{running}$. We have an invariant for each $\mathsf{Running}(i,j)$: $c(i,j) \leq C(i)$ and $d(i,j) \leq D(i)$.
3. Error denotes that the task queues are non-schedulable with Sch.

There are five types of edges labeled as follows:

1. Idling to $\mathsf{Running}(i,j)$: there is an edge labeled by

[13] In fact, for each task type, we need only one clock for computing time because only one instance of the same task type may be started.

- guard: none
- action: $\mathsf{release}_i$
- reset: $c(i,j) := 0$, $d(i,j) := 0$, and $\mathsf{status}(i,j) := \mathsf{running}$

2. $\mathsf{Running}(i,j)$ to Idling: there is only one edge labeled with
   - guard: $\mathsf{empty}(q)$ that is, $\mathsf{status}(i,j) = \mathsf{free}$ for all i,j (all positions are free).
   - action: none
   - reset: none
3. $\mathsf{Running}(i,j)$ to $\mathsf{Running}(m,n)$: there are two types of edges.
   (a) The running task P_{ij} is finished, and P_{mn} is scheduled to run by $\mathsf{Run}(m,n)$. There are two cases:
      i. P_{mn} was preempted earlier:
         - guard: $c(i,j) = C(i)$, $\mathsf{status}(m,n) = \mathsf{preempted}$ and $\mathsf{Run}(m,n)$
         - action: none
         - reset: $\mathsf{status}(i,j) := \mathsf{free}$, $\{c(k,l) := c(k,l) - C(i) | \mathsf{status}(k,l) = \mathsf{preempted}\}$, and $\mathsf{status}(m,n) := \mathsf{running}$
      ii. P_{mn} was released, but never preempted (not started yet):
         - guard: $c(i,j) = C(i)$, $\mathsf{status}(m,n) = \mathsf{released}$ and $\mathsf{Run}(m,n)$
         - action: none
         - reset: $\mathsf{status}(i,j) := \mathsf{free}$, $\{c(k,l) := c(k,l) - C(i) | \mathsf{status}(k,l) = \mathsf{preempted}\}$, $c(m,n) := 0, d(m,n) := 0$ and $\mathsf{status}(m,n) := \mathsf{running}$
   (b) A new task P_{mn} is released, which preempts the running task P_{ij}:
      - guard: $\mathsf{status}(m,n) = \mathsf{free}$, and $\mathsf{Run}(m,n)$
      - action: $\mathsf{released}_m$
      - reset: $\mathsf{status}(m,n) := \mathsf{running}$, $c(m,n) := 0$, $d(m,n) := 0$, and $\mathsf{status}(i,j) := \mathsf{preempted}$
4. $\mathsf{Running}(i,j)$ to $\mathsf{Running}(i,j)$. There is only one edge representing the case when a new task is released, and the running task P_{ij} will continue to run:
   - guard: $\mathsf{status}(k,l) = \mathsf{free}$, and $\mathsf{Run}(i,j)$
   - action: $\mathsf{released}_k$
   - reset: $\mathsf{status}(k,l) := \mathsf{released}$ and $d(k,l) := 0$
5. $\mathsf{Running}(i,j)$ to Error: for each pair (k,l), there is an edge labeled by $d(k,l) > D(k)$ and $\mathsf{status}(k,l) \neq \mathsf{free}$ meaning that the task P_{kl} which is released (or preempted) fails in meeting its deadline.

The third step of the encoding is to construct the product automaton $E(Sch) \parallel E(A)$ in which both $E(Sch)$ and $E(A)$ can only synchronize on identical action symbols. Now we show that the product automaton is bounded.

Lemma 4. *All clocks of* $E(\mathsf{Sch})$ *in* $E(Sch) \parallel E(A)$ *are bounded and non negative.*

Proof. All computing clocks $c(k,l)$ are bounded by $D(k)$. This is due to the fact that all edges labeled with a subtraction is guarded by the constraint: $\mathsf{Run}(m,n)$ which requies $d(k,l) \leq D(k)$ (no deadline is violated in order to stay in $\mathsf{Running}$). $d(k,l) \leq D(k)$ implies that $c(k,l) \leq D(k)$ because $c(k,l)$ and $d(k,l)$ are always reset to zero at the same time when a new instance of P_k is released. Thus $c(k,l)$ is bounded.

Secondly, the only possibility for a computing clock, say $c(k,l)$ for task P_{kl}, to become negative is by subtractions. But a subtraction is done on $c(k,l)$ only when a task, say P_{ij}, is finished i.e. $c(i,j) = C(i)$ holds. As $c(i,j)$ is reset to zero before $c(k,l)$ is reset to zero ($\mathsf{status}(k,l) = \mathsf{preempted}$ implying that P_{kl} was released and preeempted earlier), we have $c(i,j) \geq c(k,l)$ implying that $c(i,j) - C(k) \geq 0$. Thus all clocks are non-negative.

Now we have the correctness theorem for our encoding. Assume, without losing generality, that the initial task queue of an automaton is empty.

Lemma 5. *Let A be an extended timed automaton and* Sch *a scheduling strategy. Assume that (l_0, u_0, q_0) and $(\langle l_0, \mathsf{Idling}\rangle, u_0)$ are the initial states of A and the product automaton $E(A)||E(\mathsf{Sch})$ respectively where l_0 is the initial location of A, u_0 and v_0 are clock assignements assigning all clocks with 0 and q_0 is the empty task queue. Then for all l, u, v:*

$$(l_0, u_0, q_0)(\longrightarrow)^*(l, u, \mathsf{Error}) \text{ iff } (\langle l_0, \mathsf{Idling}\rangle, u_0 \cup v_0)(\longrightarrow)^*(\langle l, \mathsf{Error}\rangle, u \cup v)$$

Proof. It is by induction on the length of transition sequence (i.e. reachability steps).

The above lemma states that the schedulability analysis problem can be solved by solving a reachability problem for timed automata extended with subtraction. From Lemma 4, we know that $E(\mathsf{Sch})$ is bounded. Because the reachability problem is decidable due to Lemma 3, we complete the proof for our main result stated in Theorem 2.

5 Conclusions and Related Work

We have studied a model of timed systems (timed automata extended with tasks), which unifies timed automata with the classic task models from scheduling theory. The model can be used to specify resource requirements and hard time constraints on computations, in addition to features offered by timed automata. It is general and expressive enough to describe concurrency and synchronization, and tasks which may be periodic, sporadic, preemptive and (or) non-preemptive. The classic notion of schedulability is nicely extended to automata model. Our main technical contribution is the proof that the schedulability checking problem is decidable. The problem has been conjectured to be undecidable for years. To our knowledge, this is the first decidability result for preemptive scheduling in dense-time models. Based the results, we have developed a symbolic schedulability checking algorithm based using the DBM techniques. It has been implemented in a tool for modeling and scheduling of timed systems. As future work, we shall study the schedule synthesis problem. More precisely given an automaton, it is desirable to characterize the set of schedulable traces accepted by the automaton.

Related work. Scheduling is a well-established area. A numerous number of analysis methods have been published in the literature. For systems restricted to periodic tasks, algorithms such as rate monotonic scheduling and earliest-deadline first are widely used and exact analysis methods exist, see e.g. [KS97]. These methods can be extended to handle non-periodic tasks by considering them as periodic with the minimal inter-arrival time as the task periods. Our work is more related to work on using automata to model and solve scheduling problems. In several papers, e.g. [CL00,Cor94,AGS00], stopwatch automata [ACH+95] are applied to model scheduling algorithms with sporadic tasks. However, as reachability analysis for stopwatch automata is undecidable, this approach can not be applied in general to solve scheduling problems. Approximative and semi-decision algorithms exits but they can only be used to prove that a system is not schedulable (inconclusive for schedulability) and they do not guarantee termination. In [MV94], McManis and Varaiya presents a restricted class of stopwatch automata, called suspension automata, and give a set of conditions for

which reachability analysis of suspension automata is decidable. Unfortunately, the given conditions restrict the applicability of the result as scheduling algorithms such as rate-monotonic and earliest-deadline first, can not be analysed. The work presented in this paper is more general and we show that the presented analysis is guaranteed to terminate for all inputs. Another model for timed system is presented in [AGS00]. It is designed to be compositional and suitable for describing priority-driven scheduling algorithms. The authors show that a number of scheduling algorithms, with or without preemption, can be modeled. However, the problem of schedulability analysis is not addressed in the paper. Related to this is the work on integrating specification and scheduler generation of real-time systems, presented in [AGP+99]. The modeling language, which can be used to describe both sporadic and preemptive tasks, is a Petri-net model with time. The authors presents a controller synthesis technique that can be used to construct an online non-preemptive scheduler that satisfies all given constraints in the model. The problem addressed is restricted to non-preemptive schedulers. In this paper, we use the idea of replacing suspensions of timers by subtraction of clock values, as suggested in [MV94]. It has been shown in [BDFP00] that updating clock variables by subtraction of integer values in timed automata is undecidable in general. We identify a decidable class of such updatable automata, which is precisely what we need to solve scheduling problems.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AGP+99] Karine Altisen, Gregor Gößler, Amir Pnueli, Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. A framework for scheduler synthesis. In *IEEE Real-Time Systems Symposium*, pages 154–163, 1999.

[AGS00] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, Pune, India*, 2000.

[BDFP00] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Expressiveness of updatable timed automata. In *Proc. 25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000), Bratislava, Slovakia, Aug. 2000*, volume 1893, pages 232–242. Springer, 2000.

[BGK+96] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In *Computer Aided Verification*, pages 244–256, 1996.

[BLL+96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool suite for automatic verification of real-time systems. *Lecture Notes in Computer Science*, 1066:232–243, 1996.

[But97] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications.* Kulwer Academic Publishers, 1997.

[CL00] F. Cassez and F. Laroussinie. Model-checking for hybrid systems by quotienting and constraints solving. In *Proceedings of the twelfth International Conference on Computer-Aided Verification*, volume 1855, pages 373–388, Standford, California, USA, 2000. Springer-Verlag.

[Cor94] J. Corbett. Modeling and analysis of real-time ada tasking programs. In *Proceedings Real-Time Systems Symposium, IEEE Computer Society Press*, pages 132–141, 1994.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th IEEE Real-Time Systems Symposium,* Pisa, Italy, pages 66–75, 1995.

[EWY98] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of Nordic Workshop on Programming Theory*, 1998.

[HK89] Hans Hüttel and Kim G. Larsen. The use of static constructs in a modal process logic. *Logic at Botik*, 363:163–180, 1989.

[KS97] C. M. Krishna and K. G. Shin. *Real-Time Systems.* McGraw-Hill, 1997.

[LP97] H. Lönn and P. Pettersson. Formal verification of a tdma protocol startup mechanism. In *IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, 1997.

[LPY95] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 76–89, 1995.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 281–297, 1998.

[LW97] Kim Guldstrand Larsen and Yi Wang. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, 1997.

[MV94] J. McManis and P. Varaiya. Suspension automata: A decidable class of hybrid automata. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification*, volume 818, pages 105–117, Standford, California, USA, 1994. Springer-Verlag.

[YPD94] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proc. of the 7th International Conference on Formal Description Techniques*, 1994.

Integrating Reliability and Timing Analysis of CAN-based Systems*

Hans Hansson, Thomas Nolte, Christer Norström and Sasikumar Punnekkat[1]
Mälardalen Real-Time Research Centre
Department of Computer Engineering
Mälardalen University, Västerås, SWEDEN
http://www.mrtc.mdh.se

Abstract— **This article presents and illustrates a reliability analysis method developed with focus on Controller Area Network (CAN) based automotive systems. The method considers the effect of faults on schedulability analysis and its impact on the reliability estimation of the system, and attempts to integrate both to aid system developers. We illustrate the method by modeling a simple distributed antilock braking system, and showing that even in cases where the worst-case analysis deem the system unschedulable, it may be proven to satisfy its timing requirements with a sufficiently high probability. From a reliability and cost perspective, this article underlines the tradeoffs between timing guarantees, the level of hardware and software faults, and per-unit cost.**

I. Introduction

During the last decade or so, real-time researchers have extended schedulability analysis to a mature technique which for non-trivial systems can be used to determine whether a set of tasks executing on a single CPU or in a distributed system will meet their deadlines or not [3][4][5]. The main focus of the real-time research community is on hard real-time systems, and the essence of analysing such systems is to investigate if deadlines are met in a worst case scenario. Whether this worst case actually will occur during execution, or if it is likely to occur, is not normally considered.

Reliability modeling, on the other hand, involves study of fault models, characterization of distribution functions of faults and development of methods and tools for composing these distributions and models in estimating an overall reliability figure for the system.

In [6] we present a model for calculating worst-case latencies of messages under error assumptions for the Controller Area Network (CAN). In many situations, this analysis might infer that a given message set is not feasible under worst case fault interferences. Such a result, though correct, is of limited help to the system designers except to prompt them to overdesign the system and waste resources to tackle a situation, which might never happen during the life time of the system.

When performing schedulability analysis (or any other type of formal analysis) it is important to keep in mind that the analysis is only valid under some specific model assumptions, typically under some assumed "normal condition", e.g., no hardware failures and a "friendly" environment. The "abnormal" situations are typically catered for in the reliability analysis, where probabilities for failing hardware and environmental interferences are combined into a system reliability measure. This separation of deterministic (0/1) schedulability analysis and stochastic reliability analysis is a natural simplification of the total analysis, which unfortunately is quite pessimistic, since it assumes that the "abnormal" is equivalent to failure. Especially for transient errors/failures, this may not at all be the case.

Consider, for instance, occasional external interference on a communication link. The interference will lead to transmission errors and subsequent retransmission of messages. The effect will be increased message latencies which may lead to missed deadlines, especially if the interference coincides with the worst case message transmission scenario considered when performing schedulability analysis. In other scenarios, the interference will not increase the worst case message latency, as illustrated in Figure 1. The figure shows a system with 3 periodic messages M_1, M_2 and M_3, with the parameters shown in Table I. Assuming an overhead, $O = 1$ for error signaling and recovery (but not including retransmission of the corrupted message), we have shown the effects of 3 different scenarios, corresponding to an external interference hitting the system at different points in time. In the first case the error caused by the interference results in a retransmission of M_1, causing M_2 and M_3 to miss their deadlines. In the second case, though a re-transmission is necessitated, still the message set meets its deadlines, whereas in the third scenario, the error has no effect at all since it falls in a period of inactivity of bus.

This simple example shows that there are situations (scenarios) when system requirements (e.g. deadlines) are not violated by the "abnormal". Hence, there is a potential for obtaining a more accurate and tight reliability analysis by considering the likelihood of the "abnormal" actually causing a deadline violation. The basic argument of our work, is that for any system (even the most safety critical one) the analysis is only valid as long as the underlying assumptions hold. A system can only be guaranteed up to some level, after which we must resort to reliability analysis. The main contribution of this article is in providing a method-

* This article extends and improves preliminary results presented in [1] and [2], by refining the fault model and providing a more rigorous treatment.

1 Vikram Sarabhai Space Centre, Trivandrum, India. Work performed while on leave at Mälardalen University.

Msg ID	Period	Deadline	Transmission time	Priority
M_1	5	5	2	High
M_2	10	7	1	Medium
M_3	20	8	1	Low

TABLE I
MESSAGE SET PARAMETERS

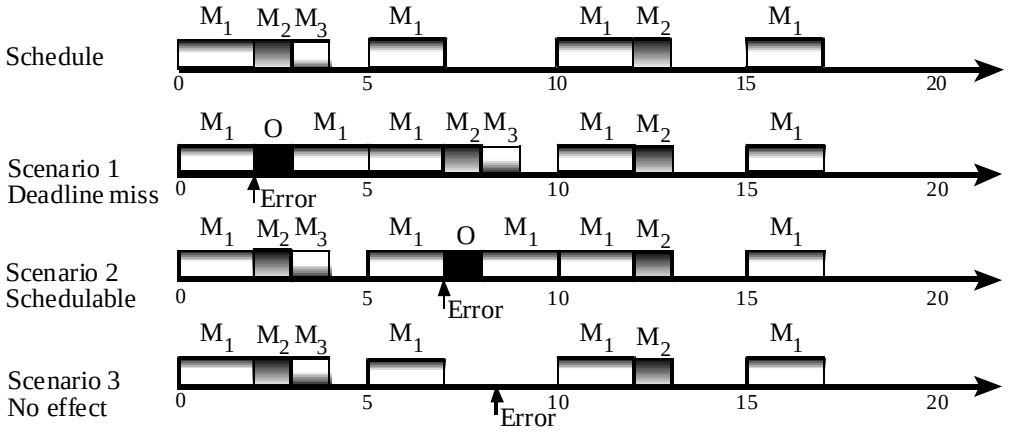


Fig. 1. Dependency of Effects of Faults on Phasings

ology to calculate such an estimate by way of integrating schedulability and reliability analysis.

Though we use automotive examples throughout the article, it should be pointed put that our results have substantially wider applicability, both in the sense that CAN is used in other application domains, such as factory automation, and that the general approach is applicable also to other communication systems. Furthermore, our type of reasoning is especially pertinent considering the growing trend of using wireless networks in factory automation and elsewhere. The error behaviour of such systems will most likely require reliability to be incorporated into the analysis of timing guarantees.

The outline of the article is as follows. Section II presents general reliability modelling for distributed real-time systems and introduces our approach. Section III specifically discusses the scheduling of message sets in Controller Area Networks under a general fault model, presents schedulability analysis for the model, and introduces a simulation based approach for analysis of arbitrary samples of phasings and interferences. Section IV illustrates our results, with a sample message set used in a distributed computer network inside passenger cars. Section V discusses possible extensions and presents some conclusions.

II. Reliability Modelling

Reliability is defined as the probability that a system can perform its intended function, under given conditions, for a given time interval. In the context of an automobile its intended function is to provide reliable and cost-effective transport of men and material. At a subsystem level, such as for an Antilock Braking System (ABS) for automobiles, this boils down to performing the tasks (mainly input_sensors, compute_control, and output_actuators etc.,) as per the specifications. Being part of a real-time system, the specifications for ABS, imply the necessity for the results to be both functionally correct and within timing specifications.

A major issue here is how to compose hardware reliability, software reliability, environment model, and timing correctness to arrive at reasonable estimates of overall system reliability (Fig 2).

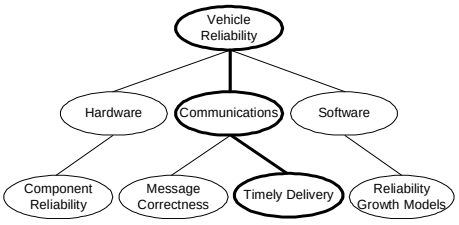


Fig. 2. System Reliability: A Top-Down View

Let us define

$$
\begin{aligned}
p_{HF}(t) &= Probability(\text{Hardware failure at } t) \\
p_{SF}(t) &= Probability(\text{Software failure at } t) \\
p_{F}(t) &= Probability(\text{Communication failure at } t)
\end{aligned}
$$

The reliability of the system, $R(t)$, is the probability that the system performs all its intended functions correctly for a period t. This is given by the product of cumulative probabilities that there are no failures in hardware, software and communication subsystem during the period $(0, t)$. That is,

$$R(t) = \left(1 - \int_0^t p_{HF}(t)\right)\left(1 - \int_0^t p_{SF}(t)\right) \times \times \left(1 - \int_0^t p_F(t)\right) \quad (1)$$

In this article, we concentrate only on the final term in Equation 1, i.e., the probability that no errors occur in the communication subsystem. Please note that, when we talk about communication subsystem, we are not concentrating on the faults in the hardware (as in Pinho et al. [7]) or in the software of such a system. Instead we look at it from a system level and treat its correctness as the probability of correct and timely delivery of message sets. Since the main cause for an incorrect (corrupted, missing or delayed) message delivery is environmental interferences, an appropriate modelling of such factors in the context of the environment in which the system will operate is essential for performing reliability analysis. A completely accurate modeling of the probability of timely delivery of messages is far from being trivial and hence we have made certain simplifying assumptions in order to divide the problem into manageable proportions.

A. Problem Statement

The premises of our problem (from a designer's perspective) are:

- We are given the overall reliability requirements of the system (say for example, a vehicle), from which we derive the reliability requirement of a particular subsystem (say, the computer system controlling pedal brake and ABS), which in turn defines a requirement on the reliability of its communication subsystem.
- The given requirements of the subsystem, after a series of design steps, get converted to a set of relevant tasks along with their timing properties. In our example of cars, the overall vehicle level safety and performance requirements, in conjunction with the vehicle dynamics and properties of subsystems allow us to have separation of concerns between hydraulic/mechanical systems and the electrical system. In the next step we convert these requirements and constraints into the timing specifications for the individual tasks and messages.
- We have an environment model and know how to perform response time analysis of CAN messages under normal and error conditions [6].

The problem analysed is, given the above information, how to find a suitable way of predicting the reliability of the communication subsystem. The simple approach will be to give a 0/1 weight to the schedulability aspect in evaluating the system 'correctness' and calculate the reliability. However, the environment model provides worst case scenarios, which may not occur in practice and its impact may depend on the actual phasing of messages and the way in which they interact with the environment/fault model. So, the major issues are:

- Can we partition the schedulability analysis under faults by considering a set of scenarios corresponding to different message and fault model phasings? (As illustrated in Figure 1; and where the worst case considered in our previous analysis [6] is only one of several scenarios.)
- How can we use such an analysis to obtain a more accurate reliability estimate?

B. Reliability Estimation

By definition, reliability is specified over a mission time. Normally we can assume a repetitive pattern of messages (over the least common multiple (LCM) of the individual message periods). Each LCM is typically a very small fraction of the mission time.

We will now outline a methodology for estimating communication failures due to external interference in a CAN-bus. It should be noted that, the methodology presented in this section can easily be applied to other communication models as well.

Let t represent an arbitrary time point when the external interference hits the bus and causes an error. If we can assume zero error latency and instantaneous error detection then t becomes the time point of detection of an error in the bus due to external interferences.

We now define the following probabilities:

$$\begin{aligned} p_I(t) &= Probability(\text{Interference at } t) \\ p_C(t) &= Probability(\text{Msg corruption}|\text{Interference at } t) \\ p_M(t) &= Probability(\text{Deadline miss}|\text{Interference at } t) \end{aligned}$$

By relying on the extensive error detection and handling features available in CAN, we can safely assume that an error in message corruption is either detected and corrected by re-transmission or will ultimately result in a timing error. This allows us to ignore $p_C(t)$ and define in our context, the probability of communication failure due to an interference starting at t as:

$$p_F(t) = p_M(t) \times p_I(t) \quad (2)$$

In the environment model in [6], we have assumed the possibility of an interference I_1, having a certain pattern hitting the message transmission. Let $p_{I_1}(t)$ be the probability of such an event occurring at time t. We also assume that another interference I_2, having a different pattern, can hit the system at time t with a probability, say, $p_{I_2}(t)$. In [6], we assumed that both these interferences hit the message transmission in a worst case manner and looked at their impact on the schedulability. In this article, we will increase the realism by relaxing the requirement on the worst-case phasing between schedule and interference. It should be noted that there is an implicit assumption that these interferences are independent.

C. Failure Semantics

In the above presentation we assume that a single deadline miss causes a failure. This may be true for many systems, but in general, the *failure semantics* does not have to be restricted to this simple case. For instance, most control engineers would require the system to be more robust,

i.e., the system should not loose stability if single deadlines, or even multiple deadlines, are missed. A tolerable failure semantics for such a system [8] could for instance be "3 consecutive deadlines missed or 5 out of 50 deadlines missed". Such a definition of failure is more realistic and also leads to a substantial increase in reliability estimates, as compared to the single deadline miss case. To simplify the presentation we will, however, stick to the simple "single missed deadline" failure semantics for the time being, but return to this issue in the reliability analysis in Section IV-C.

D. Calculating Failure Probabilities

To calculate the subsystem reliability, first we need to calculate the failure probability (in our case of the communication subsystem subject to possible external interference), i.e., the probability of at least one failure (defined as a missed deadline) during the mission time. In doing this we assume that the interference free system is schedulable, i.e. that it meets all deadlines with probability 1. This can for a CAN-bus be verified by using the analysis presented in Section III-A. Furthermore, due to the bit wise behaviour of the CAN-bus, we can with respect to the external interference make a discretization of the time scale, with the time unit corresponding to a single bit time τ_{bit} ($1\mu s$ for a 1Mbps bus), i.e., we make no distinction between an interference hitting the entire bit or only a fraction of it. In either case, the corruption will both occur and be detected.

We will distinguish between two types of interference sources:

- *Intermittent sources*, which are bursty sources that interfere during the entire mission time $\mathcal{T}$, and are for an interference source I characterized by a period T_I and a burst length l_I (where $l_I < T_I$), as illustrated in Figure 3.
- *Transient sources*, which are bursty sources of limited duration. These occur at most once during a mission time $\mathcal{T}$, and are for an interference source J characterized by a period T_J, a burst length l_J, and a number of bursts n_J (where $l_J < T_J$ and $T_J * n_J < \mathcal{T}$), as illustrated in Figure 4.

For a single intermittent source I we define the probability of a communication failure during the mission time as follows:

$$P_F^{\mathcal{T}}(I) = \sum_{t \in [0, T_I - 1]} p_I(t) \times P_F^{\mathcal{T}}(I | h(I) = t) \qquad (3)$$

where $P_F^{\mathcal{T}}(I|h(I) = t)$ denotes the conditional probability of a communication failure, given that the system was hit by interference from source I at time t, denoted by $h(I) = t$. It follows, since interference and bus communication are independent, that $p_I(t) = \frac{1}{T_I}$. To calculate $P_F^{\mathcal{T}}(I|h(I) = t)$ we will use simulation, i.e., we will simulate the bus traffic during a mission, including the effects of interference, to determine if any communication failure (deadline violation) occurs. This will for each t result in either 0, if no deadline is missed, or 1, if a deadline miss is detected.

For a single transient source J we define the probability of a communication failure during the mission time as follows:

$$P_F^{\mathcal{T}}(J) = \sum_{t \in [-n_J T_J, \mathcal{T} - 1]} p_J(t) \times P_F^{\mathcal{T}}(J | s(J) = t) \qquad (4)$$

where $p_J(t)$ denotes the probability that the first transient interference hits the system at time t, and $P_F^{\mathcal{T}}(J|s(J) = t)$ denotes the conditional probability of a communication failure during $\mathcal{T}$ due to interference from J, given that the interference J starts at time t. In the above summation all possible full or partial interferences from the transient source during mission time are considered. The interval $[-n_J T_J, 0]$ specifically captures initial partial interferences, starting before mission time, but ending during mission. It follows, since interference and bus communication are independent, that $p_J(t) = \frac{1}{\mathcal{T} + n_J T_J}$. To calculate $P_F^{\mathcal{T}}(J|s(J) = t)$ we will use simulation, just as above. The number of scenarios to simulate here is potentially much larger than for an intermittent source, since typically $\mathcal{T} >> T_I$. However, the number of simulations can be reduced, since the probability of failure is independent of the LCM, in which the interference hits the system. We can prove that

$$\begin{aligned} P_F^{\mathcal{T}}(J) \quad \leq \\ & \frac{\mathcal{T} + n_J T_J}{LCM} \times \sum_{t \in [0, LCM - 1]} p_J(t) \times P_F^{\mathcal{T}}(J | s(J) = t) \\ = \quad & \frac{\sum_{t \in [0, LCM - 1]} P_F^{\mathcal{T}}(J | s(J) = t)}{LCM} \end{aligned} \qquad (5)$$

It should be noted that we introduce a slight pessimism (which is the reason for the $\leq$) since the probability for failure in $\mathcal{T}$ is lower towards the end, when the interference bursts are extending past the end of $\mathcal{T}$. However, since we assume $n_J \times T_J << \mathcal{T}$, the introduced pessimism can be considered negligible.

Also, note that in (5) the effects of the interference starting before the LCM is covered by the interference on the subsequent LCM, which is the reason for starting the summation with $t = 0$ (rather than $t = -n_j T_j$ as in (4)).

Finally, note that by not counting the interferences starting outside the very first LCM (at the beginning of the mission time), we introduce some optimism, but since the assumption of $n_J \times T_J << \mathcal{T}$ this optimism is insignificant. To be more precise a fraction of (4), $\sum_{t \in [-n_J T_J, 0]} p_J(t) \times P_F^{\mathcal{T}}(J|s(J) = t)$, could be added to (5) in order to cover for the pre-mission time trigged transient faults.

We now extend the above basic analysis to the analysis of multiple sources of interference. First, we consider the case of two independent sources of interference. There are three cases to consider:

1. Two intermittent sources I_1 and I_2:

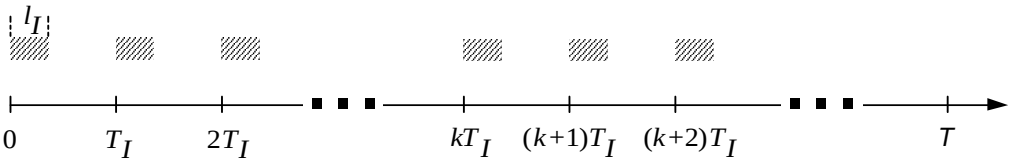


Fig. 3. Pattern of interference from an intermittent source, with burst length l_I and period T_I.

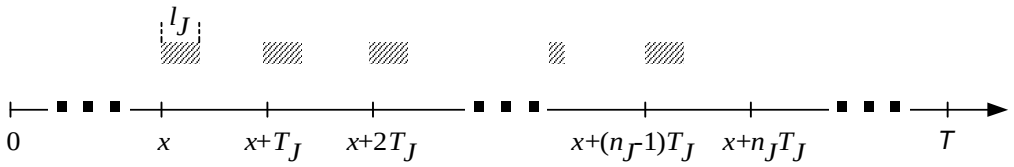


Fig. 4. Pattern of interference from a transient source, hitting the system at time x, with burst length l_J, period T_J and consisting of n_J bursts.

$$P_F^{\mathcal{T}}(I_1,I_2) = \sum_{\substack{t_1\in[0,T_{I_1}-1]\\ t_2\in[0,T_{I_2}-1]}} p_{I_1}(t_1)\times p_{I_2}(t_2)\times \times P_F^{\mathcal{T}}(I_1,I_2|h(I_1)=t_1 \ \wedge \ h(I_2)=t_2) \quad (6)$$

2. Two transient sources J_1 and J_2:

$$P_F^{\mathcal{T}}(J_1,J_2) = \sum_{\substack{t_1\in[-n_{J_1}T_{J_1},\mathcal{T}-1]\\ t_2\in[-n_{J_2}T_{J_2},\mathcal{T}-1]}} p_{J_1}(t_1)\times p_{J_2}(t_2)\times \times P_F^{\mathcal{T}}(J_1,J_2|s(J_1)=t_1 \ \wedge \ s(J_2)=t_2) \quad (7)$$

3. One intermittent and one transient source:

$$P_F^{\mathcal{T}}(I_1,J_2) = \sum_{\substack{t_1\in[0,T_{I_1}-1]\\ t_2\in[-n_{J_2}T_{J_2},\mathcal{T}-1]}} p_{I_1}(t_1)\times p_{J_2}(t_2)\times \times P_F^{\mathcal{T}}(I_1,J_2|h(I_1)=t_1 \ \wedge \ s(J_2)=t_2) \quad (8)$$

The number of scenarios to simulate for the above three cases are in the order of $T_{I_1}\times T_{I_2}$, $\mathcal{T}^2$, and $T_{I_1}\times\mathcal{T}$, respectively. This may be rather large, especially for the two latter cases. By observing symmetries in the formulas we can however reduce the number of scenarios. For case 3, consider the following two mutually exclusive situations:

- $LCM \geq T_{I_1}$, which leads to the following reduced formula:

$$P_F^{\mathcal{T}}(I_1,J_2) = \frac{\mathcal{T}+n_{J_2}T_{J_2}}{LCM}\times \sum_{\substack{t_1\in[0,T_{I_1}-1]\\ t_2\in[0,LCM-1]}} p_{I_1}(t_1)\times \times p_{J_2}(t_2)\times \times P_F^{\mathcal{T}}(I_1,J_2|h(I_1)=t_1 \ \wedge \ s(J_2)=t_2) \quad (9)$$

- $LCM < T_{I_1}$, which leads to the following reduced formula:

$$P_F^{\mathcal{T}}(I_1,J_2) = \frac{\mathcal{T}+n_{J_2}T_{J_2}}{T_{I_1}}\times \sum_{\substack{t_1\in[0,T_{I_1}-1]\\ t_2\in[0,T_{I_1}-1]}} p_{I_1}(t_1)\times \times p_{J_2}(t_2)\times \times P_F^{\mathcal{T}}(I_1,J_2|h(I_1)=t_1 \ \wedge \ s(J_2)=t_2) \quad (10)$$

The above two equations can be combined into:

$$P_F^{\mathcal{T}}(I_1,J_2) = \frac{\mathcal{T}+n_{J_2}T_{J_2}}{\max(LCM,T_{I_1})}\times \times \sum_{\substack{t_1\in[0,T_{I_1}-1]\\ t_2\in[0,\max(LCM,T_{I_1})-1]}} p_{I_1}(t_1)\times p_{J_2}(t_2)\times \times P_F^{\mathcal{T}}(I_1,J_2|h(I_1)=t_1 \ \wedge \ s(J_2)=t_2) \quad (11)$$

and thus, we have reduced the number of scenarios from the order of $T_{I_1}\times\mathcal{T}$ to the order of $T_{I_1}\times\max(LCM,T_{I_1})$.

Finally, we present the general formula for an arbitrary number of interference sources of either type (n intermittent and m transient sources of interference):

$$P_F^{\mathcal{T}}(I_1,\ldots,I_n,J_1,\ldots,J_m) = \quad (12)$$

$$\sum_{\substack{t_1\in[0,T_{I_1}-1]\\ \cdots\\ t_n\in[0,T_{I_n}-1]\\ t_1'\in[-n_{J_1}T_{J_1},\mathcal{T}-1]\\ \cdots\\ t_m'\in[-n_{J_m}T_{J_m},\mathcal{T}-1]}} \left(\begin{array}{c} p_{I_1}(t_1)\times \\ \cdots \\ \times p_{I_n}(t_n)\times \\ \times\, p_{J_1}(t_1')\times \\ \cdots \\ \times p_{J_m}(t_m')\times \end{array} \right. \times P_F^{\mathcal{T}} \left(\begin{array}{c} I_1,\ldots,I_n, \\ J_1,\ldots,J_m \mid \\ h(I_1)=t_1\wedge \\ \cdots \\ \wedge\, h(I_n)=t_n\wedge \\ \wedge\, s(J_1)=t_1'\wedge \\ \cdots \\ \wedge\, s(J_m)=t_m' \end{array} \right) \left. \vphantom{\begin{array}{c} p_{I_1}(t_1)\times \\ \cdots \\ \times p_{I_n}(t_n)\times \\ \times\, p_{J_1}(t_1')\times \\ \cdots \\ \times p_{J_m}(t_m')\times \end{array}} \right)$$

E. Approach to Analysis

The above equations define the probability of communication failure given a set of sources interfering with the communication according to specified patterns. In our modeling we will additionally specify the probability that an interference source actually is active during the mission. This gives us the following:

- We have a set of external sources of interference $\mathcal{K} = \mathcal{I} \cup \mathcal{J}$, where $\mathcal{I}$ is a set of intermittent sources of interference and $\mathcal{J}$ is a set of transient sources, as defined above.
- Each interference source $k \in \mathcal{K}$ is either active or inactive during a mission. The probability for the source to be active is P_k^{act}, and the probability for inactivity is consequently $1 - P_k^{act}$.

For a scenario with a single intermittent interference source I_1 and a single transient interference source J_2, we can now (with a slight generalization of the notation) define the probability of a communication failure in a mission as follows:

$$\begin{aligned} Q_F^{\mathcal{T}}(\{I_1, J_2\}) &= P_{I_1}^{act} \times P_{J_2}^{act} \times P_F^{\mathcal{T}}(I_1, J_2) + \\ &+ P_{I_1}^{act} \times (1 - P_{J_2}^{act}) \times P_F^{\mathcal{T}}(I_1) + \\ &+ (1 - P_{I_1}^{act}) \times P_{J_2}^{act} \times P_F^{\mathcal{T}}(J_2) \end{aligned} \quad (13)$$

which for an arbitrary set $\mathcal{I}$ of intermittent sources and an arbitrary set $\mathcal{J}$ of transient interference sources can be generalized to

$$\begin{aligned} Q_F^{\mathcal{T}}(\mathcal{I} \cup \mathcal{J}) &= \sum_{A \subseteq \mathcal{I} \cup \mathcal{J}} \left(\prod_{a \in A} P_a^{act} \right) \times \\ &\times \left(\prod_{b \in \overline{A}} (1 - P_b^{act}) \right) \times P_F^{\mathcal{T}}(A) \end{aligned} \quad (14)$$

where $\overline{A} = (\mathcal{I} \cup \mathcal{J}) \setminus A$. Intuitively, Equation 15 defines the failure probability for a system that is potentially subjected to interference from a set $\mathcal{I} \cup \mathcal{J}$ of interference sources as the sum of weighted probabilities that a failure occurs in each of the possible combination of sources.

The value of $P_F^{\mathcal{T}}(A)$ is, as mentioned above, obtained by simulation. This amounts to:

- For each intermittent interference source $I \in A$ make a selection from the set of discrete time points $[0, T_I - 1]$. Each picked sample s indicates the phasing of the corresponding source at time 0. The selected phasing will give a scenario with interferences from source I starting at s, $s + T_I$, $s + 2T_I, \ldots$, $s + nT_I$, where $n = \lceil \frac{\mathcal{T}}{T_I} \rceil$.
- For each transient interference source $J \in A$ we make a selection from the set of discrete time points $[-n_J T_J, \mathcal{T} - 1]$. Each picked sample indicates the time when the interference from the corresponding source hits the system.
- Perform the simulation (as detailed in Section III) to obtain $P_F^{\mathcal{T}}(A | \textit{selected phasings of interferences})$. The result will be either 1 or 0, corresponding to no communication failures or communication failure detected, respectively.

We can then calculate $Q_F^{\mathcal{T}}(A)$ using Equation 15.

For most realistic systems with multiple sources of interference, complete analysis involving simulation of all combinations of phasings and interference sources will not be computationally feasible. An alternative approach can then be to use a statistical sampling based method in which only a restricted set of phasings are simulated. That approach is further elaborated on in Section III-C.

III. Schedulability Analysis of CAN Messages

The Controller Area Network (CAN) is a broadcast bus designed to operate at speeds of up to 1 Mbps. Data is transmitted in messages containing between 0 and 8 bytes of data. An 11 bit identifier is associated with each message. The identifier is required to be unique, in the sense that two simultaneously active messages originating from different sources must have distinct identifiers. The identifier serves two purposes: (1) assigning a priority to the message, and (2) enabling receivers to filter messages.

CAN is a collision-detect broadcast bus, which uses deterministic collision resolution to control access to the bus. The basis for the access mechanism is the electrical characteristics of a CAN bus: if multiple stations are transmitting concurrently and one station transmits a '0' then all stations monitoring the bus will see a '0'. Conversely, only if all stations transmit a '1' will all processors monitoring the bus see a '1'. During arbitration, competing stations are simultaneously putting their identifiers, one bit at the time, on the bus. By monitoring the resulting bus value, a station detects if there is a competing higher priority message and stops transmission if this is the case. Because identifiers are unique within the system, a station transmitting the last bit of the identifier without detecting a higher priority message must be transmitting the highest priority queued message, and hence can start transmitting the body of the message.

A. Classical CAN Bus Analysis

Tindell et al. [9] [10] present analysis to calculate the worst-case latencies of CAN messages. This analysis is based on the standard fixed priority response time analysis for CPU scheduling [3].

Calculating the response times requires a bounded worst case queuing pattern of messages. The standard way of expressing this is to assume a set of traffic streams $\mathcal{S}$ (corresponding to CPU tasks), each generating messages with a fixed priority. The worst case behaviour of each stream is to periodically queue messages. Each $S_i \in \mathcal{S}$ is a triple $< P_i, T_i, C_i >$, where P_i is the priority (defined by the message identifier), T_i is the period and C_i the worst case transmission time of messages sent on stream S_i. The worst-case latency R_i of a CAN message sent on stream S_i is defined by

$$R_i = J_i + q_i + C_i \quad (15)$$

where J_i is the queuing jitter of the message, i.e., the maximum variation in queuing time relative T_i, inherited from

the sender task which queues the message, and q_i represents the effective queuing time, given by:

$$q_i = B_i + \sum_{j \in HP(i)} \left\lceil \frac{q_i + J_j + \tau_{bit}}{T_j} \right\rceil C_j + E(q_i + C_i) \quad (16)$$

where the term B_i is the worst-case blocking time of messages sent on S_i, $HP(i)$ is the set of streams with priority higher than S_i, τ_{bit} (the bit-time) caters for the difference in arbitration start times at the different nodes due to propagation delays and protocol tolerances, and $E(q_i+C_i)$ is an error term denoting the time required for error signalling and recovery. The reason for the blocking factor is that transmissions are non-preemptive, i.e., after a bus arbitration has started the message with the highest priority among competing messages will be transmitted till completion, even if a message with higher priority gets queued before the transmission is completed. However, in case of errors a message can be interrupted/preempted during transmission, requiring a complete retransmission of the entire message. The extra cost for this is catered for in the error term E above.

B. Our Previous Generalization

In [6] we present a generalization of the relatively simplistic error model by Tindell and Burns [9]. Our error model specifically addresses,

- multiple sources of errors: Handling of each source separately is not sufficient; instead they have to be composed into a worst case interference with respect to the latency on the bus.
- the signalling pattern of individual sources: Each source can typically be characterized by a pattern of shorter or longer bursts, during which the bus is unavailable, i.e., no signalling will be possible on the bus.

The above model is, just as Tindell and Burns' model, deterministic in that it models specific fixed patterns of interferences. An alternative is to use a stochastic model with interference distributions. Such a model is proposed by Navet et al. [11], which use a Generalized Poisson Process to model the frequency of interference, as well as their duration (single errors and error bursts).

In this article, we will use the following deterministic error model, which is a simplified version of the model introduced in [6]:

- There is a set $\mathcal{K}$ of sources of interference, with each source $k_i \in \mathcal{K}$ contributing an error term $E_{k_i}(t)$. Their combined effect $E(t)$ can be defined as,

$$E(t) = E_{k_1}(t) | E_{k_2}(t) | \ldots | E_{k_{|\mathcal{K}|}}(t) \quad (17)$$

where | denotes composition of error terms.

- Each source $k_i \in \mathcal{K}$ interferes by inducing an undefined bus value during a characteristic time period T_{k_i}. Each such interference will (if it coincides with a transmission) lead to a transmission error. If T_{k_i} is larger than τ_{bit}, then the error recovery will be delayed accordingly.
- Patterns of interferences for each source $k_i \in \mathcal{K}$ can independently be specified as a sequence of n_{k_i} bursts of length l_{k_i} with period T_{k_i}.

From the generic parameters it is possible to model both intermittent and transient sources of interference, as introduced in Section II-D. An intermittent source k is defined by letting $n_k > \frac{\mathcal{T}}{T_k}$. Any interference source with smaller n_k is a transient source. See figures 3 and 4 for illustrations.

Using such a model we can see that

$$E_{k_i}(t) = B_{k_i}(t) \times (O + \max(0, l_{k_i} - \tau_{bit})) \quad (18)$$

where the number of interferences until t, $B_{k_i}(t)$, is given by

$$B_{k_i}(t) = \min\left(n_{k_i}, \left\lceil \frac{t}{T_{k_i}} \right\rceil\right) \quad (19)$$

Note that, $\max(0, l_{k_i} - \tau_{bit})$ defines the length of l_{k_i} exceeding τ_{bit}, whereas $\left\lceil \frac{t}{T_{k_i}} \right\rceil$ is the number of initiated bursts until t.

We assume that the overhead O_i is given by:

$$O_i = 31 \times \tau_{bit} + \max_{l \in HP(i) \cup \{i\}} (C_l) \quad (20)$$

where $31 \times \tau_{bit}$ is the time required for error signalling in CAN and the *max*-term denotes the worst-case retransmission time as the largest transmission time of any message of higher priority ($HP(i)$) or the considered message (i). Retransmissions of corrupted messages with lower priority will not interfere, since the considered message will, due to the priority based arbitration, be transmitted before any such message.

C. Analysis with Random Phasings of Interferences

The analysis in Section III-B above assumes worst-case phasings of queuings and interferences. In combining timing and reliability modeling, we will use a relaxed model which considers the probability of different interference scenarios, not only the extreme worst-case. Our relaxed model will be based on

1. Worst-case phasings of message queuings at time 0 in the LCM (actually this could be at any time, so why not choose 0?). This introduces some pessimism, since the worst case may not occur in every LCM, but is consistent with the assumed traffic in the interference free model.
2. Random phasings of interferences. This can be expressed as an offset from the beginning of the mission time to when the first interference hits. For each interference source I_i that hits, such an offset should be "sampled" (as outlined in Section II-E).

To calculate the probability of a deadline miss we perform a simulation of the message transfer and interference during the mission time $\mathcal{T}$, as described in Section II-E.

The analysis we make is based on either exhaustive simulation or by sampling. If the LCM is small and the number of combined interference sources to be analysed are few, then exhaustive simulation is recommended. Algorithms for both exhaustive and sampling based simulation are presented in Appendix A.

In calculating the final failure probability $Q_F^{\mathcal{T}}$ (Equation 15) we can use either of the algorithms in Appendix A

can be used to calculate $P_F^{\mathcal{T}}$. In general, the total number of required simulations in the exhaustive case is rather large, since there are $2^{|\mathcal{K}|}-1$ combinations of interference sources to consider, and for each combination A, there are $\prod_{a\in A}\begin{cases} T_a & \text{if } a \in \mathcal{I} \\ \mathcal{T} & \text{if } a \in \mathcal{J} \end{cases}$ phasings to consider. The actual situation is however not as bad as this may indicate, since the algorithms can be optimized for special cases (e.g., using Equation 12 instead of Equation 13) and using *Rnd_sim* rather than *Ex_sim*. Due to the regular pattern of many scenarios and that we immediately can conclude "communication failure" if a single failure is detected, the time to perform simulations can be substantially reduced as well. The details of this are however outside the scope of this article.

We have implemented the analysis using a simulator developed by Lindgren et al. [12].

IV. Example : A Distributed Braking System

We now present a case study of a simplified intelligent Antilock Braking System (ABS), where each separate brake is controlled by a computer. Furthermore, there is one computer that controls the brake pedal. All nodes are connected by a CAN-bus (see Figure 5). The application is a distributed control algorithm, which calculates the brake force for each wheel depending on the brake pressure achieved from the driver. Therefore, each wheel-computer has to receive information about the state of the other wheels, to be able to make correct calculation and actuation. Thus each wheel is equipped with a sensor that monitors the rotation of the wheel. Each node sends the monitored values periodically.

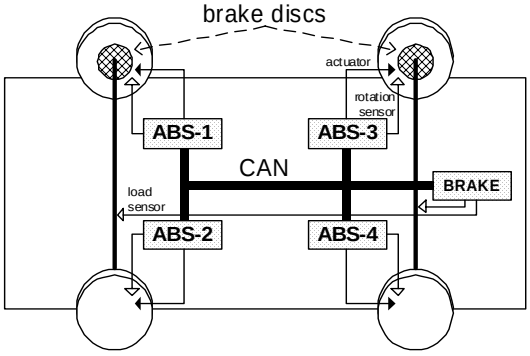


Fig. 5. Typical Computer Network in a Car with ABS

Our task is to implement a 'reliable' antilock braking system for vehicles. Since this is a subsystem of the entire vehicle system, we assume an appropriate reliability figure (say less than 10^{-9} faults/hour) to be attained by the ABS. This figure is in fact mandated by an assumed overall system reliability requirement of less than 10^{-7} faults/hour.

Table II specifies a typical subset of messages sent in this simplified ABS and the timing details (in ms) of these messages sent via CAN in a car. The timing parameters are typically requirements derived from the vehicle dynamics by a control engineer. Priority 1 is assumed to be the highest and 6 is the lowest. We assume a maximum blocking time of $135\tau_{bit}$ due to background message traffic. We also assume that the CAN bus operates at 250 Kbps.

Msg ID	Priority	T_i	D_i	C_i	Sender
OPERATOR-1	1	8	8	0.54	BRAKE
ABS-1	2	4	4	0.54	ABS-1
ABS-2	3	4	4	0.54	ABS-2
ABS-3	4	4	4	0.54	ABS-3
ABS-4	5	4	4	0.54	ABS-4
OPERATOR-2	6	15	15	0.54	BRAKE

TABLE II
A Typical Message Set in Car

Please note that the task set above is a quite simplified one from those used in practice. Our aim here is, however, not to present an accurate model of an ABS-system, but to illustrate our methodology and indicate its usefulness.

A. Interference Characteristics

In our example, we will consider two sources of interference, viz., a mobile phone lying inside the vehicle and radar transmissions from ships while the vehicle crosses bridges. These are considered to represent typical sources of interference. We characterize them in the following manner.

Mobile phones (such as GSM-phones) typically operate at 900MHz - 1800 MHz frequencies. The carrier when a call is active or being activated is for a period of about 500 μs duration out of a 4ms cycle (since each frequency can be shared by up to 8 phones). When inactive, the mobile phone will send signals to the base station once in every half-an-hour. In addition, on a moving vehicle extra signals are sent when the phone switches between base-stations. It should be noted that these interferences may have impact only when the mobile phone is lying close to the network cables. We assume a typical interval between bursts to be 30 secs, i.e. $T_{phone} = 30s$ and $l_{phone} = 500\mu s$.

For our second interference source, viz., radar transmissions from a ship, we assume the duration of an interference burst to be 1 ms with a single burst in each interference, i.e., $l_{radar} = 1ms$ and $n_{radar} = 1$.

B. Experiments

To illustrate our method we will provide the following analysis of our simple ABS system:

- Classical worst-case analysis, without considering any faults, using the analysis of Section III-A.
- Worst-case analysis under worst-case fault phasings, using the analysis of Section III-B.
- Worst-case analysis under arbitrary fault phasings, using the analysis of Section III-C. Since we are considering two independent sources of interferences, three cases will be considered: a) interference from the mobile phone only, b) interference from the radar only, and c) interference from both the sources.

In combining the obtained results to an overall reliability estimate for the considered mission time of eight hours, we will make the following assumptions:

- If a mobile phone is lying too close to a network cable, it will remain there for the entire mission.
- The probability that a mobile phone is lying too close to a network cable is 10^{-4}.
- The probability of passing a bridge (under which a ship equipped with a powerful radar may pass) is for each mission 0.5.
- The probability that a ship equipped with a powerful radar is passing under a bridge when a specific car (which is known to pass such a bridge during its current mission) is passing is 10^{-3}.
- The probability of a ship having activated its radar while passing a bridge is 0.7.

From the above information we derive: $P^{act}_{phone} = 10^{-4}$ and $P^{act}_{radar} = 0.5 \times 10^{-3} \times 0.7 = 3.5 \times 10^{-4}$. Note that we have no strong basis for the assumed values above. They merely represent intelligent guesses indicating the type of parameters that need to be identified.

C. Reliability Analysis Results

We have performed reliability analysis on our example message set. First, we calculated the response time without interference using Equation 15 in Section III-A. The resulting values (in milliseconds) in column (0) in Table III show that the message set is schedulable under interference free conditions.

Next we analysed the system under worst case interference phasings using the analysis from Section III-B. The results, presented in columns (1)-(3) in Table III, show that some messages miss deadlines (indicated by a '*') in all three cases: interferences from phone only (column (1)), radar only (column (2)), and both sources (column (3)).

According to the analysis under worst case assumptions, we can see that 1 out of 6 and 2 out of 6 messages miss their deadlines considering individual interference from phone only and radar only, respectively, whilst 5 out of 6 messages miss their deadlines under combined interferences from both the sources.

We finally conducted simulation runs by varying the points of start of interferences according to Algorithm 2 in Appendix A. The results are shown in Table IV. Combining the obtained failure probabilities, using the probabilities for the different interference scenarios derived from the assumptions in the previous section, we get the following failure probability formula (derived from Equation 15):

$$\begin{aligned} Q^{\mathcal{T}}_F(\{phone, radar\}) = {} & P^{act}_{phone}(1 - P^{act}_{radar}) \times 0.000968 + \\ & + P^{act}_{radar}(1 - P^{act}_{phone}) \times 0.001256 + \\ & + P^{act}_{phone} \times P^{act}_{radar} \times 0.002722 \end{aligned} \quad (21)$$

which evaluates to 5.36×10^{-7}.

Unfortunately this is an unacceptable high failure probability compared to the admissible failure probability of 10^{-9}. However, returning to the discussion on failure semantics at the end of Section II-C, we note that systems that fail due to a single deadline miss should be avoided, since they are extremely sensitive. Especially for critical systems, the designer has an obligation to make the system more robust. For instance, for our simple system a more reasonable failure semantics would be: "a failure occurs if more than 2 out of 10 deadlines are missed". It should be noted that changing failure semantics may have implications for the design, since we have to make sure that the system appropriately can handle the new situation, e.g., in our case that a few deadline misses can be tolerated.

Table V reports the results from a simulation of exactly the same system as above, except that we now use the new failure semantics.

Recalculating the overall mission failure probability $Q^{\mathcal{T}}_F$ using the failure probabilities from Table V gives $Q^{\mathcal{T}}_F(\{phone, radar\}) = 7.84 \times 10^{-12}$, which is quite negligible in relation to the required failure probability of 10^{-9}. This means that the system has sufficiently high reliability to be acceptable. Hence, our analysis has relieved the designer from the costly redesign process which would have been chosen if only the analysis with worst case phasing was considered.

V. Conclusions

We have presented a method that allows controlled relaxation of the timing requirements of safety-critical hard real-time systems. The underlying rational is that no real system is (or can ever be) hard real-time, since the behaviour of neither the design nor the hardware components can be completely guaranteed. By integrating hard real-time schedulability with the reliability analysis normally used to estimate the imperfection of reality, we obtain a more accurate reliability analysis framework with high potential for providing solid arguments for making design tradeoffs, e.g., that allow a designer to choose a slower (and less expensive) bus or CPU, even though the timing requirements are violated in some rare worst-case scenario.

Using traditional schedulability analysis techniques, the designer will in many cases have no other choice than to redesign the system (in hardware, software or both). However, by resorting to our new analysis, we may see that the probability of an extreme error situation arising is very low and thus the designer may not need to perform a costly re-design.

It is well known [8] that a control system that fails due to a single deadline miss is not robust enough to be of much practical use. Rather the system should tolerate single deadline misses, or even multiple deadline misses or more complex requirements on the acceptable pattern of deadline misses. These requirements should of course be derived from the requirements on stability in the control of the external process. The possibility to handle such requirements in the analysis can makes the use of the resources even more efficient, i.e., we achieve a tradeoff situation between algorithmic fault-tolerance and resource usage. By considering each message separately in our example we could increase the reliability by incorporating algorithmic fault tolerance for functions which are dependent on a message that has

Msg ID	Priority	T_i	D_i	C_i	Response Time			
					(0) no interf.	(1)phone	(2)radar	(3) phone&radar
OPERATOR-1	1	8	8	0.54	1.08	2.24	2.74	3.549
ABS-1	2	4	4	0.54	1.64	2.78	3.28	* 4.374
ABS-2	3	4	4	0.54	2.16	3.32	3.82	* 6.131
ABS-3	4	4	4	0.54	2.70	3.86	* 4.36	* 7.339
ABS-4	5	4	4	0.54	3.24	* 4.40	* 6.52	* 8.419
OPERATOR-2	6	15	15	0.54	3.78	7.86	7.60	*16.384

TABLE III

Response Time Analysis- Normal and under Faults

Interference Sources	Average Case in Simulation			
	Total Messages	Missed Deadlines	Failure Probability	99.9% Confidence Interval
phone only	4530000	4385	96.8×10^{-5}	$\pm$ 4.2×10^{-5}
radar only	4530000	5691	125.6×10^{-5}	$\pm$ 5.3×10^{-5}
phone and radar	4530000	12331	272.2×10^{-5}	$\pm$ 8.3×10^{-5}

TABLE IV

Simulation results: Simple failure semantics

Interference Sources	Average Case in Simulation			
	Total Messages	Number of Failures	Failure Probability	99.9% Confidence Interval
phone only	4530000	0	0	0
radar only	4530000	0	0	0
phone and radar	4530000	1015	22.4×10^{-5}	$\pm$ 6.6×10^{-5}

TABLE V

Simulation results: Refined failure semantics

the lowest reliability.

The presented method is tailored for analysis of the effects of external interference on CAN-bus communication. The method could be extended in various directions, such as including stochastic modeling of external interferences, distributions of transmission times due to bit-stuffing, distributions of actual queuing times, distributions of queuing jitter, as well as applying the framework to CPU scheduling, including variations in execution times of tasks, jitter, periods for sporadic tasks, etc. Some of these extensions require dependency issues to be carefully considered. For instance, message queuing jitter may for all messages on the same node be dependent on interrupt frequencies. Assuming independence in such a situation may lead to highly inaccurate results. Another critical issue which should be given further attention is the sensitivity of the analysis to variations in model parameters and assumptions, such as the assumed probabilities for the interference sources to be active in our example.

We are convinced that a successful development of a holistic analysis framework, taking both reliability and schedulability (as well as other pertinent issues) into account will be of immense value for the development of resource constrained safety-critical real-time systems. The method presented here is an important step in that direction.

VI. Acknowledgement

The authors wish to express their gratitude to Ralf Elvsén for useful discussions and to the anonymous reviewers for their helpful comments. The work presented in this paper was supported by the Swedish Foundation for Strategic Research (SSF) via ARTES, the Swedish Foundation for Knowledge and Competence Development (KK-stiftelsen), and Mälardalen University.

References

[1] H. Hansson, C. Norström, and S. Punnekkat, "Integrating Reliability and Timing Analysis of CAN-based Systems," in *Proc. 2000 IEEE International Workshop on Factory Communication Systems (WFCS'2000)*, Porto, Portugal, September 2000, IEEE Industrial Electronics Society.

[2] H. Hansson, C. Norström, and S. Punnekkat, "Reliability Modelling of Time-Critical Distributed Systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, M. Joseph, Ed., 6th International Symposium, FTRTFT 2000, Pune, India, September 2000, vol. 1926 of *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag.

[3] N. C. Audsley, A. Burns, M.F. Richardson, K. Tindell, and A.J. Wellings, "Applying New Scheduling Theory to Static Priority

Pre-emptive Scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, September 1993.

[4] L. Sha, R. Rajkumar, and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[5] J. Xu and D. L. Parnas, "Priority scheduling versus pre-run-time scheduling," *Real-Time Systems Journal*, vol. 18, no. 1, pp. 7–23, January 2000.

[6] S. Punnekkat, H. Hansson, and C. Norström, "Response Time Analysis under Errors for CAN," in *Proceedings of IEEE Real-Time Technology and Applications Symposium (RTAS 2000)*. June 2000, pp. 258–265, IEEE Computer Society.

[7] L. M. Pinho, F. Vasques, and E. Tovar, "Integrating inaccessibility in response time analysis of can networks," in *Proc. 2000 IEEE International Workshop on Factory Communication Systems (WFCS'2000)*, Porto, Portugal, September 2000, pp. 77–84, IEEE Industrial Electronics Society.

[8] M. Törngren, "Fundamentals of implementing real-time control applications in distributed computer systems," *Real-Time Systems Journal*, vol. 14, no. 3, pp. 219–250, May 1998.

[9] K. W. Tindell and A. Burns, "Guaranteed message latencies for distributed safety-critical hard real-time control networks," Tech. Rep. YCS229, Dept. of Computer Science, University of York, June 1994.

[10] K. W. Tindell, H. Hansson, and A. J. Wellings, "Analysing Real-Time Communications: Controller Area Network (CAN)," in *Proceedings 15th IEEE Real-Time Systems Symposium*. December 1994, pp. 259–265, IEEE Computer Society.

[11] N. Navet, Y.-Q. Song, and F. Simonot, "Worst-case deadline failure probability in real-time applications distributed over controller area network," *Journal of Systems Architecture*, vol. 7, no. 46, pp. 607–617, September 2000.

[12] M. Lindgren, H. Hansson, C. Norström, and S. Punnekkat, "Deriving reliability estimates of distributed real-time systems," in *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, Cheju Island, South Korea, December 2000, IEEE Computer Society.

Appendix

I. Simulation algorithms

The following are high level pseudo code descriptions of the algorithms for the simulation based analysis in Section III-C.

Algorithm 1 (Exhaustive Simulation)
The algorithm takes as input a list of interference sources A and returns the probability that an interference scenario causes a communication failure.

```
Algorithm Ex_sim(A : "list of interference sources"):
probability
sample : array ["A"] of int;

Function startsample(k: interference source): Int
    If k ∈ 𝓘 {intermittent source}
    then startsample:= 0
    else {transient source}
        startsample:= −n_k T_k;
end.func

Function maxsample(k: interference source): Int
    If k ∈ 𝓘 {intermittent source}
    then maxsample:= T_k - 1
    else {transient source}
        maxsample:= 𝒯 -1;
end.func

Function simulate("list of phasings"): 1/0
    "Simulate behavior during 𝒯"
    If deadline_violation
    then simulate:=1
    else simulate:=0;
end.func
begin.alg
    scenarios:=0;
    fail:=0;
    For sample[First(A)]:=startsample(First(A)) to
            maxsample(First(A)) do
        For sample[next(First(A))]:=
                startsample(next(First(A))) to
                maxsample(next(First(A))) do
        ⋮
            For sample[Last(A)]:=
                startsample(Last(A)) to
                maxsample(Last(A)) do
                    scenarios:= scenarios + 1;
                    fail:= fail + simulate(sample);
            od
        ⋮
        od
    od
    Ex_sim:= fail/scenarios;
end.alg
```

The algorithm *Ex_sim* investigates for each combination of phasings if the interference sources cause a communication failure or not.

Algorithm 2 (Random Simulation)
The algorithm takes as input a list of interference sources A and returns the probability that an interference scenario causes a communication failure.

```
Algorithm Rnd_sim(A : "list of interference sources"):
probability
sample : array ["A"] of int;
begin.alg
    scenarios:=0;
    fail:=0;
    Do until confidence ≥ required_confidence
        For a = First(A) to Last(A) do
            sample[a]:=
                pick_a_sample(startsample(a),
                maxsample(a))
        od
        scenarios:= scenarios + 1;
        fail:= fail + simulate(sample);
    od
    Rnd_sim:= fail/scenarios;
end.alg
```

The algorithm *Rnd_sim* randomly selects phasings of the interference sources and investigates if the selected combination of interferences causes a communication failure. The procedure is repeated until required level of confidence is reached.

Real-Time Systems, , 1–21 ()

Compact Data Structures and State-Space Reduction for Model-Checking Real-Time Systems

KIM G. LARSEN kgl@cs.auc.dk
BRICS, Aalborg University, Denmark.

FREDRIK LARSSON, fredrikl@docs.uu.se
Department of Information Technology, Uppsala University, Sweden.

PAUL PETTERSSON paupet@docs.uu.se
Department of Information Technology, Uppsala University, Sweden.

WANG YI yi@docs.uu.se
Department of Information Technology, Uppsala University, Sweden.

Abstract. During the past few years, a number of verification tools have been developed for real-time systems in the framework of timed automata. One of the major problems in applying these tools to industrial-sized systems is the huge memory-usage for the exploration of the state-space of a network (or product) of timed automata, as the model-checkers must keep information about not only the control structure of the automata but also the clock values specified by clock constraints.

In this paper, we present a compact data structure for representing clock constraints. The data structure is based on an $\mathcal{O}(n^3)$ algorithm which, given a constraint system over real-valued variables consisting of bounds on differences, constructs an equivalent system with a *minimal* number of constraints. In addition, we have developed an on-the-fly reduction technique to minimize the space-usage. Based on static analysis of the control structure of a network of timed automata, we are able to compute a set of symbolic states that cover all the dynamic loops of the network in an on-the-fly searching algorithm, and thus ensure termination in reachability analysis.

The two techniques and their combination have been implemented in the tool UPPAAL. Our experimental results demonstrate that the techniques result in truly significant space-reductions: for six examples from the literature, the space saving is between 75% and 94%, and in (nearly) all examples time-performance is improved. Noteworthy is also the observation that the two techniques are completely orthogonal.

Keywords: Real-Time Systems, Model Checking, Design Tool, Formal Specification and Verification, Timed Automata

1. Introduction

Reachability analysis has been one of the most successful methods for automated analysis of concurrent systems. Many verification problems e.g. invariant checking can be solved by means of reachability analysis. It can in many cases also be used for checking whether a system described as an automaton satisfies a requirement specification formulated e.g. in linear temporal logic, by converting the requirement to an automaton and thereafter checking whether the parallel composition of the system and requirement automata can reach certain annotated states [32, 21, 2]. However, the major problem in applying reachability analysis is the potential combinatorial

```
PASSED:= {}
WAITING:= {(l_0, D_0)}
repeat
    begin
        get (l, D) from WAITING
        if (l, D) ⊨ φ then return "YES"
        else if D ⊈ D' for all (l, D') ∈ PASSED then
            begin
                add (l, D) to PASSED
                SUCC:={(l_s, D_s) | (l, D) ⇝ (l_s, D_s) and D_s ≠ ∅}
                for all (l_s, D_s) in SUCC do
                    put (l_s, D_s) to WAITING
            end
    end
until WAITING={}
return "NO"
```

Figure 1. An Algorithm for Symbolic Reachability Analysis.

explosion of state spaces. To attack this problem, various symbolic and reduction techniques have been put forward over the last decade to efficiently represent state space and to avoid exhaustive state space exploration (e.g. [11, 17, 31, 12, 13, 16, 5]); such techniques have played a crucial role for the successful development of verification tools for finite-state systems.

In the last few years, new verification tools have been developed, for the class of infinite-state systems known as timed systems [19, 14, 9]. Notably the verification engines of most tools in this category are based on reachability analysis of timed automata following the pioneering work of Alur and Dill [4]. A timed automaton is an extension of a finite automaton with a finite set of real-valued clock-variables. The foundation for decidability of reachability problems for timed automata is Alur and Dill's region technique, by which the infinite state space of a timed automaton due to the density of time, may effectively be partitioned into finitely many equivalence classes i.e. *regions* in such a way that states within each class will always evolve to states within the same classes. However, reachability analysis based on the region technique is practically infeasible due to the potential state explosions arising from not only the control-structure (as for finite-state systems) but also the region space [23].

Efficient data structures and algorithms have been sought to represent and manipulate timing constraints over clock variables (e.g. by Difference Bounded Matrices [7, 15], or Binary Decision Diagrams [11, 6]) and to avoid exhaustive state space

exploration (e.g. by application of partial order reductions [17, 31, 26] or compositional methods [5, 23]). One of the main achievements in these studies is the symbolic technique [15, 33, 20, 34, 23], that converts the reachability problem to that of solving simple constraints. The technique can be simply formulated in an abstract reachability algorithm[1] as shown in Figure 1. The algorithm is to check whether a timed automaton may reach a state satisfying a given state formula φ. It explores the state space of the automaton in terms of *symbolic states* in the form (l, D) where l is a control-node and D is a constraint over clocks variables.

We observe that several operations of the algorithm are critical for efficient implementations. Firstly, the algorithm depends heavily on the test operations for checking the inclusion $D \subseteq D'$ (i.e. the inclusion between the solution sets of D, D') and the emptiness of D_s in constructing the successor set SUCC of (l, D). Clearly, it is important to design efficient data structures and algorithms for the representation and manipulation of clock constraints. One such well-known data structure is that of DBM (*Difference Bounded Matrix*), which offers a canonical representation for constraint systems. It has been successfully used in several real-time verification tools, e.g. UPPAAL [9] and KRONOS [14]. A DBM representation is in fact a weighted directed graph where the vertices correspond to clocks (including a zero-clock) and the weights on the edges stand for the bounds on the differences between pairs of clocks [7, 15, 33]. As it gives an explicit bound for the difference between each pair of clocks, its space-usage is in the order of $\mathcal{O}(n^2)$ where n is the number of clocks. However, in practice it often turns out that most of these bounds are redundant.

In this paper, we present a compact data structure for DBM, which provides *minimal* and *canonical* representations of clock constraints and also allows for efficient inclusion checks. We have developed an $\mathcal{O}(n^3)$ algorithm that given a DBM constructs a minimal number of constraints equivalent to the original constraints represented by the DBM (i.e. with the same solution set). The algorithm is essentially a minimization algorithm for weighted directed graphs, and hence solves a problem of independent interest. Note that the main global data structure of the algorithm in Figure 1 is the passed list (i.e. PASSED) holding the explored states. In many cases, it will store all the reachable symbolic states of the automaton. Thus, it is desirable that when saving a (symbolic) state in the passed list, we save the (often substantially smaller) minimal constraint system. The minimal representation also makes the inclusion-checking of the algorithm more efficient. Our experimental results demonstrate truly significant space-savings as well as better time-performance (see statistics in section 5).

In addition to the *local* reduction technique above, which is to minimize the space-usage of each individual symbolic state, as the second contribution of this paper, we have developed a *global* reduction technique to reduce the total number of states to save in the global data structure, i.e. the passed list. It is completely orthogonal to the local technique. In the abstract algorithm of Figure 1, we notice the step of saving the new encountered state (l, D) in the passed list when the inclusion-checking for $D \subseteq D'$ fails (i.e. $D \not\subseteq D'$). Its purpose is first of all to guarantee termination but also to avoid repeated exploration of states that have several predecessors. However, this is not necessary if all the predecessors of (l, D) are already

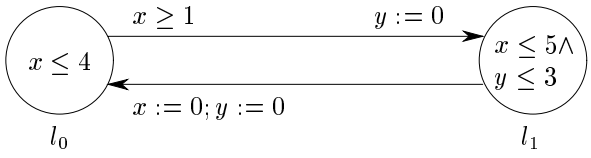


Figure 2. A Timed Automaton.

present in the passed list. In fact, to ensure termination, it suffices to save only one state for each dynamic loop. An improved on-the-fly reachability algorithm according to the global reduction strategy has been implemented in UPPAAL [9] based on static analysis of the control structure of timed automata. Our experimental results demonstrate significant space-savings and also better time-performance (see statistics in section 5).

The outline of this paper is as follows: In the next section we review the semantics of timed automata and the notion of Difference Bounded Matrix (DBM) for clock constraints. Section 3 presents the compact data structure for DBM and the local reduction technique (i.e. the minimization algorithm for weighted directed graphs). Section 4 is devoted to develop the global reduction technique based on control structure analysis. Section 5 presents our experimental results for both techniques and their combination. Section 6 concludes the paper.

2. Preliminaries

2.1. Timed Automata

The model of timed automata was first introduced in [4] and has since then established itself as a standard model for real-time systems. For the reader not familiar with the notion of timed automata we give a short informal description.

Consider the timed automaton of Figure 2. It has two control nodes l_0 and l_1 and two real-valued clocks x and y. A *state* of the automaton is of the form (l, s, t), where l is a control node, and s and t are non-negative reals giving the value of the two clocks x and y. A control node is labelled with a condition (the invariant) on the clock values that must be satisfied for states involving this node. Assuming that the automaton starts to operate in the state $(l_0, 0, 0)$, it may stay in node l_0 as long as the invariant $x \leq 4$ of l_0 is satisfied. During this time the values of the clocks increase synchronously. Thus from the initial state, all states of the form (l_0, t, t), where $t \leq 4$, are reachable. The edges of a timed automaton may be decorated with a condition (guard) on the clock values that must be satisfied in order to be enabled. Thus, only for the states (l_0, t, t), where $1 \leq t \leq 4$, is the edge from l_0 to l_1 enabled. Additionally, edges may be labelled with simple assignments reseting clocks. For example, when following the edge from l_0 to l_1 the clock y is reset to 0 leading to states of the form $(l_1, t, 0)$, where $1 \leq t \leq 4$.

In general, a timed automaton is a standard finite-state automaton extended with a finite collection $\mathcal{C}$ of real-valued clocks ranged over by x, y etc. We use $\mathcal{B}(\mathcal{C})$ ranged over by g (and latter D), to stand for the set of formulas that can be an atomic constraint of the form: $x \sim n$ or $x - y \sim n$ for $x, y \in \mathcal{C}$, $\sim \in \{\leq, \geq\}^2$ and n being a natural number, or a conjunction of such formulas. Elements of $\mathcal{B}(\mathcal{C})$ are called *clock constraints* or *constraint systems* over $\mathcal{C}$.

Definition 1. [Timed Automata] A timed automaton A over clocks $\mathcal{C}$ is a tuple $\langle N, l_0, \longrightarrow, I\rangle$ where N is a finite set of nodes (control-nodes), l_0 is the initial node, $\longrightarrow \subseteq N \times \mathcal{B}(\mathcal{C}) \times 2^{\mathcal{C}} \times N$ corresponds to the set of edges, and finally, $I : N \mapsto \mathcal{B}(\mathcal{C})$ assigns invariants to nodes. In the case, $\langle l, g, r, l'\rangle \in \longrightarrow$, we write $l \xrightarrow{g,r} l'$. □

Formally, we represent the values of clocks as functions (called clock assignments) from $\mathcal{C}$ to the non-negative reals $\mathbb{R}_+$. We denote by $\mathbb{R}_+^{\mathcal{C}}$ the set of clock assignments for $\mathcal{C}$. A semantical *state* of an automaton A is now a pair (l, u), where l is a node of A and u is a clock assignment for $\mathcal{C}$, and the semantics of A is given by a transition system with the following two types of transitions (corresponding to delay-transitions and edge-transitions):

- $(l, u) \longrightarrow (l, u \oplus d)$ if $I(l)(u)$ and $I(l)(u \oplus d)$
- $(l, u) \longrightarrow (l', u')$ if there exist g and r such that $l \xrightarrow{g,r} l'$, $g(u)$ and $u' = r[u]$

where for $d \in \mathbb{R}_+$, $u \oplus d$ denotes the time assignment which maps each clock x in $\mathcal{C}$ to the value $u(x) + d$, and for $r \subseteq \mathcal{C}$, $r[u]$ denotes the assignment for $\mathcal{C}$ which maps each clock in r to the value 0 and agrees with u over $\mathcal{C} \backslash r$.

Clearly, the semantics of a timed automaton yields an infinite transition system, and is thus not an appropriate basis for decision algorithms. However, efficient algorithms may be obtained using a finite-state *symbolic* semantics based on *symbolic states* of the form (l, D), where $D \in \mathcal{B}(\mathcal{C})$ [20, 34]. We shall consider a clock constraint as a set of clock assignments and use $u \in D$ to stand for u satisfied D.

The symbolic counterpart to the standard semantics is given by the following two (fairly obvious) types of symbolic transitions:

- $(l, D) \rightsquigarrow \Big(l, (D \wedge I(l))^{\uparrow} \wedge I(l)\Big)$
- $(l, D) \rightsquigarrow \Big(l', r(g \wedge D)\Big)$ if $l \xrightarrow{g,r} l'$

where $D^{\uparrow} = \{u \oplus d \mid u \in D \wedge d \in \mathbb{R}_+\}$ and $r(D) = \{r[u] \mid u \in D\}$. It may be shown that $\mathcal{B}(\mathcal{C})$ (the set of clock constraints) is closed under these two operations (and $\wedge$) [15]. Moreover, the symbolic semantics characterize the standard semantics in the sense that, whenever $u \in D$ and $(l, D) \rightsquigarrow (l', D')$ then $(l, u) \longrightarrow (l', u')$ for $u' \in D'$.

Finally, we introduce the notion of networks of timed automata [34, 23]. A network is the parallel composition of a finite set of automata for a given synchronization function. To illustrate the on-the-fly verification technique, we only need to study the case dealing with interleaving, that is, the network of automata $A_1 \dots A_n$, is the Cartesian product of A_i's. Assume a vector l of control nodes. We shall use $l[i]$ to stand for the ith element of l and $l[l'_i/l_i]$ for the vector where the ith element l_i of l is replaced by l'_i. A control node (i.e. *control vector*) l of a network $A_1 \dots A_n$ is a vector where $l[i]$ is a node of A_i and the invariant $I(l)$ of l is the conjunction of $I(l[1]) \dots I(l[n])$. The symbolic semantics of networks is given in terms of control vectors by the following two types of symbolic transitions:

- $(l, D) \rightsquigarrow \Big(l, (D \wedge I(l))^{\uparrow} \wedge I(l)\Big)$
- $(l, D) \rightsquigarrow \Big(l[l'_i/l_i], r(g \wedge D)\Big)$ if $l_i \xrightarrow{g,r} l'_i$

In the later case, we shall say that the symbolic transition is derived by the edge $l_i \xrightarrow{g,r} l'_i$.

2.2. Difference Bounded Matrices & Shortest-Path Closure

To utilize the symbolic semantics of (networks of) timed automata algorithmically, as for example in the reachability algorithm of Figure 1, it is important to design efficient data structures and algorithms for the representation and manipulation of clock constraints.

One such well-known data structure is that of difference bounded matrices (DBM, see [7, 15]), which offers a canonical representation for constraint systems. A DBM representation of a constraint system D is simply a weighted, directed graph, where the vertices correspond to the clocks of C and an additional zero-vertex 0. The graph has an edge from x to y with weight m provided $y - x \leq m$ is a constraint of D. Similarly, there is an edge from 0 to x with weight m, whenever $x \leq m$ is a constraint of D [3]. As an example, consider the constraint system E over $\{x_0, x_1, x_2, x_3\}$ being a conjunction of the atomic constraints $x_0 - x_1 \leq 3$, $x_3 - x_0 \leq 5$, $x_3 - x_1 \leq 2$, $x_2 - x_3 \leq 2$, $x_2 - x_1 \leq 10$, and $x_1 - x_2 \leq -4$. The graph representing E is given in Figure 3 (a).

In general, the same set of clock assignments may be described by several constraint systems (and hence graphs). To test for inclusion between constraint systems D and D' [4], which we recall is essential for the termination of the reachability algorithm of Figure 1, it is advantageous if D is *closed under entailment* in the sense that no constraint of D can be strengthened without reducing the solution set. In particular, for D a closed constraint system, $D \subseteq D'$ holds if and only if for any constraint in D' there is a constraint in D at least as tight; i.e. whenever $(x - y \leq m') \in D'$ then $(x - y \leq m) \in D$ for some $m \leq m'$. Thus, closedness provides a canonical representation, as two closed constraint systems describe the same solution set precisely when they are identical. To close a constraint system D

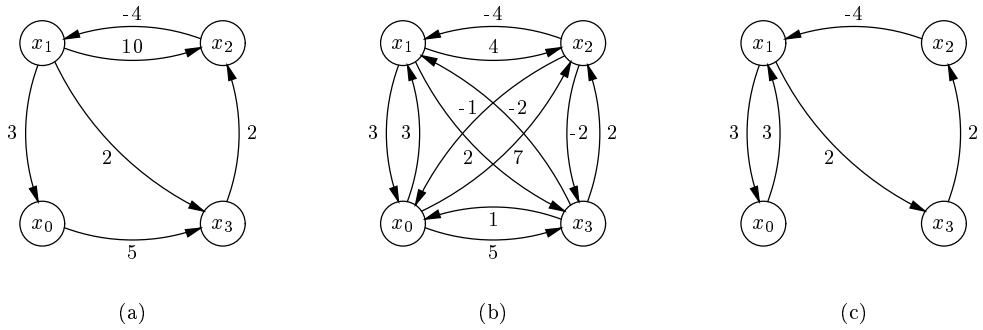


Figure 3. Graph for E (a), its shortest-path closure (b), and shortest-path reduction (c).

amounts to derive the shortest-path closure for its graph and can thus be computed in time $\mathcal{O}(n^3)$, where n is the number of clocks of D. The graph representation of the closure of the constraint system E from Figure 3 (a) is given in Figure 3 (b). The emptiness-check of a constraint system D simply amounts to checking for negative-weight cycles in its graph representation. Finally, given a closed constraint system D the operations $D^{\uparrow}$ and $r(D)$ may be performed in time $\mathcal{O}(n)$.

3. Minimal Constraint Systems & Shortest Path Reductions

For the reasons stated above a matrix representation of constraint systems in closed form is an attractive data structure, which has been successfully employed by a number of real-time verification tools, e.g. UPPAAL [9] and KRONOS [14]. As it gives an explicit (tightest) bound for the difference between each pair of clocks (and each individual clock), its space-usage is of the order $\mathcal{O}(n^2)$. However, in practice it often turns out that most of these bounds are redundant, and the reachability algorithm of Figure 1 is consequently hampered in two ways by this representation. Firstly, the main data structure PASSED, will in many cases store all the reachable symbolic states of the automaton. Thus, it is desirable, that when saving a symbolic state in the PASSED-list, we save a representation of the constraint system with as few constraints as possible. Secondly, a constraint system D added to the PASSED-list is subsequently only used in checking inclusions of the form $D' \subseteq D$. Recalling the method for inclusion-check from the previous section, we note that (given D' is closed) the time-complexity of the inclusion-check is linear in the number of constraints of D. Thus, again it is advantageous for D to have as few constraints as possible.

In the following subsections we shall present an $\mathcal{O}(n^3)$ algorithm, which given a constraint system constructs an equivalent reduced system with the minimal number of constraints. The reduced constraint system is canonical in the sense that two constrain systems with the same solution set give rise to identical reduced sys-

tems. The algorithm is essentially a minimization algorithm for weighted directed graphs. Given a weighted, directed graph with n vertices, it constructs in time $\mathcal{O}(n^3)$ a reduced graph with the minimal number of edges having the same shortest path closure as the original graph. Figure 3 (c) shows the minimal graph of the graphs in Figure 3 (a) and (b), which is computed by the algorithm.

3.1. Reduction of Zero-Cycle Free Graphs

A weighted, directed graph G is a structure (V, E_G), where V is a finite set of vertices and E_G, is a partial function from $V \times V$ to Z (the integers). The domain of E_G constitutes the edges of G, and when defined, $E_G(x, y)$ gives the weight of the edge between x and y. We assume that $E_G(x, x) = 0$ for all vertices x, and that G has no cycles with negative weight[5].

Given a graph G, we denote by G^C the *shortest-path closure* of G, i.e. $E_{G^C}(x, y)$ is the length of the shortest path from x to y in G. A *shortest-path reduction* of a graph G is a graph G^R with the minimal number of edges such that $(G^R)^C = G^C$.

The key to reduce a graph is obviously to remove *redundant edges*, where an edge (x, y) is redundant if there exist an alternative path from x to y whose (accumulated) weight does not exceed the weight of the edge itself. For example, in the graph of Figure 3 (a), the edge (x_1, x_2) is clearly redundant as the accumulated weight of path $(x_1, x_0), (x_0, x_3), (x_3, x_2)$ has a weight (10) not exceeding the weight of the edge itself (also 10). The path $(x_1, x_3), (x_3, x_2)$ makes also the edge (x_1, x_2) redundant. Being redundant, the edge (x_1, x_2) may be removed without changing the shortest-path closure. We shall use $G \setminus (x_1, x_2)$ to denote the result of removing the edge (x_1, x_2) from the graph G.

Now, consider the edge (x_1, x_2) in the graph of Figure 3 (b). Clearly, the edge is redundant as the path $(x_1, x_3), (x_3, x_2)$ has equal weight. Similarly, the edge (x_3, x_2) is redundant as the path $(x_3, x_1), (x_1, x_2)$ has equal weight. However, though redundant, we cannot just remove the two edges (x_1, x_2) and (x_3, x_2) as removal of one clearly requires the presence of the other. In fact, all edges between the vertices x_1, x_2 and x_3 are redundant, but obviously we cannot remove them all simultaneously. The key explanation of this complicating phenomena is that x_1, x_2, x_3 constitutes a cycle with length zero (a *zero-cycle*). However, for zero-cycle free graphs the situation is the simplest possible:

LEMMA 1 *Let G_1 and G_2 be zero-cycle free graphs such that ${G_1}^C = {G_2}^C$. If there is an edge $(x, y) \in G_1$ such that $(x, y) \notin G_2$, then $(G_1 \setminus \{(x, y)\})^C = {G_1}^C = {G_2}^C$.*

Proof: Let α denote the edge (x, y) and let m be the weight of α in G_1. We will show that there is an alternative path in G_1 *not* using α with weight no more than m. From this fact the Lemma obviously follows.

As ${G_1}^C = {G_2}^C$, the shortest path from x to y in G_2 has weight no more than m. As $\alpha \notin G_2$, this path must visit some vertex z different from x and y. Now let m_1 be the shortest path-weight from x to z and let m_2 be the shortest path-weight from z to y; note that G_1 and G_2 agrees on m_1 and m_2, as they have the same shortest-path closure. Then clearly, $m \geq m_1 + m_2$.

Now assume that the shortest path in G_1 from x to z uses $\alpha = (x, y)$. Then, as a sub-path, G_1 will be a path from y to z. Since G_1 also has a path from z to y, it follows that G_1 will have a cycle from y via z back to y. The weight of this cycle can be argued to be no more than $(m_1 - m) + m_2$. However, as $m \geq m_1 + m_2$ and there are no negative cycles, this cycle must have weight 0 contradicting the assumption that G_1 is zero-cycle free.

Similarly, a contradiction with the zero-cycle free assumption of G_1 is obtained, if the shortest path in G_1 from z to y uses α. thus we can conclude that there is an path from x to y not using α with length no greater than m. □

From the above Lemma it follows immediately that all redundant edges of a zero-cycle free graph may be removed without affecting the closure. On the other hand, removal of an edge which is not redundant will of course change the closure of the graph, and must be present in any graph with the same closure. Thus the following theorem follows:

THEOREM 1 *Let G be a zero-cycle free graph, and let $\{\alpha_1, \ldots, \alpha_2\}$ be the set of redundant edges of G. Then $G^R = G^C \setminus \{\alpha_1, \ldots, \alpha_k\}$.*

Proof: Follows from Lemma 1. □

From an algorithmic point of view, redundancy of edges is easily determined given the closure G^C of a graph G as only path of length 2 needs to be considered: An edge (x, y) is redundant precisely when there is a vertex z ($\neq x, y$) such that $E_{G^C}(x, y) \geq E_{G^C}(x, z) + E_{G^C}(z, y)$. Thus for zero-cycle free graphs computing G^R is $\mathcal{O}(n^3)$.

3.2. Reduction of Negative-Cycle Free Graphs

For general graphs (without negative cycles) our reduction construct relies on a partitioning of the vertices according to zero-cycles. We say that two vertices x and y are *equivalent* or *zero-equivalent*, if there is a zero-cycle containing them both. We write $x \equiv y$ in this case. Given the closure G^C of a graph G, it is extremely easy to check for zero-equivalence: $x \equiv y$ holds precisely when $E_{G^C}(x, y) = -E_{G^C}(y, x)$. Thus, in the graphs of Figure 3 (a) and (b), $\equiv$ partitions the vertices into the two classes $\{x_0\}$ and $\{x_1, x_2, x_3\}$.

To obtain a canonical reduction, we assume that the vertices of G are ordered by assigning them indices as $x_1, x_2, \ldots, x_n$. The equivalence $\equiv$ now induces a natural transformation $G_\equiv$ on the graph G:

Definition 2. Given a graph G, the vertices of the graph $G_\equiv$ are $\equiv$-equivalence classes, denoted E_k, of G. There is an edge between the classes E_i and E_j ($i \neq j$) if for some $x \in E_i$ and $y \in E_j$ there is an edge in G between x and y. The weight of this edge is $E_{G^C}(E_i^{min}, E_j^{min})$, where E^{min} is the vertex in E with the smallest index. □

Thus, the distance between E_i and E_j in $G_{\equiv}$ is the weight of the shortest path in G between the elements of E_i and E_j with smallest index. It is obvious that $G_{\equiv}$ is a zero-cycle free graph. It is also easy to see that $G_{1\equiv} = G_{2\equiv}$ if ${G_1}^C = {G_2}^C$. Let H be the graph of Figure 3 (a). Then $H_{\equiv}$ will have vertices $E_0 = \{x_0\}$ and $E_1 = \{x_1, x_2, x_3\}$. The two vertices are connected by two edges both having weight 3.

The following provides a dual to the operator of Definition 2:

Definition 3. Let F be a graph with vertices being $\equiv$-equivalence classes with respect to a graph $G = (V, E_G)$. Then the expansion of F is a graph F^+ with vertices V and with weight satisfying:

- For any multi-member equivalence class[6] $\{z_1 < z_2 < \cdots < z_k\}$ of F, F^+ contains a single cycle $z_1, z_2, \ldots, z_k, z_1$, with the weight of the edge (z_i, z_{i+1}) being the weight of the shortest path from z_i to z_{i+1} in G.
- Whenever (E_i, E_j) is an edge in F with weight m, then F^+ will have an edge from E_i^{min} to E_j^{min} with weight m. □

We are now ready to state the main Theorem giving the shortest-path reduction construct for arbitrary negative-cycle free graphs:

THEOREM 2 *Let G be negative-cycle free graph. Then the shortest-path reduction G^R of G is given by the graph $({G_{\equiv}}^R)^+$, i.e. $G^R = ({G_{\equiv}}^R)^+$.*

Proof: We show: (1) that $({G_{\equiv}}^R)^+$ is a candidate for a shortest-path reduction of G in the sense that $({G_{\equiv}}^R)^+ = G^C$, and (2) that $({G_{\equiv}}^R)^+$ is minimal.

1. We first prove that $({G_{\equiv}}^R)^+ = G^C$. As all edges (x, y) of $({G_{\equiv}}^R)^+$ have weight of the form $E_{G^C}(x, y)$, it follows that for any path in $({G_{\equiv}}^R)^+$ there is a path in G with same weight.

   Now consider an edge (x, y) of G. We will demonstrate that there is a path in $({G_{\equiv}}^R)^+$ with no greater weight.

   - If $x = E_i^{min}$ and $y = E_j^{min}$ for two $\equiv$-classes E_i and E_j, it follows that $E_{G_{\equiv}}(E_i, E_j) \leq E_G(x, y)$. Furthermore, due to the property of reduction construction, there is a path in ${G_{\equiv}}^R$ between E_i and E_j with weight no greater than $E_{G_{\equiv}}(E_i, E_j)$. The same path, but now between the nodes with the minimal indices of the $\equiv$-classes, can be found in $({G_{\equiv}}^R)^+$. Thus, there is a path in $({G_{\equiv}}^R)^+$ with weight no greater than $E_G(x, y)$.
   - If $x, y \in E_i$ for some $\equiv$-class E_i, an easy argument gives that $E_{({G_{\equiv}}^R)^+}(x, y) = E_{G^C}(x, y) \leq E_G(x, y)$.
   - Consider the case when $x \in E_i$ and $y \in E_j$ for two different $\equiv$-classes, and assume that $E_G(x, y) = m$.

Let $m_1 = E_{(G_{\equiv^R})^+}(x, E_i^{min})$, $m_2 = E_{(G_{\equiv^R})^+}(E_i^{min}, E_j^{min})$, and $m_3 = E_{(G_{\equiv^R})^+}(E_j^{min}, y)$. Note that by the reduction construction $m_2 \leq E_{G^C}(E_i^{min}, E_j^{min})$. Then there is a path in $(G_{\equiv}{}^R)^+$ from x to y via E_i^{min} and E_j^{min} with weight $m_1 + m_2 + m_3$. Now, if $m < m_1 + m_2 + m_3$, there is a path in G from E_i^{min} to E_j^{min} of weight $m - m_1 - m_3 < m_2$ contradicting that m_2 is the weight of the shortest path in G between E_i^{min} and E_j^{min}. Thus the path x, E_i^{min}, E_j^{min}, y in $(G_{\equiv}{}^R)^+$ has weight no greater than the edge (x, y) in G.

2. Next we prove that $(G_{\equiv}{}^R)^+$ has minimal number of edges by showing that whenever $H^C = G^C$ then H has at least as many edges as $(G_{\equiv}{}^R)^+$.

   As $H^C = G^C$, H and G induces the same $\equiv$-equivalence relation on the same zero-length cycles. Obviously the fewest edges that will identify k (> 1) vertices, with respect to $\equiv$ is k. Hence, $(G_{\equiv}{}^R)^+$ uses a minimal number of edges between vertices in the same $\equiv$-equivalence class.

   Now let (E_i^{min}, E_j^{min}) be an edge in $(G_{\equiv}{}^R)^+$ with weight m. We claim that H must have at least one edge from E_i to E_j.

   Assume that this is not the case. Then, as $H^C = G^C$, there must be a path in H from E_i^{min} to E_j^{min} as shown in Figure 4 such that $m = \sum_{i=0}^{k+2} v_i + \sum_{i=0}^{k+1} w_i$.

   Now let $m_0 = E_{(G_{\equiv^R})^+}(E_i^{min}, E_0^{min})$, $m_1 = E_{(G_{\equiv^R})^+}(E_0^{min}, E_1^{min})$, $\ldots$, $m_{k+1} = E_{(G_{\equiv^R})^+}(E_k^{min}, E_i^{min})$ (illustrated with dashed lines in Figure 4). Then $m_0 \leq v_0 + w_0 + v_1'$, $m_1 \leq v_1'' + w_1 + v_2'$, $\ldots$, $m_{k+1} \leq v_{k+1}'' + w_{k+1} + v_{k+2}$, where $v_1 = v_1' + v_1''$, $v_2 = v_2' + v_2''$, $\ldots$, $v_{k+1} = v_{k+1}' + v_{k+1}''$.

   It follows that $\sum_{i=0}^{k+1} m_i \leq m$. Hence (E_i, E_j) is redundant in $(G_{\equiv}{}^R)^+$ and can be removed, contradicting Lemma 1. □

First, note that the above construction of $(G_{\equiv}{}^R)^+$ is well-defined as $G_{\equiv}$ is a zero-cycle free graph and the reduction construction of Theorem 1 thus applies. Given the closure G^C of G the constructions of Definitions 2 and 3 can be computed in $\mathcal{O}(n^2)$. Since G^R is computed from G in $\mathcal{O}(n^3)$, it follows that also $(G_{\equiv}{}^R)^+$ can be constructed in $\mathcal{O}(n^3)$. Now applying the above construction to the graph H of Figure 3 (a), we first note that $H_{\equiv}{}^R = H_{\equiv}$ as $H_{\equiv}$ has no redundant edges. Expanding $H_{\equiv}$ with respect to the vertex ordering $x_0 < x_1 < x_2 < x_3$ gives the graph of Figure 3 (c), which according to Theorem 2 above is the shortest-path reduction of H.

Experimental results show that the use of minimal constrain systems (obtained by the above shortest-path reduction algorithm) as a compact data structure leads to truly significant space-savings in practical reachability analysis of timed systems: the space-savings are in the range 68–85%. We refer to Section 5 for more details.

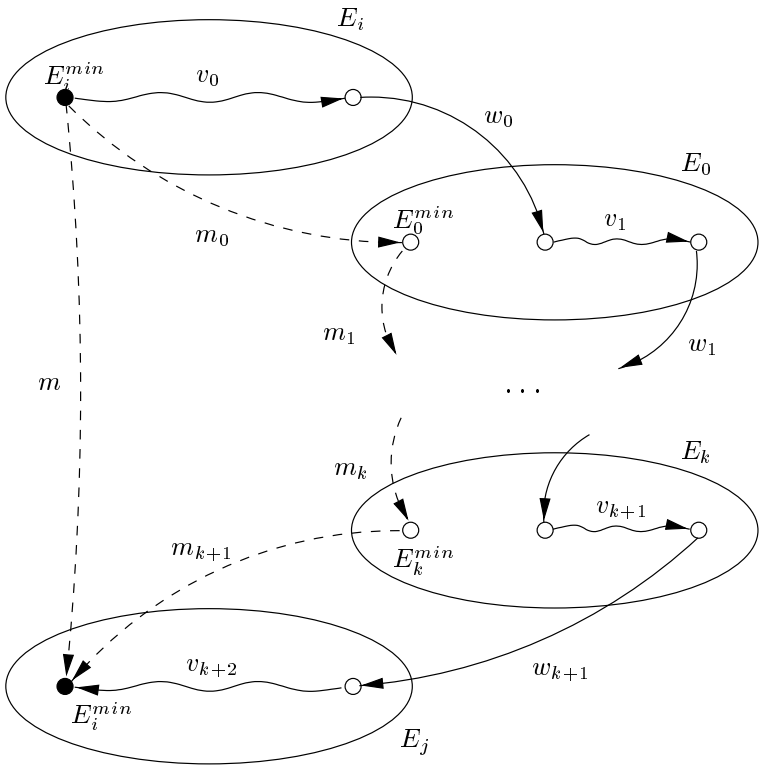


Figure 4. A path in the graph H.

4. Global Reductions and Control Structure Analysis

The preceding section is about *local* reductions in reachability analysis in the sense that the technique developed is for *each* individual symbolic state. In this section, we shall develop a *global* reduction technique to reduce the total number of symbolic states to save in the *global* data structure i.e. the passed list.

4.1. Potential Space-Reductions

We recall the standard reachability analysis algorithm for finite graphs (see e.g. [27]). It is similar to the one in Figure 1, but simpler as no constraints but only control nodes are involved. The algorithm repeats three main operations: *examining* every new encountered node (to see if it is in the passed list), *exploring* the new encountered nodes (computing all their successors for further analysis), and *saving* the explored nodes in the passed list until all reachable nodes are present in the list (i.e. all new encountered nodes are already in the passed list).

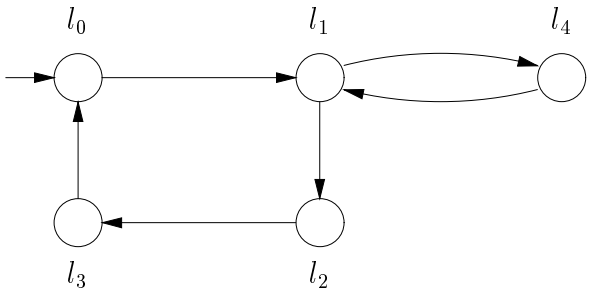


Figure 5. Illustration of Space-Reduction.

Note that the saving of an explored node is to ensure termination and also to avoid repeated exploration of nodes with more than one incoming edge. However it is not necessary to save all reachable nodes. Consider for example, the simple graph in Figure 5 with initial node l_0. Clearly, there is no need to save node l_2, l_3 or l_4 as they will be visited only once if l_1 is present in the passed list.

In fact, to guarantee termination on a finite graph, it is sufficient to save only one node for each cycle in the graph. For example, as l_1 covers the two cycles of the graph in Figure 5, in addition to l_2, l_3, and l_4, it is not necessary to save l_0 either. In general, for a finite graph, there is a minimal number of nodes to save in the passed list in order to guarantee termination. However the trade-off of the space-saving strategy may be increased time-consumption. Consider the same graph of Figure 5. If node l_0 is not present in the passed list, it will be explored again whenever l_3 is explored. This can be avoided by saving l_0 when it is first visited. But the difference from saving l_1 is that saving l_0 is for efficiency and l_1 for termination.

Now we again recall the abstract reachability algorithm in Figure 1 for timed systems. To ensure termination and also to avoid repeated exploration of states (that have more than one predecessors), it saves every new encountered state (l, D) in the passed list when the inclusion-checking for $D \subseteq D'$ fails (i.e. $D \not\subseteq D'$). Obviously this is not necessary if all the predecessors of (l, D) already exist in the PASSED-list. Similar to the case for finite graphs, for termination, we need to save only one state for every *dynamic loop* of a timed automaton.

Definition 4. [Dynamic Loops] Assume a timed automaton with an initial state (l_0, D_0). The set of symbolic states $L_d = \{(l_1, D_1) \ \ldots \ (l_n, D_n)\}$ is a dynamic loop of the timed automaton if $(l_1, D_1) \rightsquigarrow (l_2, D_2) \ \ldots \ (l_{n-1}, D_{n-1}) \rightsquigarrow (l_n, D_n)$ and $(l_n, D_n) \rightsquigarrow (l_1, D_1')$ with $D_1' \subseteq D_1$, and (l_0, D_0) is reachable in the sense that $(l_0, D_0) \rightsquigarrow \ldots \rightsquigarrow (l_1, D_1)$. A symbolic state is said to cover a dynamic loop if it is a member of the loop. □

We claim that to ensure termination, it is sufficient (but not necessary) to save a set of symbolic states that cover all the dynamic loops. Now, the problem is how to compute efficiently such a set.

4.2. Control Structure Analysis and Application

We shall utilize the statical structure of an automaton to identify potential candidates of states to cover dynamic loops.

Definition 5. [Statical Loops and Entry Nodes] A set of nodes $L = \{l_1, \ldots, l_n\}$ of a timed automaton is a statical loop if there is a sequence of edges $l_1 \longrightarrow l_2 \cdots l_{n-1} \longrightarrow l_n$ and $l_n \longrightarrow l_1$ where $l_i \longrightarrow l_j$ denotes that $l_i \xrightarrow{g,r} l_j$ for some g, r is an edge of the automaton. A node $l_i \in L$ is an entry node of the statical loop L if it is an initial node of the automaton or there exists a node $l \notin L$ (outside of the loop) and an edge $l \longrightarrow l_i$. Further, we say that a vector of nodes (i.e. a node of a network) is an entry node if any of its components are entry nodes. □

For example, nodes l_0, l_1, l_2 and l_3 in Figure 5 constitute a statical loop with entry nodes l_0 and l_1; another statical loop is nodes l_1 and l_4 with entry node l_1. In general, since the sets of control nodes and edges of a timed automaton are finite, the number of statical loops is finite and so is the set of entry nodes of all statical loops. In fact the set of entry nodes of a timed automaton can be easily computed by statical analysis using a stack or a slightly modified loop detecting algorithm (see e.g. [29]).

Now note that according to Definition 4, a dynamic loop (a set of symbolic states) must contain a subset of symbolic states whose control nodes constitute a statical loop. As a statical loop always contains an entry node, we have the following fact.

PROPOSITION 1 *Every dynamic loop of a timed automaton contains at least one symbolic state (l, D) where l is an entry node.*

Proof: Standard proof by contradiction. □

Following Proposition 1, to cover all the dynamic loops, we may simply save all the states whose control-nodes are an entry node, and ignore the others. Obviously, this will not give much reduction when dynamic loops include mostly entry nodes, which is the case when a network of automata contains a component whose nodes are mostly entry nodes e.g. a testing automaton. For networks of automata, we adopt the strategy of saving the *first derived* states whose control nodes are an entry node, known as *covering states* in the following sense.

Definition 6. [Covering States] Assume a network of timed automata with an initial state (l_0, D_0) and a given symbolic state (l, D). We say that (l, D) is a covering state of the network if it is reachable in the sense that there exists a sequence of symbolic transitions $(l_0, D_0) \rightsquigarrow (l_1, D_1) \ldots (l_n, D_n) \rightsquigarrow (l, D)$ and

an i (standing for the ith component of the network) such that $l[i]$ is an entry node and $(l_n, D_n) \rightsquigarrow (l, D)$ is derived by an edge $l_n[i] \xrightarrow{g,r} l[i]$ for some g and r. □

From the above definition, it should be obvious that we can easily decide whether a reachable symbolic state is a covering state by an on-the-fly algorithm when the entry nodes of all the component automata are known through statical analysis as discussed earlier.

Finally, we claim that the set of *covering* states of a network covers all its dynamic loops and therefore it suffices to keep them in the passed list for the sake of termination in reachability analysis[7].

THEOREM 3 *Every dynamic loop of a network of timed automata contains at least one covering state.*

Proof: Assume a dynamic loop $L_d = (l_1, D_1) \rightsquigarrow \ldots \rightsquigarrow (l_k, D_k)$ with no covering states. However according to Proposition 1, L_d contains at least one entry node. Further, assume (without loss of generality) that the symbolic state $(l, D) \in L_d$ is an entry node and the components $l[1], \ldots, l[m]$ of l are all in an entry node, and all the other components of l, i.e. $l[m+1], \ldots, l[n]$, are not.

Now, we claim that if L_d contains no covering states, the set of components $l_i[1], \ldots, l_i[m]$ will remain in an entry node in all symbolic states $(l_i, D_i) \in L_d$. Otherwise, if the set of local entry nodes changes, either grows or reduces, it will introduce a covering state. The case of growing is obvious due to the definition for covering states. The argument for the case of reducing is the same as the control nodes of all the components will reach l_1 again by the end of L_d, meaning that the set will sooner or later grows again.

In fact, the assumption that L_d contains no covering states, implies an even stronger property, that is, all symbolic transitions in L_d are derived by components in $l_i[m+1], \ldots, l_i[n]$. A transition is derived by a local transition of a component in $l[1], \ldots, l[m]$, means that the set of local entry nodes will either grow or reduce (discussed above) or the local transition leaves the current entry node and enters an another entry node. The later case implies that the new entry node is a covering state.

Now we construct L'_d by removing $l_i[1], \ldots, l_i[m]$ from all symbolic states $(l_i, D_i) \in L_d$, that is, L'_d contains only the components that are not in an entry nodes. Obviously, all the symbolic transitions of L_d are also in L'_d; thus L'_d must be a loop by definition. However, L'_d contains no components that are in an entry node. This contradicts Proposition 1. □

An improved reachability algorithm according to the saving strategy induced from Theorem 3 (i.e. saving only the covering sates in the passed list) has been implemented in UPPAAL. Our experimental results show that the space-reduction is between 13–72% (see Table 1 and 2 in Section 5).

5. Experimental Results

The techniques developed in preceding sections have been implemented and added to the tool UPPAAL[8][9]. In this section we present the results of an experiment where both the original version of UPPAAL and its extension were applied to verify the following six well-studied examples from the literature:

Philips Audio Protocol (Audio) The protocol was developed and implemented by Philips to exchange control information between components in audio equipment using Manchester encoding. The correctness of the encoding relies on timing delays between signals. It is first studied and manually verified in [10].

We have verified that the main correctness property holds of the protocol, i.e. all bit streams sent by the sender are correctly decoded by the receiver [24], if the timing error is $\pm 5\%$.

Philips Audio Protocol with Bus Collision (Audio w. Collision) This is an extended variant of Philips audio control protocol with bus collision detection [8]. It is significantly larger than the version above since several new components (and variables) are introduced, and existing components are modified to deal with bus collisions.

In the experiment we checked that correct bit sequences are received by the receiver (i.e. Property 1 of [8]), using the error tolerances set by Philips.

Bang & Olufsen Audio/Video Protocol (Bang & Olufsen) This is an audio control protocol highly dependent on real-time. The protocol is developed by Bang & Olufsen, to transmit messages between audio/video components over a single bus, and further studied in [18].

In the experiment we have verified the correctness criteria of the protocol. We refer the reader to Section 5.1 of [18] for more details.

Box Sorter (Box Sorter) The example of [25] is a model of a sorter unit that sorts red and blue boxes. When the boxes moves down a lane they pass a censor and a piston. The sorter reads the information from the censor and sorts out the red boxes by controlling the position of the piston. We have shown, using UPPAAL, that only blue boxes arrive at the end of the lane.

Manufacturing Plant (Manufact. Plant) The example is a model of the manufacturing plant of [28, 14]. It is a production cell with: a 50 feet belt moving from left to right, two boxes, two robots and a service station. Robot A moves boxes off the rightmost extreme of the belt to the service station. Robot B moves boxes from the service station to the left-most extreme of the belt.

Assuming an initial distance between the boxes on the belt we verified that no box will fall off the belt.

Mutual Exclusion Protocol (Mutex 2–Mutex 5) It is the so-called Fischer's protocol that has been studied previously in many experiments, e.g. [1, 30].

	Current	Local		Global		Local+Global	
	#	#	%	#	%	#	%
Audio	828	219	26	774	93	206	25
Audio w. Collision	646 092	198 178	31	370 800	57	111 632	17
Bang & Olufsen	778 288	249 175	32	642 752	83	204 795	26
Box Sorter	625	139	22	175	28	36	6
Manufact. Plant	92 592	27 042	29	50 904	55	14 933	16
Mutex 2	225	44	20	99	44	18	8
Mutex 3	3 376	621	18	1 360	40	240	7
Mutex 4	56 825	9 352	16	22 125	39	3 532	6
Mutex 5	1 082 916	158 875	15	416 556	38	59 720	6
Train Crossing	464	130	28	384	83	114	25

Table 1. Space performance statistics: number of constraints (#) and percentage of Current (%).

The protocol is to ensure mutual exclusion among several processes competing for a critical section using timing constraints and a shared variable. In the experiment we use the version of the protocol where a process may recover from failed attempts to enter the critical section, and also eventually leave the critical section [22].

The protocol is shown to enjoy the invariant property: There is never more than one process existing in the critical section. The results for 2 to 5 processes are shown in Table 1 and 2.

Train Crossing Controller (Train Crossing) It is a variant of the train gate controller [19]. An approaching train signals to the controller which reacts by closing the gate. When the train have passed the controller opens the crossing. We have verified that the gate is closed whenever a train is close to the crossing.

In Table 1 and 2 we present the space (in number of timing constraints stored on the PASSED-buffer) and in the time requirements (in seconds) of the examples on a Sun SPARCstation4 equipped with 64 MB of primary memory. Each example was verified using the current algorithm of UPPAAL (Current), and using modified algorithms for: Compact Data Structure for Constraints (Local), Control Structure Reduction (Global), and their combination (Local+Global).

As shown in Table 1 and 2 both techniques give truly significant space savings: Compact Data Structure for Constraints saves 68–85% of the original consumed

	Current	Local		Global		Local+Global	
	sec	sec	%	sec	%	sec	%
Audio	0.44	0.43	98	0.44	100	0.47	107
Audio w. Collision	3 465.22	2 067.37	60	1 515.88	44	929.22	27
Bang & Olufsen	13 240.49	6 967.38	53	9 348.48	71	4 966.79	38
Box Sorter	0.20	0.18	90	0.41	205	0.41	205
Manufact. Plant	155.61	39.85	26	56.61	36	24.22	16
Mutex 2	0.13	0.14	108	0.15	115	0.14	108
Mutex 3	1.40	0.67	48	0.65	46	0.51	36
Mutex 4	102.49	24.48	24	25.97	25	12.14	12
Mutex 5	14 790.56	3 299.96	22	3 111.21	21	1 138.32	8
Train Crossing	0.19	0.18	95	0.20	105	0.18	95

Table 2. Time performance statistics: seconds (sec) and percentage of Current (%).

space while Control Structure Reduction demonstrates more variation saving 13–72%. Both methods result in better time-performance on the examples consuming more than half a second, whereas the time-perfomance is worse on the smaller examples. Most significant is that the two techniques are completely orthogonal, witnessed by the numbers for the combined technique which shows a space-saving between 75% and 94%.

6. Conclusion

In this paper, we have two contributions to the development of efficient data structures and algorithms for memory-usage reduction in the automated analysis of timed systems.

Firstly, we have presented a compact data structure, for representing the subsets of Euclidean space that arise during verification of timed automata, which provides *minimal* and *canonical* representations for clock constraints, and also allows for efficient inclusion checks between constraint systems. The data structure is based on an $\mathcal{O}(n^3)$ algorithm which, given a constraint systems over real-valued variables consisting of bounds on differences, constructs an equivalent system with a minimal number of constraints. It is essentially a minimization algorithm for weighted directed graphs, that extends the transitive reduction algorithm of [3] to weighted graphs. Given a weighted, directed graph with n vertices, it constructs in time

$\mathcal{O}(n^3)$ a reduced graph with the minimal number of edges having the same shortest path closure as the original graph.

Secondly, we have developed an on-the-fly reduction technique to minimize the space-usage by reducing the total number of symbolic states to save in reachability analysis for timed systems. The technique is based on the observation that to ensure termination in reachability analysis, it is not necessary to save all the explored states in memory, but only certain critical states. Based on static analysis of the control structure of timed automata, we are able to compute a set of *covering states* that cover all the dynamic loops of a system. The set of covering states may not be minimal but sufficient to guarantee termination in an on-the-fly reachability algorithm.

The two techniques and their combination have been implemented in the tool UPPAAL. Our experimental results demonstrate that the techniques result in truly significant space-reductions: For a number of well-studied examples in the literature the space saving is between 75% and 94%, and in all large examples time-performance is improved. Noteworthy is also the observation that the two techniques are completely orthogonal.

As future work, we wish to further study the global on-the-fly reduction technique to identify the *minimal* sets of covering states that ensure termination and also avoid repeated explorations in reachability analysis for timed systems.

Notes

1. Several verification tools for timed systems (e.g. UPPAAL [9]) have been implemented based on this algorithm.
2. For reasons of simplicity and clarity in presentation we have chosen only to consider the non-strict orderings. However, the techniques given extends easily to strict orderings.
3. We assume that D has been simplified to contain at most one upper and lower bound for each clock and clock-difference.
4. To be precise, it is the inclusion between the *solution sets* for D and D'.
5. This would correspond to constraint systems with empty solution set.
6. "$<$" refers to the assumed ordering on the vertices of G.
7. Note that this is only a sufficient condition but not necessary.
8. For more information about the tool UPPAAL, see the web site `http://www.uppaal.com/`.

References

1. Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, number 600 in Lecture Notes in Computer Science, 1992.
2. Luca Aceto, Auguto Bergueno, and Kim G. Larsen. Model checking via reachability testing for timed automata. In Bernard Steffen, editor, *Proc. of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 263–280. Springer–Verlag, 1998.
3. A.V. Aho, M.R. Garey, and J.D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, 1(2):131–137, June 1972.

4. R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of Int. Colloquium on Algorithms, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 322–335, July 1990.
5. H. R. Andersen. Partial Model Checking. In *Proc. of Symp. on Logic in Computer Science*, 1995.
6. Eugene Asarin, Oded Maler, and Amir Pnueli. Data–structures for the verification of timed automata. In *Proc. of the Int. Workshop on Hybrid and Real-Time Systems*, 1997.
7. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
8. Johan Bengtsson, W.O. David Griffioen, Kare J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of the 8*th *Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.
9. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, March 1996.
10. D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 863 in Lecture Notes in Computer Science, 1994.
11. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. In *Proc. of IEEE Symp. on Logic in Computer Science*, 1990.
12. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *Proc. of the 5*th *Int. Conf. on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, 1993.
13. E. M. Clarke, O. Grümberg, and D. E. Long. Model Checking and Abstraction. *Principles of Programming Languages*, 1992.
14. C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 66–75, December 1995.
15. David Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 197–212. Springer–Verlag, 1989.
16. E. A. Emerson and C. S. Jutla. Symmetry and Model Checking. In *Proc. of the 5*th *Int. Conf. on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, 1993.
17. P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proc. of IEEE Symp. on Logic in Computer Science*, pages 406–415, 1991.
18. Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, December 1997.
19. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A Users Guide to HYTECH. Technical report, Department of Computer Science, Cornell University, 1995.
20. Thomas. A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.
21. Gerard Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.
22. Kare J. Kristoffersen, Francois Larroussinie, Kim G. Larsen, Paul Pettersson, and Wang Yi. A Compositional Proof of a Real-Time Mutual Exclusion Protocol. In *Proc. of the 7*th *Int. Joint Conf. on the Theory and Practice of Software Development*, April 1997.
23. Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87, December 1995.
24. Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 575–586. Springer–Verlag, October 1995.

25. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
26. Florence Pagani. Partial orders and verification of real-time systems. In Bengt Jonsson and Joachim Parrow, editors, *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1135 in Lecture Notes in Computer Science, pages 327–346. Springer–Verlag, 1996.
27. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
28. A. Puri and P. Varaiya. Verification of hybrid systems using abstractions. In *Hybrid Systems Workshop*, number 818 in Lecture Notes in Computer Science. Springer–Verlag, October 1994.
29. Robert Sedgewick. *Algorithms*. Addison-Wesley, 2*nd* edition, 1988.
30. N. Shankar. Verification of Real-Time Systems Using PVS. In *Proc. of the* 5th *Int. Conf. on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science. Springer–Verlag, 1993.
31. A. Valmari. A Stubborn Attack on State Explosion. *Theoretical Computer Science*, 3, 1990.
32. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of Symp. on Logic in Computer Science*, pages 322–331, June 1986.
33. Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real–time transition systems. In *Proc. of the* 5th *Int. Conf. on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 210–224, 1993.
34. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the* 7th *Int. Conf. on Formal Description Techniques*, pages 223–238, 1994.

, , 1–37 ()

Formal Verification of Dynamic Properties in an Aerospace Application

SIMIN NADJM-TEHRANI simin@ida.liu.se
Dept. of Computer and info. Science,
Linköping University, S-581 83 Linköping, Sweden

JAN-ERIK STRÖMBERG janerik@dst.se
DST Control AB, Mjärdevi Science Park,
Teknikringen 6, S-583 30 Linköping, Sweden

Editor: Thomas Henzinger

Abstract. Formal verification of computer-based engineering systems is only meaningful if the mathematical models used are derived systematically, recording the assumptions made at each modelling stage. In this paper we give an exposition of research efforts in cooperation with aerospace industries in Sweden. We emphasize the need for modelling techniques and languages covering the whole spectrum from informal engineering documents, to hybrid mathematical models. In this modelling process we give as much weight to the physical environment as to the controlling software. In particular, we report on our experience using switched bond graphs for the modelling of hardware components in hybrid systems. We present the basic ideas underlying bond graphs and illustrate the approach by modelling an aircraft landing gear system. This system consists of actuating hydromechanic and electromechanic hardware, as well as controlling components implemented in software and electronics. We present a detailed analysis of the closed loop system with respect to safety and timeliness properties. The proofs are carried out within the proof system of Extended Duration Calculus.

Keywords: formal verification, hybrid system, physical modelling, bond graph, aerospace application, Duration Calculus

1. Introduction

In this article we present some insights gained from the work performed in a multidisciplinary project in cooperation with the Swedish aerospace industries, by computer science, electrical engineering, and mechanical engineering departments at Linköping university. The project concerns generation of models and analysis of *hybrid* systems – mathematical models including both continuous and discrete elements. The article addresses modelling and formal verification of a system, involving hydromechanical and electromechanical sensors and actuators as well as electronic and software modules performing diagnosis and control. The technical system, hereafter referred to as "the system", is moreover in dynamic interaction with a human operator (the pilot).

Our main thesis is that verifying properties of computer based systems benefits from modelling both the embedded controller and its surrounding physical environment. Furthermore, unless care is taken in developing the mathematical models which form the basis of formal verification, the results of verification can not be

relied upon. Therefore, rather than verifying properties of ad hoc models we take a number of steps for systematically deriving the models from engineering design documents. The models of the hardware and the software are typically fundamentally different in their character[1]. Hence, different competences are required in development and documentation of these models. However, it is the composition of the models for these two parts, also referred to as the *closed loop* model, which is needed for proving the properties we are interested in. The work reported here has been partially presented in international conferences in the form of shorter articles. This article can therefore be seen as the full version of [21, 32].

Deriving mathematical models from engineering documents, is an inherently difficult task. We have approached this problem by adopting an iterative process in which models are successively refined as a result of earlier analysis. The pragmatic reason for this is the following. The level of detail in the model is dependent on factors such as the requirement specification, assumptions made about the physical components, and the choice of verification technique. Hence, in real-world applications, the modelling and the analysis processes must go hand-in-hand. For this kind of iterative process to be realistic in a multi-disciplinary setting, the formal models derived and the assumptions made must be easy to convey to the domain specialists. Also, new assumptions derived from results made available by earlier analysis, must be easy to incorporate into the model at any time.

The above process necessitates the employment of a range of formalisms and methodologies in our application. To mention a few, we have, at the physical level of abstraction, ideal physical model diagrams and switched bond graphs [33]. At the mathematical level of abstraction, we have employed hybrid transition systems (HTS) [18, 19], and an architecture for the top-level decomposition of hybrid systems [17]. This modular framework allows us to plug in a refined version of an arbitrary module without having to consider (or redo) the rest of the model. In connection with the verification technique, we have experimented with several other mathematical modelling languages: Linear Hybrid Automata (LHA) [1] with the tool HyTech, and Extended Duration Calculus (EDC) [6] from the hybrid family of languages, as well as Esterel (I/O automata) [4] and statecharts [12] from the synchronous family of languages.

In this article we concentrate on the derivation of mathematical models for a landing gear system and the analysis of safety and timeliness properties of the system within EDC. The application of the other verification techniques are covered in a longer technical report [22].

1.1. The Example

The techniques above are illustrated in the context of a landing gear in which the response to the landing command by a pilot are analysed (see Figure 1). When deriving the mathematical models we note that some major elements of the physical configuration have already been fixed due to other demands on the aircraft as a whole (weight, etc.). Also, some back-up mechanisms have already been envisaged. That is, under abnormal situations there should always be possible to initiate land-

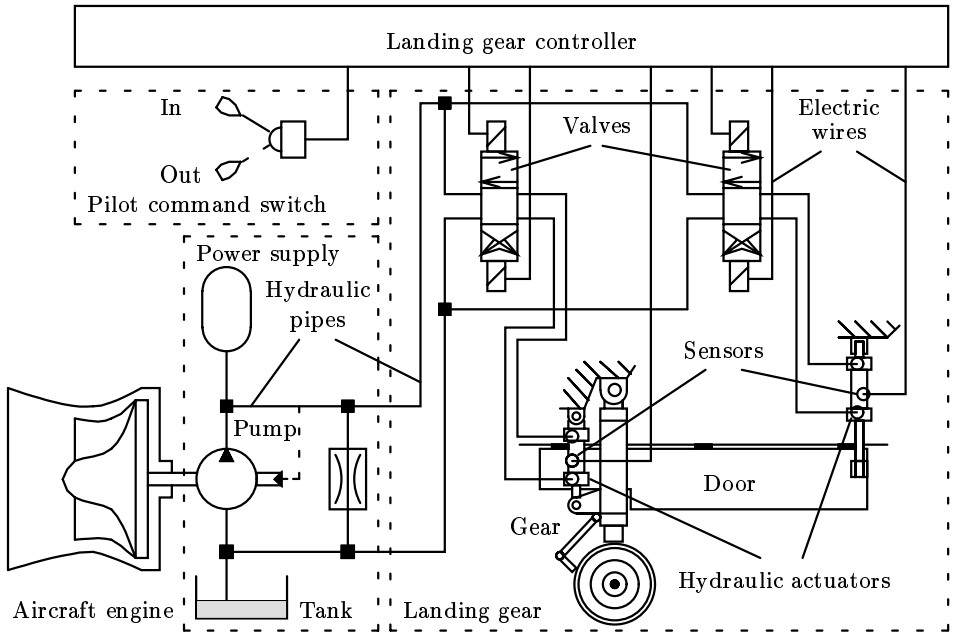


Figure 1. The landing gear system as documented by a mechanical engineer.

ing using a reserve power supply. Thus we have started the modelling activity at an architectural level.

A basic problem at this level is that the segregation of the landing gear subsystem from the rest of the aircraft is not a straight-forward task. This is due to the fact that achieving the goals of the landing gear is not solely dependent on the functional correctness of a single controller. Rather, the proper operation of the landing gear is possible under certain conditions in the rest of the aircraft. For example the physical operation of the mechanical door and gear components is dependent on hydraulic power at certain locations in the hydraulic power supply system and at certain times. The load on the hydraulic system itself is dependent on several other "clients" to which hydraulic pressure is delivered. Thus, in order to identify the mode of operation in the landing gear control system there exists another module for monitoring and diagnosis of the hydraulic power supply.

Based on the above observations we have identified a sub-system consisting of two blocks for modelling and analysis. The two blocks cover the power supply and the mechanical linkage parts respectively – each consisting in turn of physical mechanisms and control subsystems. Figure 2 shows the topology of the system under study, where the full arrows denote directions of information exchange and half arrows directions of (positive) energy exchange. In other words, full arrows correspond to signals transmitted (one-way) via some abstract communication channel (e.g. computer programs and communication protocols) and half arrows to bilateral

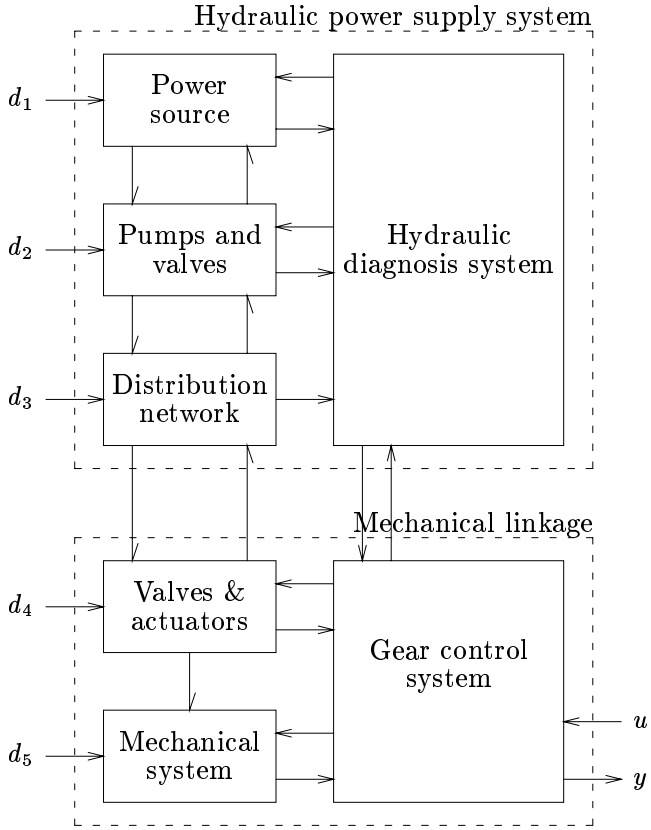


Figure 2. An abstraction of the landing gear system.

signals transmitted via some physical structure (e.g. electromechanical or hydromechanical hardware). The inputs denoted by d_i are abstractions of other activities in the aircraft treated as disturbances, u denotes pilot commands, and y denotes system state information delivered to the pilot.

1.2. Summary of the Work

The work in the project has led to derivation of physical and mathematical abstractions for all components of the system. In the current presentation we exclude the diagnosis subsystem, since only proofs pertaining to one of its modes are presented here.

At the first stage, models for the hydromechanical subsystems, in terms of schematic engineering diagrams, have been developed. These models have been transformed to the energy-based graphical language of switched bond graphs. The

latter step significantly aids the derivation of mathematical models for mode-switching physical systems. These are physical systems which can not be described by one set of differential and algebraic equations (DAE); rather different sets of DAE are needed depending on the discrete mode in the system. Next, models for diagnosis and controller modules (to be implemented in hardware or software) have been developed.

The modelling phase was followed by proofs of the following three properties for several versions of landing gear models.

- The door and the gear do not collide under movement
- Whenever the pilot issues the landing (airborne) command, the gear will be out (in) and the door closed within T seconds.

This article presents the proofs for a static controller in combination with two different environment models: one in which the environment variables change as a linear function of time, and the second where the environment model is a non-linear system of DAE. This corresponds to a mathematical model for a more refined version of the hydraulic supply system in which the hydraulic pressure is not assumed to be constant as in the first case. It is rather assumed to be regulated by a hydromechanic pump.

The model of the composed system was in both cases analysed using the proof system of EDC, leading to proofs for the above properties. While the closed loop model based on the time-linear evolutions in the environment could also be analysed using model checking (e.g. HyTech), the non-linear model required a combination of analysis and logical deductions. Thus, the point with the proof of the time-linear model with EDC was to provide the necessary structure over which the same proofs for the non-linear model of the hydromechanical system could be based.

2. From Schematic Diagrams to Mathematical Models

We let the term 'apparatus' refer to an object for which the behaviour is described by physical laws. By behaviour we mean the evolution in time of measurable physical quantities such as pressure, flow, velocity, current etc. In the context of computer controlled systems, apparatus models generally cover models of actuators, the plant actuated by these actuators, and the sensors used to measure the effects of the actuation. As a consequence, apparatus models generally involve several different physical domains. The apparatus model in Figure 1 involves at least three different physical domains: electric, hydraulic and mechanic.

Industrial apparatus models are typically provided in terms of *schematic diagrams*. Within each physical domain there is a set of generally accepted icons representing more or less complex standard apparatuses such as pilot controlled valves, check valves, double-acting cylinders etc. These icons are fairly well defined in the sense that they represent specific functions, e.g. electrical control of flow, rectification of flow, mechanical actuation by flow etc. They are well understood

by experts within the same physical domain, but are often difficult to comprehend by experts outside that domain.

Also, the icons used in schematic drawings are ambiguous with respect to the underlying physics; it is not clear from the syntax which physical effects are taken into account. The reason for this is that the behaviour of an apparatus depends on the circumstances in which it is used. Often the effects needed to be considered depend on the required accuracy of the model, timing properties, dynamic range etc. Moreover, certain interaction effects are not so naturally captured in terms of apparatus models – an example of such interaction effects being friction. Hence, the type of models typically used in industry are in themselves not informative enough for determining a unique mathematical model suitable for e.g. formal verification. From this perspective there is therefore a genuine need for modelling according to physical principles.

2.1. Physical Modelling

In our project the apparatus models provided by the industry were made precise by means of *switched bond graphs*. Switched bond graphs [34, 35, 33, 30] are based on the graphical bond graph language [15], earlier introduced by Henry M. Paynter [25]. The language was originally proposed as a means to bridge the gap between different physical domain experts in the context of complex industrial plants.

To meet these requirements, bond graphs are based on *energy* concepts. By means of a set of well defined primitive energy concepts, bond graphs can be precisely interpreted by experts in several different domains since they have a precise mathematical and physical interpretation. Further, the bond graph concepts are powerful enough to capture even very complex mechanisms using only a few symbols.

2.2. Elementary Bond Graph Concepts

More specifically, bond graphs are based on the theory known as 'first principle of energy conservation'. As a consequence, the nodes in a bond graph represent degenerate primitive energy processes and the directed arcs represent *potential* flow of energy. The arcs are also referred to as *power bonds*, or simply *bonds*. The direction of the bonds represent the direction of *positive* energy flow. Every bond is associated with a pair (e, f) of variables referred to as *generalised power variables*. The power variable e is further referred to as *effort* and the variable f as *flow*.

For a given physical domain, the generalised power variables represent a pair of *physical quantities* such that $[ef]$ = Watts, where $[q]$ denotes the *unit* of the physical quantity represented by q. Recall that a physical quantity is a pair (value, unit) and that a variable q representing it is a real-valued function on time, i.e. $q : \mathbb{R} \to \mathbb{R}$.

By means of e and f the semantics of a bond can now be summarised:

- $ef \geq 0$ iff the direction of energy flow coincides with the direction of the bond.

- $|ef|$ represents the magnitude of the energy flow, i.e. the power transferred by the bond.

EXAMPLE: Consider the hydraulic domain. Let e represent pressure, i.e. $[e] = \mathrm{N/m^2}$, and f volumetric flow, i.e. $[f] = \mathrm{m^3/sec}$. Then we have $[ef] = [e][f] = \frac{\mathrm{N}}{\mathrm{m^2}}\frac{\mathrm{m^3}}{\mathrm{sec}} = \frac{\mathrm{Nm}}{\mathrm{sec}} = \mathrm{Watts}$, i.e. the pair (e, f) is indeed an instance of power variables. In the same way we show that voltage (e) and current (f) is a pair of power variables in the electrical domain and that torque (e) and angular velocity (f) is a pair in the rotating mechanical domain. □

Bond graphs define a necessary and sufficient set of primitives for the modelling of a wide range of practical systems. The necessary and sufficent set of primitives consists of five elements, but normally a more practical set of nine elements is used. These nine elements are grouped into five categories as follows:

1. **Ideal sources:** ideal source of effort denoted **Se** and ideal source of flow denoted **Sf**.

2. **Ideal storages:** ideal storage of effort denoted **I** and ideal storage of flow denoted **C**.

3. **Ideal dissipation:** ideal energy dissipation denoted **R**.

4. **Ideal conversions:** ideal transformer denoted **TF** and ideal gyrator denoted **GY**.

5. **Ideal distributions:** common effort junction denoted **E** and common flow junction denoted **F**.

For example, the following instances of the abstract energy primitives are found in the hydraulic domain: a wide container (**Se**), accumulator (**C**), hydraulic restriction (**R**), hydromechanic pump (**TF**), parallel connection of components (**E**), and series connection of components (**F**).

It is important to note that the nine energy primitives are degenerate ideals. Hence, 'real' phenomena found in nature must be modelled by combining two or more of these primitives. Typically the nodes of a bond graph are labelled by the type (**Se**, **Sf**, **C**, ...) and a graph-unique identity. Hence a bond graph node is labelled **X:id**, where **X** is the type and **id** the unique identity.

2.3. Constitutive Relations

From an energy perspective, the behaviour of a physical mechanism is an n-ary relation over entities representing power variable pairs. Hence we have $n = 2m, m \in \mathbb{N}$, and m is sometimes referred to as the number of *power ports*.

For the ideal primitive mechanisms defined above, the set of possible behaviours is further restricted. The formal definition of this structure will be exemplified in what follows; further details, though informal, can be found in e.g. [2].

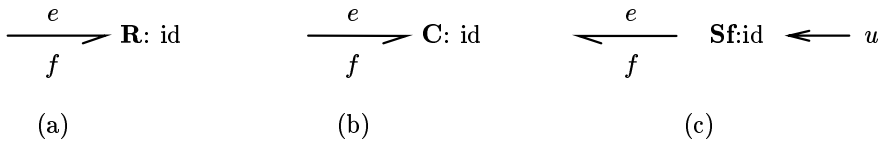


Figure 3. Some example bond graph primitives.

2.3.1. The Ideal Dissipation **R** Consider the primitive ideal dissipation, i.e. the **R**-element as depicted in Figure 3 (a). As a one-port **R**-element its behaviour is a relation over two time functions. Let e, f represent the power variables of the **R**-element. Then $\Phi \subseteq \mathbb{R}^2$, where $(e(t), f(t)) \in \Phi$ for all t, is a *constitutive relation* with the following properties:

1. $(0, 0) \in \Phi$.
2. $e(t)\, f(t) \geq 0$.
3. There exists a bijective function $\Phi_f : \mathbb{R} \to \mathbb{R}$ such that $(e(t), f(t)) \in \Phi$ iff $\Phi_f(e(t)) = f(t)$.

The first two requirements ensure that an ideal dissipation can never *generate* energy. The third requirement ensures existence of an inverse function Φ_e such that $(e(t), f(t)) \in \Phi$ implies $\Phi_e(f(t)) = e(t)$.

EXAMPLE: Consider a long narrow pipe carrying a hydraulic fluid. Let e be the pressure drop across, and f be the flow through the pipe. Let us assume the flow is *laminar*. Then, according to any standard book in fluid power,

$$\Phi = \{(e(t), f(t)) \,|\, k_1\, e(t) \Leftrightarrow f(t) = 0 \text{ for a specific } k_1 \in \mathbb{R}^+\}$$

where k_1 is a *physical parameter* computed from properties of the pipe (e.g. friction) and the fluid (e.g. viscousity). □

2.3.2. The Ideal Flow Storage **C** Consider the primitive ideal flow storage, i.e. the **C**-element as depicted in Figure 3 (b). Let e, f represent the power variables of the element, and x a *state variable* defined through $\dot{x} \Leftrightarrow f = 0$. By definition, x represents a physical quantity which is referred to as the *energy state* of the mechanism. If for instance $[f] = \mathrm{m}^3/\mathrm{sec}$, then it follows that $[x] = \mathrm{m}^3$, i.e. volume, which further supports the intuition behind the energy state concept.

Provided $\dot{x}(t)$ exists for all t, we have $(e(t), x(t)) \in \Phi$ for all t where $\Phi \subseteq \mathbb{R}^2$ is a relation with the same properties as the constitutive relation defined for the **R**-element.

EXAMPLE: Consider an open hydraulic accumulator. The hydraulic pressure e at the bottom of the accumulator (in which the fluid is affected by gravity g only) is proportional to the height h of the fluid level. Hence, if the cross section area A is

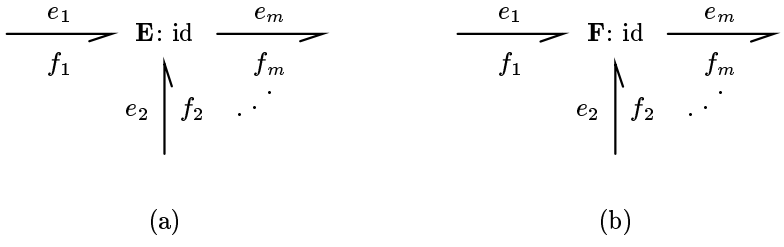


Figure 4. The two primitive ideal distribution elements. In this particular example $\sigma_1 = \sigma_2 = -\sigma_m = 1$.

constant with respect to the height, the pressure is proportional to the accumulated flow f. Let x be accumulated flow and ρ the density of the fluid. Then, for all t, we get

$$\Phi = \{(e(t), x(t)) \,|\, e(t) \Leftrightarrow k_2\, x(t) = 0 \text{ and } \dot{x}(t) \Leftrightarrow f(t) = 0\}$$

where $k_2 = \rho g / A$. □

2.3.3. The Ideal Flow Source **Sf** Consider the primitive ideal flow source **Sf** as depicted in Figure 3 (c). Let e, f be the power variables of the element and u an independent physical quantity referred to as the *modulation signal* or *control signal*. The full arrow in Figure 3 (c) underlines the fact that u represents a causally directed entity. We now have $(e(t), f(t)) \in \Phi$ for all t and $\Phi = \{(e(t), f(t)) \,|\, u(t) \Leftrightarrow f(t) = 0\}$. Note that an ideal effort source actually does not restrict e in any way.

2.3.4. The Ideal Distribution Elements Consider the primitive ideal junctions **E** and **F** as depicted in Figure 4. We note that the elements are *multi-ports* in that $m > 1$. As the name suggests, distribution elements are used to model interaction between other primitives, and hence $m \geq 2$ follows as an immediate consequence.

Let e_i, f_i be the power variables of bond i. For an **E**-junction we have, for all t

$$\{(e_1(t), f_1(t), \ldots, e_m(t), f_m(t)) \,|\, \sum_{i=1}^{m} \sigma_i\, f_i = 0 \text{ and } \forall_{i=1}^{m-1} e_i \Leftrightarrow e_{i+1} = 0\}$$

and for an **F**-junction

$$\{(e_1(t), f_1(t), \ldots, e_m(t), f_m(t)) \,|\, \sum_{i=1}^{m} \sigma_i\, e_i = 0 \text{ and } \forall_{i=1}^{m-1} f_i \Leftrightarrow f_{i+1} = 0\}$$

where $\sigma_i \in \{\Leftrightarrow 1, 1\}$ represents the relative direction of bond i. If bond i is directed *towards* the junction, then $\sigma_i = 1$, otherwise $\sigma_i = \Leftrightarrow 1$.

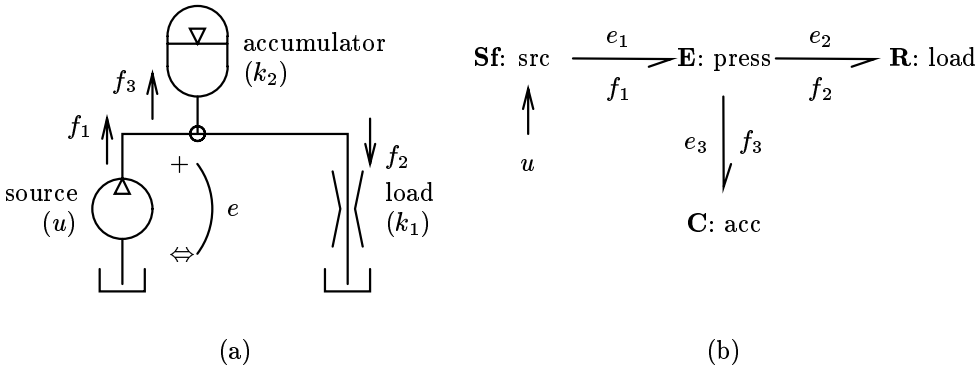


Figure 5. (a) A simple hydraulic circuit. (b) A first bond graph model of the circuit. Note that the system pressure $e = e_1 = e_2 = e_3$ is the same for all three apparatuses.

2.4. Model Composition

The composition of primitive ideals to form models of real systems is obtained after a series of steps. Here we simply outline the modelling process by providing a simple example.

EXAMPLE: Consider the hydraulic system as represented by the schematic drawing in Figure 5 (a). The flow source provides a flow f_1 proportional to the control signal u, the hydraulic load is essentially dissipative and the accumulator nearly loss-free. The interaction between the source, the accumulator and the load is such that the pressures e_1, e_2, e_3 at the ports of the mechanisms are all the same. Consequently, Figure 5 (b) depicts a first potential bond graph model of the system.

Combining the constitutive relations of the individual mechanisms, we get the following set of equations (omitting t for clarity):

$$\begin{aligned} u \Leftrightarrow f_1 &= 0 \\ k_1\, e_2 \Leftrightarrow f_2 &= 0 \\ \dot{x} \Leftrightarrow f_3 &= 0 \\ e_3 \Leftrightarrow k_2\, x &= 0 \\ f_1 \Leftrightarrow f_2 \Leftrightarrow f_3 &= 0 \\ e_1 \Leftrightarrow e_2 &= 0 \\ e_2 \Leftrightarrow e_3 &= 0 \end{aligned}$$

This set of equations uniquely defines the behaviour of the system once the initial value $x(t_0)$ is known. □

2.5. Causality in Bond Graphs

We have now seen how a given bond graph and a set of constitutive relations maps to a mathematical model of the underlying system. We have already observed that such a model is a non-minimal set of equations which can in general be difficult to solve explicitly. A preferred alternative is a sequence of directed assignment statements such that unknowns can be immediately and sequentially computed from the knowns on the right hand side. Such a model is sometimes referred to as a *computational model*.

Such a causal computational model requires the model variables to be ordered in a specific *cause-effect* relationship. Bond graph theory comes with algorithms for assigning causality to a given graph [36]. The choice of algorithm depends on the specific form of computational model required as well as on the complexity of the bond graph itself. There are graph oriented algorithms suitable for manual calculations as well as algorithms better suited for machine implementation. One classical algorithm is the so called *Sequential Causality Assignment Procedure* (SCAP) which is used to derive classical state space forms consisting of ordinary differential equations only.

EXAMPLE: Applying SCAP on the model in Section 2.4 we obtain

$$\dot{x} = \Leftrightarrow k_1 k_2\, x + u$$

□

The above equation is the basis for the physical model of the landing gear appearing in our application.

2.6. The Switched Extension

Classical bond graphs only allow for finite flows of energy and are thus restricted to the modelling of continuous change over time only. However, many practical apparatuses undergo abrupt changes in their behaviour, usually due to external or internal discrete control. Examples of such apparatuses are diodes, check valves, relays and devices with end-stops. Systems consisting of at least one such apparatus will be referred to as *mode-switching* systems.

Switched bond graphs were introduced to extend bond graphs to mode-switching physical system while maintaining the classical bond graph theory. The main ingredient in the extension is a new ideal degenerate element, namely the *ideal generalised switch* (**Sw**). As opposed to the other bond graph elements, the **Sw**-element is *not* associated with a constitutive mathematical relation over effort and flow. Instead, it is associated with a discrete mathematical structure determining the Boolean state of the switch. The two states of the **Sw**-element are referred to as the *effort* and the *flow* state. The reason for these terms is the following: In state effort the **Sw**-element is equivalent with an **Se**-element for which $u = 0$ for all t, and in state flow it is equivalent with an **Sf**-element for which $u = 0$. Given a switched

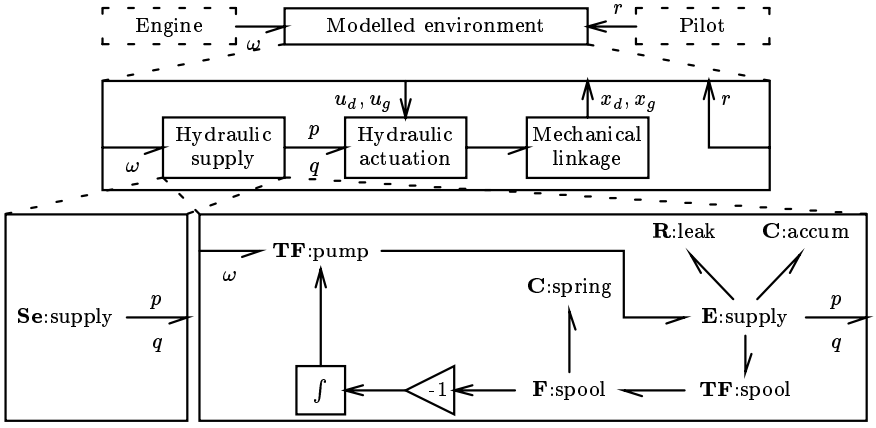


Figure 6. The physical environment models. Left: the coarsest model of the hydraulic supply. Right: the model including a hydromechanically regulated pump.

bond graph with p **Sw**-elements, once all the elements have been mathematically characterised, we get a computational hybrid system with $\leq 2^p$ sets of DAE, and a set of Boolean transition conditions. In this paper we do not elaborate further on the switched versions of the landing gear models. The interested reader is referred to [33, 30, 31].

2.7. The Landing Gear Models

The hardware components in the landing gear system consist of the landing gear itself, i.e. the wheel–and–suspension system, a pair of doors protecting the gear during flight and landing, and a pair of hydraulic actuators for the manouvering of the gear and the doors (Figure 1). As part of the landing gear system, the hydraulic power supply system provides the hydromechanic energy needed to manoeuvre the gear and the doors under all operating conditions.

For the purpose of illustration, we present two different bond graph models of the hydraulic power supply subsystem only and a single model of the door–and–gear subsystem. These models differ in the assumptions made about the subsystems, and therefore illustrate the capability of bond graphs to make such assumptions clear.

In the coarsest model of the hydraulic power supply we represent the assumption that the hydraulic power supply system behaves as an ideal constant pressure source; see the lowest left-most box in Figure 6 where p and q stand for hydraulic pressure and flow respectively. Combining it with the model of the door–and–gear subsystem, the overall plant model converts to a simple time-linear dynamic system

$$\begin{aligned} \dot{x}_d &= \alpha_d \, u_d, \quad x_d \in [0,1] \\ \dot{x}_g &= \alpha_g \, u_g, \quad x_g \in [0,1] \end{aligned}$$

where $\alpha_d, \alpha_g > 0$ are constants (time-invariants), x_d (x_g) the normalised door (gear) position and u_d (u_g) $\in \{\Leftrightarrow 1, 0, +1\}$ is the three-valued door (gear) control signal.

In the more detailed model of the power supply system (lowest right-most box in Figure 6), we no longer assume that the hydraulic pressure is ideal. Instead we make an explicit model of the hydromechanic regulator which in fact attempts to make the pressure 'as constant as possible' given a particular engine velocity ω. In this case the overall bond graph converts to a non-linear dynamic system

$$\begin{aligned} \dot{x}_d &= \alpha'_d \, p \, u_d, \quad x_d \in [0,1] \\ \dot{x}_g &= \alpha'_g \, p \, u_g, \quad x_g \in [0,1] \\ \dot{p} &= \Leftrightarrow(\gamma + \gamma_d \, |u_d| + \gamma_g \, |u_g|) \, p + \beta \end{aligned}$$

where $\alpha'_d, \alpha'_g, \gamma_d, \gamma_g > 0$ are constants. The quotient β/γ is a monotonic function of the aircraft engine velocity ω.

These models are only a small selection of models we have derived. Other models consisting of some fifty bond graph elements have been efficiently developed in cooperation with hardware experts from Saab Aerospace and the Department of Fluid Power at Linköping University. The efficiency of this iterative modelling process has significantly gained from the use of bond graphs. By means of the bond graphs we were able to communicate our models with all engineers from the industrial partners and were able to sort out misunderstandings and misconceptions at an early stage of the modelling process.

The DAE models derived can now be plugged into the model of the closed loop system as required. To do this we use a hybrid modelling architecture [17] described in the next section.

3. Framework for Iterative Modelling

This section describes a framework for putting together the environment models derived earlier with mathematical models for the other parts of the system. It can be used for refining the overall architecture in Figure 2 by making the interface between the controller (diagnoser) and the physical world more explicit.

The generic architecture as depicted in Figure 7, accommodates both the mathematical models for the physical environment and the mathematical models for the discrete controllers and supervisory systems. It has been adopted from the multi-layer architecture for hierarchical control suggested earlier [17]. This architectural decomposition is used as a framework for verification and makes the interfaces (as dictated by the choice of sensors and actuators) explicit.

The discrete controller consists of a *selector* asynchronously reacting on discrete events e detected by the *characterizer*. The characterizer generates these using a classification function over the real valued environment variables z. The output of the selector is the discrete choices c of control algorithms implemented by the *effector*. The *environment* is driven by the real valued control variables u. The unpredictability of the environment is made explicit by the disturbance variable v. This variable is typically used for allowing uncertainties in sensor measurements.

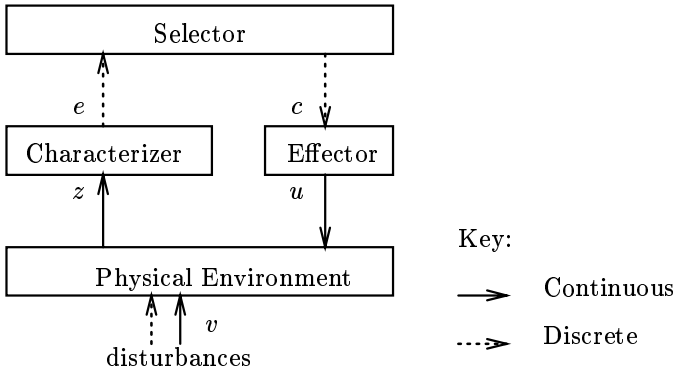


Figure 7. A generic architecture for decomposition of hybrid systems

In addition, it may be used to model major elements of the environment not under control.

This architecture has been used in a number of different applications and aids the process of modelling the closed loop system for the purpose of verification. In the landing gear study we have used the architecture as a framework for iterative modelling of the closed loop system. It was thus used to plug in different models of the environment while keeping the same selector, characterizer and effector; or different selectors while keeping the same environment model. Thus, we have been able to study and illustrate different verification techniques depending on the complexity (granularity) of the closed loop model and the property to be verified.

In the following section we will show the details of two alternative landing gear models by filling the boxes in the above architecture.

4. Hybrid Mathematical Models

In this section we describe how the models for the closed loop system can be derived using the mathematical models for the hardware derived earlier. Here, we concentrate on a static selector as depicted in Figure 8. This model can be a representation of a hard-wired electronics implementation of the controller, or alternatively, a model obtained from the compilation of a synchronous software model, e.g. in Esterel or Lustre [9]. Analysis of similar properties in presence of a dynamic selector with communication and computation delays can be found elsewhere [21, 22].

Next we will make the mappings in the interface between the controller and its environment (i.e. the characterizer and the effector) explicit.

As it can be seen in Figure 9, the inputs to the static selector are discrete states *open, closed, in, out, cmd.* Changes in these states are determined by the characterizer based on the sensed values of the door and gear position and the current pilot command. Note also that keeping the same selector and changing the environment

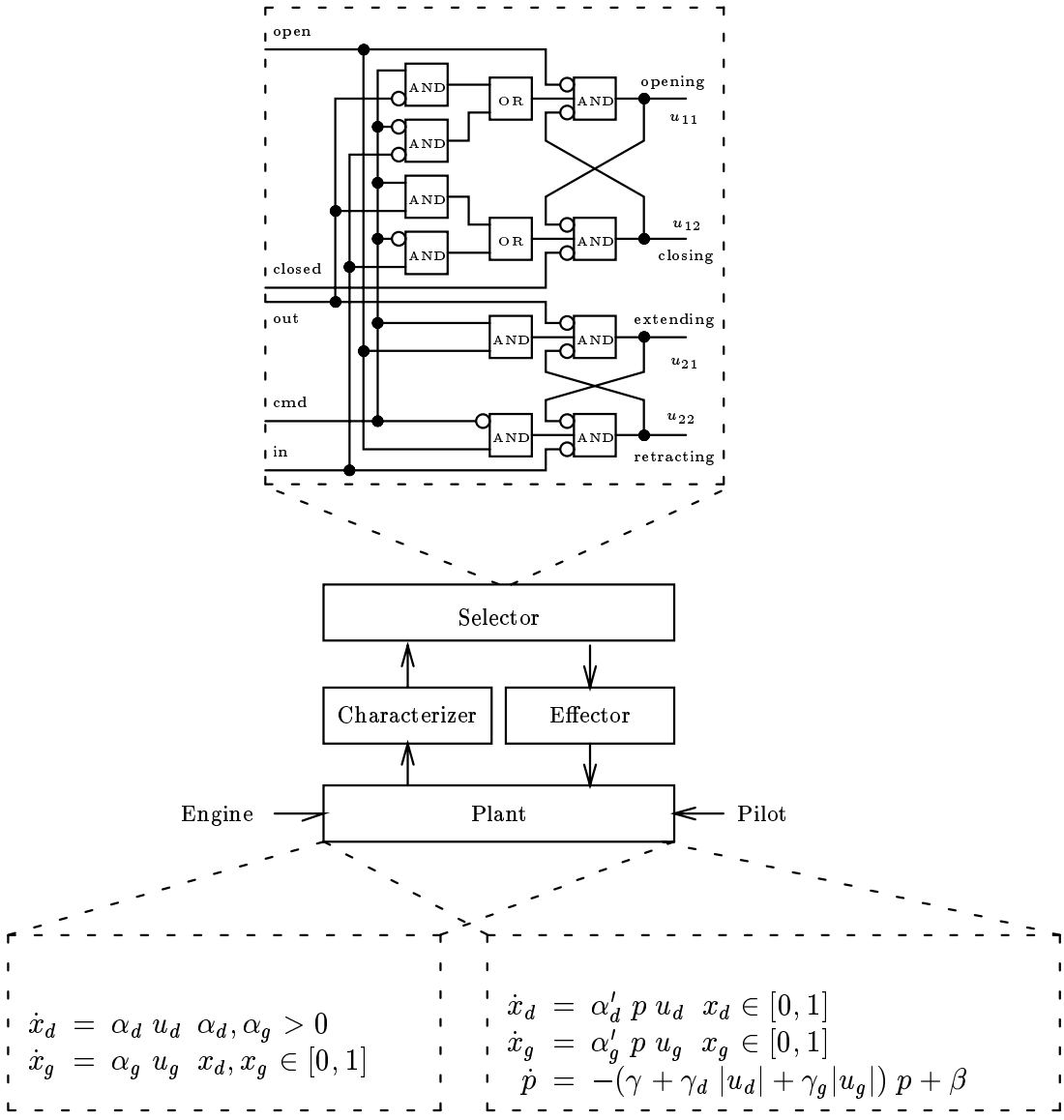


Figure 8. A static selector operating in two alternative physical environments.

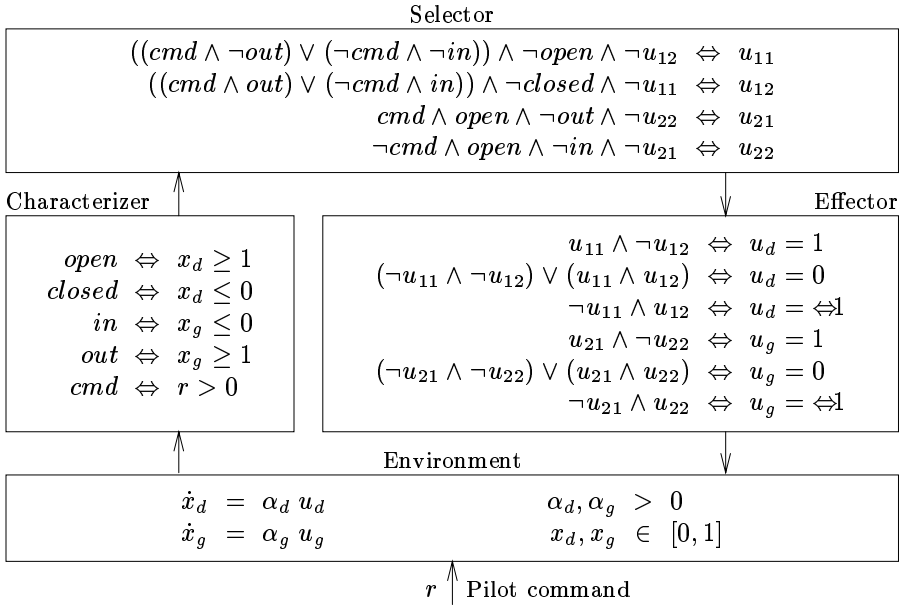


Figure 9. The system built up from the time-linear environment model and the static selector.

model to a more realistic one is simply achieved by plugging in the finer non-linear model in the architectural decomposition.

To analyse the model first we need to represent the closed loop model and the required properties in an extension of the interval temporal logic *Duration Calculus* (DC). Then verifying that the requirements are met by the given model is achieved through proving theorems in the proof system of the logic.

4.1. Brief Introduction to EDC

We use the extension of Duration Calculus in accordance with [27] in which a mathematical theory about state functions is assumed as given. The logical foundations of DC can be found in [11]. Here we give a brief exposition of the extended language. The syntactic constituents of EDC are *state names* denoting functions on time (in our case real valued functions $x_d, x_g, \dot{x}_d, \dot{x}_g$, etc), including *Boolean state names* denoting a Boolean valued function on time (in our case *open*, *closed*, etc). *State expressions* and *state assertions* are built from variables, functions and relations on real numbers, and propositional logic operators in a standard form. In our example we have $u_d\alpha_d$ as an state expression, and $x_g \leq 1$ as well as $P \wedge C$ as state assertions.

State expressions and assertions are evaluated in accordance with interpretations of states and operators. Assertions evaluate to 0 or 1 in a given interpretation. In

particular, *true* and $false$ as state assertions evaluate to 1 and 0 respectively at all times. *Durations* are formed from state expressions and assertions, using the symbol $\int$. Thus, for an assertion P, $\int P$ denotes the duration of P holding over an interval. Its semantics with respect to interpretation $\mathcal{I}$ and interval $[b, e]$ is defined by the integral $\int_b^e \mathcal{I}(P)dt$, where $\mathcal{I}$ assigns values of the right type to state names, and defines the (non standard) operator symbols of the language.

The *length* ℓ of an interval, is defined by the duration of *true* holding over the interval, expressed as $\ell \widehat{=} \int true$. Thus, the semantics for ℓ, in any interpretation $\mathcal{I}$ over any interval $[b, e]$, is defined by $\mathcal{I}(\ell)[b, e] = e \Leftrightarrow b$. Furthermore, for Q being an assertion, the notion of Q holding over an interval (represented by $\lceil Q \rceil$) is related to durations by the following definition:

$$\lceil Q \rceil \widehat{=} (\int Q = \ell) \wedge (\ell > 0)$$

Atomic duration formulas have the form $\lceil Q \rceil$ or $\triangleleft(dterm_1, \ldots, dterm_n)$ where $\triangleleft$ is an n-ary relation on reals and $dterm_i$ are duration terms. Duration terms essentially have the form $\int se$ (for state expression se), $\mathbf{b}.se$ and $\mathbf{e}.se$ denoting the value of the state expression se at the beginning and end of an interval respectively, and terms built with operators on reals. Compound duration formulas are built from atomic formulas using the usual logical connectives as well as the chop connective of interval temporal logic (denoted by ;) which chops an interval in two parts.

The truth of a formula D, determined over an interval $[b, e]$ in an interpretation $\mathcal{I}$, is denoted $\mathcal{V}_\mathcal{I}, [b, e] \models D$. Specifically, the semantics of formulas built using the chop operator is defined as follows. For duration formulas D_1 and D_2, we have:

$$\mathcal{V}_\mathcal{I}, [b, e] \models D_1; D_2 \text{ iff } \mathcal{V}_\mathcal{I}, [b, m] \models D_1 \text{ and } \mathcal{V}_\mathcal{I}, [m, e] \models D_2 \text{ for some } m \in [b, e].$$

A formula D is satisfiable in an interpretation $\mathcal{I}$ (written $\mathcal{V}_\mathcal{I} \models D$) iff $\mathcal{V}_\mathcal{I}, [0, t] \models D$ for all $t \in \mathbb{R}^{\geq 0}$. A formula D is valid (written $\models D$) iff $\mathcal{V}_\mathcal{I} \models D$ for every interpretation $\mathcal{I}$. The following abbreviations are introduced at the formula level:

$\lceil\ \rceil$	$\widehat{=}$	$\ell = 0$	the empty interval
$\Diamond D$	$\widehat{=}$	$true; D; true$	D holds in some sub-interval
$\Box D$	$\widehat{=}$	$\neg\Diamond\neg D$	D holds in every sub-interval

To avoid excessive use of parentheses, precedence rules for the operators in the language are provided [27] – e.g. $\neg$, $\Box$ and $\Diamond$ bind stronger than chop (;) which binds stronger than the other logical operators.

Next we present a number of proof rules (laws) from the proof system of EDC [27] which will be used in the proofs later on in the paper.

P-And:

$$\lceil P \wedge Q \rceil \Leftrightarrow \lceil P \rceil \wedge \lceil Q \rceil$$

P-Always:

$$\lceil P \rceil \Rightarrow \Box\,(\lceil P \rceil \vee \lceil\ \rceil)$$

Somewhere-Neg:

$$\neg \Diamond D \Leftrightarrow \Box \neg D \ \text{ and } \ \Diamond \neg D \Leftrightarrow \neg \Box D$$

Always-intro:

$$\Box D \wedge (D_1\ ;\ D2) \Rightarrow (\Box D \wedge D_1)\ ;\ (\Box D \wedge D2)$$

Always-Once-Somewhere:

$$\Box D \Rightarrow D \ \text{ and } \ D \Rightarrow \Diamond D$$

Zero:

$$\int false = 0$$

Chop-false:

$$D\ ;\ false \Rightarrow false \ \text{ and } \ false\ ;\ D \Rightarrow false$$

Chop-P-Or:

$$\lceil P_1 \vee P_2 \rceil\ ;\ true \Leftrightarrow \lceil P_1 \rceil\ ;\ true \vee \lceil P_2 \rceil\ ;\ true$$

Dur-Chop:

$$(\textstyle\int P = r_1)\ ;\ (\int P = r_2) \Leftrightarrow \int P = (r_1 + r_2)$$

Chop-Exists: *provided v does not occur free in D_1*

$$D_1\ ;\ (\exists v : T \bullet\ D_2) \Leftrightarrow \exists v : T \bullet D_1\ ;\ D_2$$

Continuity:

$$(\mathbf{e}.se = v_1)\ ;\ (\mathbf{b}.se = v_2) \Rightarrow (v_1 = v_2)$$

Note that the last law applies only to continuous state expressions. In the proofs which follow we sometimes use **analysis** where the current line can be motivated by the immediately preceding line and real arithmetic. As well as the above rules, the proofs may use another motivation denoted by **PL**, whenever the preceding line in the proof leads to the current line according to the rules in predicate logic. We further include a general EDC result which is used in later proofs.

LEMMA 1 *For duration formulas D_1 and D_2 and state assertion P*

$$\lceil P \rceil \wedge (D_2\ ;\ D_2) \Rightarrow ((\lceil P \rceil \vee \lceil\ \rceil) \wedge D_1)\ ;\ ((\lceil P \rceil \vee \lceil\ \rceil) \wedge D_2)$$

Table 1. Declaration of states and global variables.

Name: Signature	*Type: Description*
$x_d : \mathbb{R} \to \mathbb{R}$	Plant state: door position (closed: 0, opened: 1)
$x_g : \mathbb{R} \to \mathbb{R}$	Plant state: gear position (retracted: 0, extended: 1)
$\alpha_d : \mathbb{R}$	Plant parameter: door speed
$\alpha_g : \mathbb{R}$	Plant parameter: gear speed
$u_{11} : \mathbb{R} \to \{0,1\}$	Controller command: open door (hold: 0, open: 1)
$u_{12} : \mathbb{R} \to \{0,1\}$	Controller command: close door (hold: 0, close: 1)
$u_{21} : \mathbb{R} \to \{0,1\}$	Controller command: extend gear (hold: 0, extend: 1)
$u_{22} : \mathbb{R} \to \{0,1\}$	Controller command: retract gear (hold: 0, retract: 1)
$r : \mathbb{R} \to \{0,1\}$	Pilot command: extend gear (retract: 0, extend: 1)

Proof:

$$
\begin{array}{ll}
 & \lceil P \rceil \wedge (D_1 \,;\, D_2) \\
\Rightarrow & \{\text{P-Always}\} \\
 & \Box\,(\lceil P \rceil \vee \lceil\, \rceil) \wedge (D_1 \,;\, D_2) \\
\Rightarrow & \{\text{Always-intro}\} \\
 & (\Box\,(\lceil P \rceil \vee \lceil\, \rceil) \wedge D_1) \,;\, (\Box\,(\lceil P \rceil \vee \lceil\, \rceil) \wedge D_2) \\
\Rightarrow & \{\text{Always-Once-Somewhere}\} \\
 & ((\lceil P \rceil \vee \lceil\, \rceil) \wedge D_1) \,;\, ((\lceil P \rceil \vee \lceil\, \rceil) \wedge D_2)
\end{array}
$$

■

4.2. EDC Model of the Landing Gear

Table 1 gives the declaration of the constituents in the coarse plant model of the landing gear, the only variables determined externally being the reference pilot signal (r).

Using the above notation and the architectural breakdown in Figure 9, the closed loop system for the landing gear is represented by

$$S \equiv P \wedge C$$

where the plant model

$$P \equiv Env \wedge Eff$$

and the controller model

$$C \equiv Char \wedge Sel$$

Here Env, $Char$, Sel, and Eff are assertions within the underlying mathematical theory and defined in accordance with the formulas provided in the appropriate boxes in Figure 9. Then the behaviour of the system over an interval of time can be represented by lifted duration formulas over these assertions.

Note that the composition of the environment and effector model leads to the following two sets of constraints on the door and the gear parts of the model respectively. That is, $\lceil P \rceil \Rightarrow \lceil P_1 \rceil \wedge \lceil P_2 \rceil$.

$$P_1 \;\widehat{=}\; \left\{ \begin{array}{rcl} (u_{11} \wedge \neg u_{12} & \Rightarrow & \dot{x}_d = \alpha_d) \wedge \\ ((\neg u_{11} \wedge \neg u_{12}) \vee (u_{11} \wedge u_{12}) & \Rightarrow & \dot{x}_d = 0) \wedge \\ (\neg u_{11} \wedge u_{12} & \Rightarrow & \dot{x}_d = \Leftrightarrow \alpha_d) \\ 0 \leq x_d \leq 1 & & \\ 0 \leq \alpha_d \leq 1 & & \end{array} \right. \tag{1}$$

$$P_2 \;\widehat{=}\; \left\{ \begin{array}{rcl} (u_{21} \wedge \neg u_{22} & \Rightarrow & \dot{x}_g = \alpha_g) \wedge \\ ((\neg u_{21} \wedge \neg u_{22}) \vee (u_{21} \wedge u_{22}) & \Rightarrow & \dot{x}_g = 0) \wedge \\ (\neg u_{21} \wedge u_{22} & \Rightarrow & \dot{x}_g = \Leftrightarrow \alpha_g) \\ 0 \leq x_g \leq 1 & & \\ 0 \leq \alpha_g \leq 1 & & \end{array} \right. \tag{2}$$

Likewise, the selector and characterizer models can be combined to the give the following EDC formulation:

$$C \widehat{=} \tag{3}$$

$$\left\{ \begin{array}{lcl} (r > 0 \wedge \neg(x_g \geq 1)) \vee (r \leq 0 \wedge \neg(x_g \leq 0)) \wedge \neg(x_d \geq 1) \wedge \neg u_{12} & \Leftrightarrow & u_{11} \\ ((r > 0 \wedge (x_g \geq 1)) \vee (r \leq 0 \wedge (x_g \leq 0))) \wedge \neg(x_d \leq 0) \wedge \neg u_{11} & \Leftrightarrow & u_{12} \\ r > 0 \wedge (x_d \geq 1) \wedge \neg(x_g \geq 1) \wedge \neg u_{22} & \Leftrightarrow & u_{21} \\ r \leq 0 \wedge (x_d \geq 1) \wedge \neg(x_g \leq 0) \wedge \neg u_{21} & \Leftrightarrow & u_{22} \end{array} \right.$$

Given the static controller model, for example, it is possible to verify

$$\lceil C \rceil \Rightarrow \lceil \neg(u_{11} \wedge u_{12}) \rceil \tag{4}$$

$$\lceil C \rceil \Rightarrow \lceil \neg(u_{21} \wedge u_{22}) \rceil \tag{5}$$

i.e. open and close (extend and retract) commands can never be issued simultaneously. Also, the overall system model can be summarised by the following invariant

$$\Box\, (\neg \lceil \neg S \rceil \wedge \mathit{continuous}(x_d) \wedge \mathit{continuous}(x_g)).$$

4.3. Requirement Specifications in EDC

We now present the EDC formulations of the three closed loop system properties which were presented in the introduction. First we consider the safety property that the door and gear do not collide in operation. The property is proved by proving the stronger property that the gear must never move when the door is not fully open. This requirement R_1 may be formalised as

$$R_1 \equiv \Box\neg\lceil\dot{x}_g \neq 0 \wedge x_d < 1\rceil$$

The timeliness properties require that the extension (or retraction) to be completed within T time units. These requirements may now be formalised by the two formulas R_2 and R_3.

$$R_2 \equiv \Box\neg((\lceil r > 0\rceil \wedge \ell = T)\,;\,\lceil\neg(x_g \geq 1 \wedge x_d \leq 0)\rceil)$$
$$R_3 \equiv \Box\neg((\lceil r \leq 0\rceil \wedge \ell = T)\,;\,\lceil\neg(x_g \leq 0 \wedge x_d \leq 0)\rceil)$$

In other words, it is never the case that the landing (no landing) command is in operation for an interval of length T and the desired effect has not been achieved immediately after the interval. In order to verify these properties we thus have to prove

$$\lceil S\rceil \Rightarrow R_1 \wedge R_2 \wedge R_3$$

5. Verification of the Requirements

In what follows we give the proofs that the closed loop system satisfies requirements R_1 and R_2. The proof for R_3 is analogous to the one for R_2 and is therefore omitted. First we show how proofs in EDC can be used to verify the design in presence of the coarse environment model. Next, in section 6, we show another proof for the closed loop model, now including the non-linear environment model instead. This is carried out by just adding three extra lemmas and preserving the proof structure for the simpler environment model.

5.1. Verification of $\boldsymbol{R_1}$

In order to verify the first requirement we employ proof by contradiction within the proof system of EDC [27]. But before that we need to state the following lemma which gives a property of the application model as presented in 4.2.

LEMMA 2 $\lceil P \wedge C\rceil \wedge \lceil\dot{x}_g \neq 0 \wedge x_d < 1\rceil \Rightarrow false$

Proof:

$\lceil P \wedge C \rceil \wedge \lceil \dot{x}_g \neq 0 \wedge x_d < 1 \rceil$
$\Rightarrow$ {P-And}
$\lceil P \wedge C \wedge \dot{x}_g \neq 0 \wedge x_d < 1 \rceil$
$\Rightarrow$ {2, PL}
$\lceil C \wedge \neg((\neg u_{21} \wedge \neg u_{22}) \vee (u_{21} \wedge u_{22})) \wedge x_d < 1 \rceil$
$\Rightarrow$ {PL}
$\lceil C \wedge ((u_{21} \wedge \neg u_{22}) \vee (\neg u_{21} \wedge u_{22})) \wedge x_d < 1 \rceil$
$\Rightarrow$ {3, PL}
$\lceil ((x_d \geq 1) \vee (x_d \geq 1)) \wedge x_d < 1 \rceil$
$\Rightarrow$ {PL}
$\lceil false \rceil$
$\Rightarrow$ {Defn}
$\int false = \ell > 0$
$\Rightarrow$ {Zero}
$0 = \ell > 0$
$\Rightarrow$ {analysis}
$false$

■

Now we can prove the safety property.

THEOREM 1 $\quad \lceil S \rceil \wedge \neg R_1 \Rightarrow false$

Proof:

$\lceil P \wedge C \rceil \wedge \neg \Box \neg \lceil \dot{x}_g \neq 0 \wedge x_d < 1 \rceil$
$\Rightarrow$ {Somewhere-Neg, PL}
$\lceil P \wedge C \rceil \wedge \Diamond \lceil \dot{x}_g \neq 0 \wedge x_d < 1 \rceil$
$\Rightarrow$ {Defn}
$\lceil P \wedge C \rceil \wedge true \,;\, \lceil \dot{x}_g \neq 0 \wedge x_d < 1 \rceil \,;\, true$
$\Rightarrow$ {Lemma 1}
$((\lceil\ \rceil \vee \lceil P \wedge C \rceil) \wedge true) \,;\, ((\lceil\ \rceil \vee \lceil P \wedge C \rceil) \wedge \lceil \dot{x}_g \neq 0 \wedge x_d < 1 \rceil) \,;$
$((\lceil\ \rceil \vee \lceil P \wedge C \rceil) \wedge true)$
$\Rightarrow$ {Lemma 2, PL}
$(\lceil\ \rceil \vee \lceil P \wedge C \rceil) \,;\, (\lceil\ \rceil \wedge \lceil \dot{x}_g \neq 0 \wedge x_d < 1 \rceil) \,;\, (\lceil\ \rceil \vee \lceil P \wedge C \rceil)$
$\Rightarrow$ {Defn}
$(\lceil\ \rceil \vee \lceil P \wedge C \rceil) \,;\, false \,;\, (\lceil\ \rceil \vee \lceil P \wedge C \rceil)$
$\Rightarrow$ {Chop-false}
$false$

■

5.2. *Verification of* $\boldsymbol{R_2}$

To prove the timeliness property we first need to make the structure of the application more explicit. This conceptualisation then provides the necessary structure over which a case-based proof can be carried out.

5.2.1. Structure of Closed Loop System First, going back to our closed loop model as depicted in Figure 9 and the mathematical formulation of the controller in equation (3) we note that the controller represents the combination of a static selector and a static characterizer, whereas the plant equations (1), (2) can be seen as a combination of the static effector and the dynamic environment model.

Using hybrid transition systems [18, 20, 19] for modelling the environment, phases of continuous activity are interleaved with discrete mode transitions. The system stays in a mode where its behaviour is defined by a set of DAE until the guard of some transition out of the mode is true, whereby it immediately moves to the next mode as defined by a different set of DAE. This compares with a hybrid automaton in which taking an enabled discrete transition is not optional but progress is enforced.[2].

Using different expertise for deriving the models necessitates that the model of a controller and its environment are modularly developed. With no knowledge about the controller, the model of the environment could be represented by a hybrid transition system with two differential equations in each mode. The derivatives of the door and gear positions ($\dot{x}_i$) could then take any of the values $(\Leftrightarrow\alpha_i, 0, \alpha_i)$ for $i \in \{d, g\}$ in different modes. The system would have 9 modes and would change its mode depending on various choices of effector output. There would therefore be 8 transitions into each mode from all the other 8 modes, and 8 transitions out of each mode, each going to one of the other 8 modes. A parallel composition of the plant and controller would then give us a hybrid transition system in which the reachability of some modes would be constrained. In what follows we provide the structure of the closed loop system after application-based simplification (minimisation) of the hybrid transition system[3].

The following properties of the closed loop model gives us the underlying structure. First, not all 9 combinations of effector output are allowed by the combination of the effector and the selector. In other words, the composition of the characterizer, selector and effector mappings is not surjective. To see this note that if the effector is seen as a function

$$\mathcal{E} : \mathbb{B}^4 \rightarrow \{\langle u_d, u_g\rangle \mid u_d \in \{\Leftrightarrow 1, 0, 1\} \text{ and } u_g \in \{\Leftrightarrow 1, 0, 1\}\}$$

then it is surjective. However, the selector mapping $\mathcal{S} : \mathbb{B}^5 \rightarrow \mathbb{B}^4$ is not surjective. More specifically, some combinations of $\langle u_{11}, u_{12}, u_{21}, u_{22}\rangle$ are not in the range of the selector function. For example, u_{11} and u_{12} are not allowed to be true at the same time (see equation 4). Also, if the landing command is active, then based on the selector equations it is possible to eliminate some other combinations of u_{ij}. Lemma 4 formalises this property of the selector-effector composition. The allowed

values of u_{ij}, however, depend on a partitioning of the state space given below. Note that state regions S_i provide a covering of the state space in presence of the landing command.

LEMMA 3 *Let S_i be defined according to the following combinations of state variables and pilot command:*

$$\begin{aligned}
S_1 &\equiv (((0 < x_d < 1) \wedge (0 < x_g < 1)) \vee \\
&\quad ((x_d \leq 0) \wedge (0 < x_g < 1)) \vee \\
&\quad ((x_d \leq 0) \wedge (x_g \leq 0)) \vee \\
&\quad ((0 < x_d < 1) \wedge (x_g \leq 0))) \wedge r > 0 \\
S_2 &\equiv (((x_d \geq 1) \wedge (0 < x_g < 1)) \vee \\
&\quad ((x_d \geq 1) \wedge (x_g \leq 0))) \wedge r > 0 \\
S_3 &\equiv (((0 < x_d < 1) \wedge (x_g \geq 1)) \vee \\
&\quad ((x_d \geq 1) \wedge (x_g \geq 1))) \wedge r > 0 \\
S_4 &\equiv (x_d \leq 0) \wedge (x_g \geq 1) \wedge r > 0
\end{aligned}$$

Then we have

$$\lceil P \rceil \wedge \lceil r > 0 \rceil \Rightarrow \lceil S_1 \vee S_2 \vee S_3 \vee S_4 \rceil$$

LEMMA 4 *Let S_i be defined according to Lemma 3. Then the following statements can be derived using propositional logic:*

$$\lceil S_1 \wedge C \rceil \Rightarrow \lceil u_{11} \wedge \neg u_{12} \wedge \neg u_{21} \wedge \neg u_{22} \rceil$$

$$\lceil S_2 \wedge C \rceil \Rightarrow \lceil \neg u_{11} \wedge \neg u_{12} \wedge u_{21} \wedge \neg u_{22} \rceil$$

$$\lceil S_3 \wedge C \rceil \Rightarrow \lceil \neg u_{11} \wedge u_{12} \wedge \neg u_{21} \wedge \neg u_{22} \rceil$$

$$\lceil S_4 \wedge C \rceil \Rightarrow \lceil \neg u_{11} \wedge \neg u_{12} \wedge \neg u_{21} \wedge \neg u_{22} \rceil$$

Note that the above result implies that while the landing command is in operation (implicit in the region conditions), no combinations of u_{ij} other than those listed above are possible. This means that in none of the given regions, no pairs of door and gear movement commands are issued simultaneously by the selector. This is a direct corollary to Lemma 3. It rests on the fact that the domain of the function composed from the characterizer and selector functions has been completely covered by the conditions on the left hand side of $\Rightarrow$ in the above formulas. Based on this result and the effector mapping described in P_1 and P_2, we can now state the following lemma, again proved using propositional logic and Lemma 4.

LEMMA 5 *Let S_i be defined according to Lemma 3. Then we have*

$$\lceil S_1 \rceil \wedge \lceil P \wedge C \rceil \Rightarrow \lceil \dot{x}_d = \alpha_d \wedge \dot{x}_g = 0 \rceil$$

$$\lceil S_2 \rceil \wedge \lceil P \wedge C \rceil \Rightarrow \lceil \dot{x}_d = 0 \wedge \dot{x}_g = \alpha_g \rceil$$

$$\lceil S_3 \rceil \wedge \lceil P \wedge C \rceil \Rightarrow \lceil \dot{x}_d = \Leftrightarrow \alpha_d \wedge \dot{x}_g = 0 \rceil$$

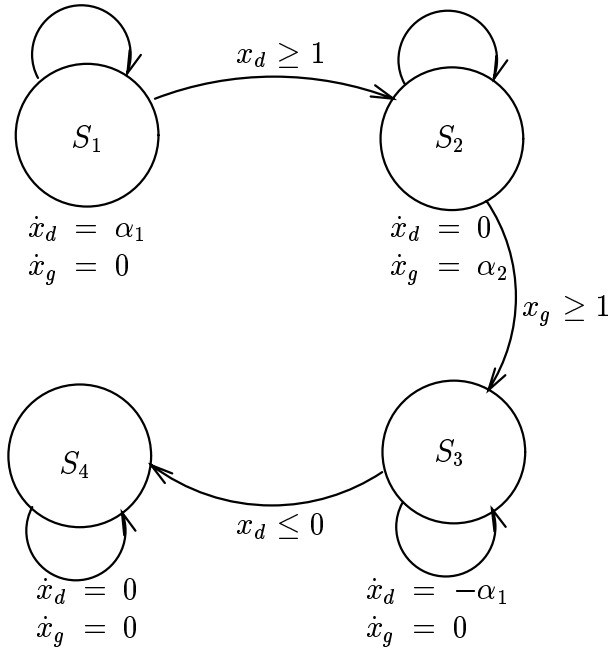


Figure 10. State space regions of the closed loop system with landing command in operation.

$$\lceil S_4 \rceil \wedge \lceil P \wedge C \rceil \Rightarrow \lceil \dot{x}_d = 0 \wedge \dot{x}_g = 0 \rceil$$

This lemma provides the necessary structure over which the proof of the timeliness property R_2 can be carried out. To make the structure of the proof for the main theorem more visible we present a fraction of a hybrid transition system which is derived for the closed loop system. With each mode in this system we associate a region in the continuous state space given that the pilot command is in operation[4].

Figure 10 associates one mode with each of the state space regions S_i in Lemma 3 above. Each mode in the graph is annotated with the corresponding door and gear equations given in Lemma 5. Note that reflexive transitions have an implicit "otherwise" condition. They have only been drawn to simplify visualisation: if the guard of the outgoing transition is not true the system remains in the current S_i region. In particular, the system will remain in the S_4 region unless the landing command is altered – a transition for moving out to the other half of the model which has not been depicted.

Based on Lemma 3 and Lemma 5 we can provide a pictorial representation of the mode changes, where the choice of arcs and their respective labels are justified by characteristics of each region S_i. This analysis is formalised in the following EDC formulation. The first formula, for example, states that if S_1 holds during a subinterval which starts an interval, then it will either continue to hold during the rest of the interval, or the subinterval will be followed by an adjacent subinterval in which S_2 holds.

LEMMA 6 *For S_i as defined in Lemma 3 we have:*

(1) $(\lceil S_1 \rceil ; true) \wedge \lceil r > 0 \rceil \Rightarrow (\lceil S_1 \rceil \vee (\lceil S_1 \rceil ; \lceil S_2 \rceil ; true))$

(2) $(\lceil S_2 \rceil ; true) \wedge \lceil r > 0 \rceil \Rightarrow (\lceil S_2 \rceil \vee (\lceil S_2 \rceil ; \lceil S_3 \rceil ; true))$

(3) $(\lceil S_3 \rceil ; true) \wedge \lceil r > 0 \rceil \Rightarrow (\lceil S_3 \rceil \vee (\lceil S_3 \rceil ; \lceil S_4 \rceil ; true))$

(4) $(\lceil S_4 \rceil ; true) \wedge \lceil r > 0 \rceil \Rightarrow \lceil S_4 \rceil$ *and* $\lceil S_4 \rceil \Rightarrow (true ; \lceil S_4 \rceil)$

Proof: By straight forward mathematical analysis, we can motivate the transition from S_1 to S_2 as follows: based on Lemma 5 the value of x_g in the region S_1 is constant. Therefore, during every interval in which the landing command remains constant ($r > 0$), if the state of the system should move out of this region, it should do so due to the value of x_d. That is, x_d should increase beyond the bounds set by the region. This will take the system to state S_2 with $x_d \geq 1$ as a condition. Similar reasoning can be performed for other transitions. ■

5.2.2. The Proof Sketch for R_2 The above structure is the basis for the following proof steps which lead to the proof of R_2. First, we show that if the system arrives at S_4 then we can be sure that the condition stipulated in the second subinterval of R_2 is not violated (i.e. the door not fully closed or the gear not fully extended) – this is shown in Lemma 7. Hence, it suffices to show that from any state, if $r > 0$ persists, then the system will reach S_4 within the time limit stipulated in the first subinterval of R_2. To begin with, we show that the duration of stay in each mode is limited according to Lemma 8. Then the main theorem shows that starting anywhere in the state space will take us to S_4 within the allowed time T *provided that* the following relation holds: $\frac{2}{\alpha_d} + \frac{1}{\alpha_g} \leq T$. Alternatively stated, we obtain these additional constraints as a by-product of the main proof. The proofs of the lemmas and the main theorem can be found in the appendix.

LEMMA 7 *Let S_4 be defined as in Lemma 3. Then we have:*

$$(\lceil P \wedge C \rceil \wedge true ; \lceil S_4 \rceil) ; \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil \Rightarrow false$$

Next, we show the constraints over the duration of stay in each of $S_1, \ldots, S_3$.

LEMMA 8 *Let S_i be defined as in Lemma 3. Then we have:*

(1) $\lceil P \wedge C \rceil \wedge \lceil S_1 \rceil \Rightarrow \ell \leq \frac{1}{\alpha_d}$

(2) $\lceil P \wedge C \rceil \wedge \lceil S_2 \rceil \Rightarrow \ell \leq \frac{1}{\alpha_g}$

(3) $\lceil P \wedge C \rceil \wedge \lceil S_3 \rceil \Rightarrow \ell \leq \frac{1}{\alpha_d}$

This will in turn imply that, under the above constraint over α_d, α_g and T, neither of these modes can last for as long as T time units.

COROLLARY 1 *Let* $\frac{1}{\alpha_d} \leq T$ *and* $\frac{1}{\alpha_g} \leq T$. *Then for* $i \in \{1,2,3\}$ *we have*

$$\lceil S_i \rceil \wedge \ell = T \Rightarrow false$$

THEOREM 2 *Let* $2/\alpha_d + 1/\alpha_g \leq T$. *Then* $\lceil S \rceil \wedge \neg R_2 \Rightarrow false$

6. Verifying the Non-linear Hybrid Model

We now present the additional proof steps necessary to show that the same properties hold in presence of the non-linear environment model. Let the environment model in Env be replaced by Env' to give the mathematical model S'. We recall the refined physical model of the hydraulic supply system (corresponding to the bond graph to the right of Figure 6) we have:

$$\begin{aligned} \dot{x}_d &= \alpha'_d \, p \, u_d \\ \dot{x}_g &= \alpha'_g \, p \, u_g \\ \dot{p} &= \Leftrightarrow(\gamma + \gamma_d \, |u_d| + \gamma_g \, |u_g|) \, p + \beta \end{aligned}$$

where $\alpha'_d, \alpha'_g, \beta, \gamma, \gamma_d, \gamma_g > 0$ are constants, $x_d, x_g \in [0,1]$ are the positions of the door and the gear actuators as before, and where p is the hydraulic supply pressure.

First we note that the proofs for Lemma 2 and Theorem 1 are not affected by the new plant model. Hence, the safety property continues to hold.

Next, we study the proof of the timeliness property R_2. The coarse model S was shown to have this property under the assumption that $2/\alpha_d + 1/\alpha_g \leq T$. We need similar assumptions in the case of S', only the difference is that due to the higher complexity in the model more insights from the physical world are needed to formulate and validate the assumptions. More specifically, the new proof requires that the values of the constants $\gamma, \gamma_d, \gamma_g, \beta$ and the pressure level p are made explicit. But before that we review the impact of the new plant model on the existing lemmas. We note that Lemma 4 still holds since it is based on the combination of the characterizer and the selector which remain unchanged. Lemma 5 will be restated as the following lemma taking account of Env' equations.

LEMMA 9 *Let* S_i *be defined according to Lemma 3. Let* $P' \mathrel{\widehat{=}} Env' \wedge Eff$. *Then we have*

$$\lceil S_1 \rceil \wedge \lceil P' \wedge C \rceil \Rightarrow \lceil \dot{x_d} = \alpha'_d p \wedge \dot{x_g} = 0 \rceil \wedge \lceil \dot{p} = \Leftrightarrow(\gamma + \gamma_d)p + \beta \rceil$$

$$\lceil S_2 \rceil \wedge \lceil P' \wedge C \rceil \Rightarrow \lceil \dot{x_d} = 0 \wedge \dot{x_g} = \alpha'_g p \rceil \wedge \lceil \dot{p} = \Leftrightarrow(\gamma + \gamma_g)p + \beta \rceil$$

$$\lceil S_3 \rceil \wedge \lceil P' \wedge C \rceil \Rightarrow \lceil \dot{x_d} = \Leftrightarrow \alpha'_d p \wedge \dot{x_g} = 0 \rceil \wedge \lceil \dot{p} = \Leftrightarrow(\gamma + \gamma_d)p + \beta \rceil$$

$$\lceil S_4 \rceil \wedge \lceil P' \wedge C \rceil \Rightarrow \lceil \dot{x_d} = 0 \wedge \dot{x_g} = 0 \rceil \wedge \lceil \dot{p} = \Leftrightarrow \gamma p + \beta \rceil$$

The above lemma can then be used to redraw Figure 10 to get a similar figure which covers the state space of S' in presence of $r > 0$. This is done by simply replacing the differential equations in Figure 10 with the equations on the right

hand side of implications in Lemma 9. Based on the above lemma, Lemma 7 will continue to hold for S'. Furthermore, the new structure and the following lemma can be used to show that the earlier result about S stated in Lemma 6 continues to hold for S'.

LEMMA 10 *Consider* $S' \equiv P' \wedge C$. *Let* $\hat{\gamma} = max\,(\gamma + \gamma_d, \gamma + \gamma_g)$. *Let* $p(0) \geq \hat{p} = \beta/\hat{\gamma}$. *Then* $p(t) \geq \hat{p} > 0$ *for all* $t \geq 0$.

Proof:
Based on physical knowledge we know that β is a positive constant dependable on the engine velocity ω (for a particular ω we get a particular β); also that γ_d and γ_g are positive. We show that if p is positive to begin with, it will remain positive under all circumstances. We consider two cases.
Case 1: $p(0) = \hat{p}$. Then $\dot{p} = 0$ and hence $p = \hat{p}$ at all times. This case reduces the problem to the coarse model analysed earlier.
Case 2: $p(0) > \hat{p}$. Then $\dot{p} < 0$ and hence p will eventually reach $\hat{p}$, in which case the argument under case 1 will continue to hold.

■

Thus, $\hat{p}$ is the minimum value that p can take if the system is initialised with adequate pressure. Next we show that the positions of the door and gear will monotonically increase (decrease) in every S_i region except for S_4.

LEMMA 11 *For any interval in which* u_d *and* u_g *have fixed values,* p *is monotonic in* t.

Proof: Follows from Lemma 9 and Lemma 10. ■

It is therefore the case that the earlier transition relation between the modes (regions) holds even in the new system S'. In other words, Lemma 6 holds also for S'. The last two results can be used to derive the upper bound for the duration of staying in each region in terms of the minimal pressure value $\hat{p}$, in a similar fashion to the non-linear case. Only the duration of stay in each mode is now also dependent on the minimal pressure $\hat{p}$. The higher the pressure the shorter time it takes for moving from an end position (for the door or gear) to another end position.

THEOREM 3 *Let* $\frac{2}{\hat{p}\alpha'_d} + \frac{1}{\hat{p}\alpha'_g} \leq T$. *Then*

$$\lceil S' \rceil \wedge \neg R_2 \Rightarrow false$$

Proof:
We follow the same proof structure as in Theorem 2 only with the initial assumption that $\frac{2}{\hat{p}\alpha'_d} + \frac{1}{\hat{p}\alpha'_g} \leq T$. In addition, we systematically replace the references to Lemma 5 and the associated right hand side differential equations, with references to Lemma 9 and the new equations from Env' as given in Lemma 9.
■

7. Concluding Remarks

We have reported here on an approach to formal verification which is initiated at the informal engineering documents. We have described the physical components, derived bond graph and mathematical models for the environment systematically, and then combined them with the mathematical model of the discrete controller, to obtain models suitable for formal verification.

We used a generic architecture to derive alternative closed loop models based on alternative environment models. These ranged from simple time-linear models to non-linear models allowing variable hydraulic pressure. We also illustrated the use of proofs in Extended Duration Calculus to ensure safety and timeliness properties of the different closed loop models.

7.1. The Insights Gained

Going back to our generic hybrid architecture, the work reported in this article illustrates a possible technique when the complexity and the dynamics of the closed loop system lies in the lower half (the environment model). Typical for these cases is that a fair amount of the formal verification is mathematical analysis over the real-valued variables. This can be contrasted with other methods illustrated in [21, 22], for cases where the complexity arises due to the size or non-deterministic timing characteristics of the upper half of the architecture (discrete controllers). This study is among the first applications of formal deductive proofs to hybrid systems with non-linear DAE.

It should be obvious from the illustration of the method that, aided by the architectural decomposition, the EDC formulations of the closed loop model were easy to obtain. Moreover, switching from one plant model to another simply amounts to changing one conjunct in the formula representing S (the closed loop system) with another (representing the refined plant model).

The EDC proofs, of course, require a general familiarity with constructing logical proofs. However, having this familiarity, the use of EDC proof system was not particularly difficult. Nevertheless, since similar proof tactics (contradiction, case analysis, etc.) tend to appear in different applications, the use of mechanised proof assistants is recommendable. Thus, the next step for making this type of analysis worthwhile in a larger example is the use of theorem provers. PC/DC, a mechanical prover for Duration Calculus implemented on top of PVS [24] has existed for some years [29]. An attempt to mechanically check some of our hand proofs was successfully carried out [23]. There is, however, more work to be done; tedious rewriting to bring the sequent in a special form can be eliminated by provision of more infrastructure in the prover. The PC/DC proof for Lemma 2 which requires 8 hand proof steps, for example, required 140 proof steps.

Another reflection over the work performed is the close connection between the region based transition system (Figure 10) and the structure of the proof. Much of the literature on formal verification assumes that the model (or axiomatisation) of the system to be analysed exists a priori. This formalisation is additionally

assumed to be in a form suitable for the application of the given proof technique. HyTech [14] could for example analyse the transition system based on the coarse model automatically, thereby eliminating the need for the EDC proof in the time-linear case. However, the exercise illustrates that the very derivation of this model may be nontrivial – or rather, once a satisfactory model is found, the proof work is more or less routine. Moreover, the insights gained while modelling can be reused for verifying later refinements of the model (e.g. for the non-linear case).

7.2. The Wider Perspective

Fundamental research in formal methods and its deployment in several real world applications has seen a major progress over the past decade. Among the notable examples are the application of formal methods and tools to the London air traffic management system [10], avionic software for the Lockheed C130J [7], the Traffic Collision avoidance system (TCAS) [13], the Motorola 68020 microcode processor [5], and the proof of correctness for an SRT division algorithm [28].

However, despite the impressive results, it is clear that the road to application of formal techniques as a routine step within the system development process is quite a long and bumpy one. To attack the problems along this road research in two directions is needed. There is a need for basic research in formal languages, data structures and algorithms for specification and reasoning about complex embedded systems. There is also a requirement to study the needs of application domain engineers, the suitable models and abstractions in different domains (e.g. telecommunications, aerospace, electronic design, etc.) and to obtain suitable interfaces accessible to non formal method experts. The key to success is to get a good synergy between both of these tracks of research.

The first phase of the reported project started along the second track. Our work confirms conclusions reported in similar projects [8], that analysis by deductive techniques requires a great deal of user knowledge and interaction. We note that this is also true with algorithmic approaches. When applying model-checking in our example, the bulk of the work was in justifying the time-linearity assumption by the physical modelling activity, and presenting the exact hardware assumptions under which this verification result makes sense.

Another approach that we studied may be labelled the 'discrete events approach', whereby the system is represented as parallel composition of a set of finite automata. Although the theory for synthesis of discrete controllers has existed for a while [26], its application to verification requires obtaining natural discrete models of the plant, and practical automatic verification tools.

Our study of the landing gear example in this framework was carried out within the development environment of Esterel/Mauto [3]. The synchronous family of languages have the benefit of a formal semantics (as synchronous I/O machines or Mealy automata) and an intuitive appeal within the engineering community – one attraction being the possibility of automatic code generation. Once the model of a controller in interaction with a plant has been formally verified for certain

properties, then the high level controller model can be automatically translated to C, Ada or a circuit design language.

Again, our application of these techniques rests on the fact that realistic discrete models of the mechanics and hydraulics could be developed based on the physically driven models.

To sum up, our experience with different modelling and verification techniques confirms that a principled derivation of the models remains a cornerstone of the verification activities. For smaller systems it is possible to manually translate between model types while preserving their essential properties; but applications in an industrial setting require more support tools for derivation and management of models.

Acknowledgments

This research was partially funded by the Swedish National Board for Industrial and Technical Development (NUTEK). Also, the authors wish to thank Göran Backlund and his associates at Saab Aerospace, and Arne Jansson at the division of Fluid Power, Department of Mechanical Engineering, Linköping University, for many valuable contributions and comments on the hydraulic and mechanical models of which only a fraction were presented here. The work by the second author was partly carried out while at the Department of Electrical Engineering at Linköping University.

Appendix

Proofs for the timeliness property

Lemma 7

Proof:

$$
\begin{array}{ll}
 & (\lceil P \wedge C \rceil \wedge \mathit{true}\ ;\ \lceil S_4 \rceil)\ ;\ \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil \\
\Rightarrow & \{\text{Lemma 1}\} \\
 & ((\lceil P \wedge C \rceil \vee \lceil\ \rceil) \wedge \mathit{true})\ ;\ ((\lceil P \wedge C \rceil \vee \lceil\ \rceil) \wedge \lceil S_4 \rceil)\ ;\ \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil \\
\Rightarrow & \{\text{PL}\} \\
 & (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ;\ ((\lceil P \wedge C \rceil \wedge \lceil S_4 \rceil) \vee (\lceil\ \rceil \wedge \lceil S_4 \rceil))\ ;\ \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil \\
\Rightarrow & \{\text{Defn, PL}\} \\
 & (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ;\ (\lceil P \wedge C \rceil \wedge \lceil S_4 \rceil)\ ;\ \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil \\
\Rightarrow & \{\text{Lemma 5}\} \\
 & (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ;\ (\lceil P \wedge C \rceil \wedge \lceil S_4 \rceil \wedge \lceil \dot{x}_d = 0 \wedge \dot{x}_g = 0 \rceil)\ ; \\
 & \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil \\
\Rightarrow & \{\text{analysis, PL}\} \\
 & (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil S_4 \rceil \wedge (\exists v_1 v_2 : \mathbb{R} \bullet \mathbf{b}.x_g = \mathbf{e}.x_g = v_1 \wedge \mathbf{b}.x_d = \mathbf{e}.x_d = v_2))\ ; \\
 & \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil \\
\Rightarrow & \{\text{Chop-Exists}\} \\
 & (\exists v_1 v_2 : \mathbb{R} \bullet (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil S_4 \rceil \wedge \mathbf{b}.x_g = \mathbf{e}.x_g = v_1 \wedge \mathbf{b}.x_d = \mathbf{e}.x_g = v_2)\ ; \\
 & \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Continuity}\} \\
 & (\exists v_1 v_2 : \mathbb{R} \bullet (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil S_4 \rceil \wedge \mathbf{b}.x_g = \mathbf{e}.x_g = v_1 \wedge \mathbf{b}.x_d = \mathbf{e}.x_d = v_2)\ ; \\
 & (\mathbf{b}.x_g = v_1 \wedge \mathbf{b}.x_d = v_2 \wedge \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil)) \\
\Rightarrow & \{\text{Lemma 3}\} \\
 & (\exists v_1 v_2 : \mathbb{R} \bullet (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ;\ (v_1 \geq 1 \wedge v_2 \leq 0)\ ; \\
 & (\mathbf{b}.x_g = v_1 \wedge \mathbf{b}.x_d = v_2 \wedge \lceil \neg(x_g \geq 1 \wedge x_d \leq 0) \rceil)) \\
\Rightarrow & \{\text{analysis}\} \\
 & (\exists v_1 v_2 : \mathbb{R} \bullet (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ;\ (v_1 \geq 1 \wedge v_2 \leq 0)\ ;\ \mathit{false}) \\
\Rightarrow & \{\text{Chop-Exists}\} \\
 & (\exists v_1 v_2 : \mathbb{R} \bullet (\lceil P \wedge C \rceil \vee \lceil\ \rceil)\ ;\ (v_1 \geq 1 \wedge v_2 \leq 0))\ ;\ \mathit{false} \\
\Rightarrow & \{\text{Chop-false}\} \\
 & \mathit{false}
\end{array}
$$

■

Lemma 8

Proof:
We show the proof for (1) which is a proof by contradiction. Proofs for (2) and (3) are analogous.

$\lceil P \wedge C \rceil \wedge \lceil S_1 \rceil \wedge \ell > \frac{1}{\alpha_d}$
$\Rightarrow$ {Lemma 3,Lemma 5, P-And}
$\lceil P \wedge C \wedge x < 1 \wedge \dot{x}_d = \alpha_d \wedge \dot{x}_g = 0 \rceil \wedge \ell > \frac{1}{\alpha_d}$
$\Rightarrow$ {Defn}
$\lceil x \geq 0 \wedge x < 1 \wedge \dot{x}_d = \alpha_d \wedge \dot{x}_g = 0 \rceil \wedge \ell > \frac{1}{\alpha_d}$
$\Rightarrow$ {analysis}
$\ell \leq \frac{1}{\alpha_d} \wedge \ell > \frac{1}{\alpha_d}$
$\Rightarrow$ {PL}
$false$

■

Theorem2

Proof:

$\lceil P \wedge C \rceil \wedge \neg \Box \neg((\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {Somewhere-Neg, PL}
$\lceil P \wedge C \rceil \wedge \Diamond((\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {Defn}
$\lceil P \wedge C \rceil \wedge (true\ ;\ (\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil\ ;\ true)$
$\Rightarrow$ {Lemma 1, PL}
$(\lceil\ \rceil \vee \lceil P \wedge C \rceil)\ ;$
$(\lceil\ \rceil \vee \lceil P \wedge C \rceil) \wedge ((\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)\ ;$
$(\lceil\ \rceil \vee \lceil P \wedge C \rceil)$

Considering the middle intervals:

$(\lceil\ \rceil \vee \lceil P \wedge C \rceil) \wedge ((\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {PL}
$(\lceil\ \rceil \wedge (\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \vee$
$(\lceil P \wedge C \rceil \wedge (\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {Defn, PL, Chop-false}
$false \vee (\lceil P \wedge C \rceil \wedge (\lceil r > 0 \rceil \wedge \ell = T)\ ;\ \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {Lemma 1}
$((\lceil\ \rceil \vee \lceil P \wedge C \rceil) \wedge \lceil r > 0 \rceil \wedge \ell = T)\ ;\ ((\lceil\ \rceil \vee \lceil P \wedge C \rceil) \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {Defn, PL}
$(\lceil P \wedge C \rceil \wedge \lceil r > 0 \rceil \wedge \ell = T)\ ;\ (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {P-And, Lemma 3}
$(\lceil P \wedge C \rceil \wedge \lceil S_1 \vee S_2 \vee S_3 \vee S_4 \rceil \wedge \lceil r > 0 \rceil \wedge \ell = T)\ ;$
$(\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$
$\Rightarrow$ {Chop-P-Or}
$(\lceil P \wedge C \rceil \wedge$
$(\lceil S_1 \rceil\ ;\ true \vee \lceil S_2 \rceil\ ;\ true \vee \lceil S_3 \rceil\ ;\ true \vee \lceil S_4 \rceil\ ;\ true) \wedge$
$\lceil r > 0 \rceil \wedge \ell = T)\ ;$
$(\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil)$

We prove each case separately. The first three cases cover the possibilities that the first subinterval is started by S_1, S_2, and S_3 respectively. Cases 2 and 3 essentially appear as a subproof in case 1, which is the longest subproof and will be shown here.
Case 1:

$$
\begin{array}{ll}
 & (\lceil P \wedge C \rceil \wedge (\lceil S_1 \rceil \; ; \mathit{true}) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 6}\} \\
 & (\lceil P \wedge C \rceil \wedge (\lceil S1 \rceil \vee (\lceil S1 \rceil \; ; \lceil S_2 \rceil \; ; \mathit{true})) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Corollary 1, PL}\} \\
 & (\lceil P \wedge C \rceil \wedge (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \mathit{true}) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 6}\} \\
 & (\lceil P \wedge C \rceil \wedge ((\lceil S_1 \rceil \; ; \lceil S_2 \rceil) \vee (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil \; ; \mathit{true})) \wedge \\
 & \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 8, PL}\} \\
 & (\lceil P \wedge C \rceil \wedge \\
 & ((\ell \leq 1/\alpha_d \; ; \ell \leq 1/\alpha_g \vee (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil \; ; \mathit{true})) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Dur-Chop, analysis}\} \\
 & (\lceil P \wedge C \rceil \wedge (\mathit{false} \vee (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil \; ; \mathit{true})) \wedge \lceil r > 0 \rceil \wedge \ell = T \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 6, PL}\} \\
 & (\lceil P \wedge C \rceil \wedge \\
 & (((\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil) \vee (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil \; ; \lceil S_4 \rceil \; ; \mathit{true})) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{analysis}\} \\
 & (\lceil P \wedge C \rceil \wedge \\
 & ((\ell \leq 1/\alpha_d \; ; \ell \leq 1/\alpha_g \; ; \ell \leq 1/\alpha_d \vee (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil \; ; \lceil S_4 \rceil \; ; \mathit{true})) \wedge \\
 & \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Dur-Chop, analysis}\} \\
 & (\lceil P \wedge C \rceil \wedge (\mathit{false} \vee (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil \; ; \lceil S_4 \rceil \; ; \mathit{true}) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 6, PL}\} \\
 & (\lceil P \wedge C \rceil \wedge (\lceil S_1 \rceil \; ; \lceil S_2 \rceil \; ; \lceil S_3 \rceil \; ; \mathit{true} \; ; \lceil S_4 \rceil) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 7, Chop-false}\} \\
 & \mathit{false}
\end{array}
$$

Next we show the case where S_4 holds at the beginning of the first subinterval. Case 4:

$$
\begin{array}{ll}
 & (\lceil P \wedge C \rceil \wedge (\lceil S_4 \rceil \; ; true) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 6}\} \\
 & (\lceil P \wedge C \rceil \wedge (true \; ; \lceil S_4 \rceil) \wedge \lceil r > 0 \rceil \wedge \ell = T) \; ; \\
 & (\lceil P \wedge C \rceil \wedge \lceil \neg(x_g \leq 1 \wedge x_d \leq 0) \rceil) \\
\Rightarrow & \{\text{Lemma 7, Chop-false}\} \\
 & false
\end{array}
$$

Based on the above case analysis and through two further applications of **Chop-false** we can conclude the proof for

$$\lceil S \rceil \wedge \neg R_2 \Rightarrow false$$

■

Notes

1. Our notion of hardware includes mechanical, electrical and hydraulic components. The paper will make some of these differences clear.
2. Note that once the guard is true, the jump is immediate in physically modelled plants, but may be subject to explicit time constraints for timed controllers.
3. In general, we need formal rewrite rules and tools to obtain the simplest form for each application, but here the model is presented in simplified form
4. A similar but slightly different fraction can also be derived for the case when $r \leq 0$ holds, in order to prove the property R_3.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Journal of Theoretical Computer Science*, 138:3–34, 1995.
2. J.J. Beaman and R.C. Rosenberg. Constitutive and modulation structure in bond graph modeling. *Journal of Dynamic Systems, Measurement and Control*, 110(4):395–402, December 1988.
3. G. Berry. The Esterel v5 Language Primer. Technical report, Ecole des Mines and INRIA, Sophia-Antipolis, `http://zenon.inria.fr/meije/esterel`, April 1997.
4. G. Berry. The Foundations of Esterel. In *Proofs, Languages and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998. To appear.
5. R. Boyer and Y. Yu. Automated Proofs of Object Code for a Widely used Microprocessor. *Journal of the ACM*, 43(1):166–192, 1996.
6. Z. Chaochen, A. P. Ravn, and M. R. Hansen. An Extended Duration Calculus for Hybrid Real-Time Systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Proc. Workshop on Theory of Hybrid Systems, October 1992, LNCS 736*, pages 36–59, Lyngby, Denmark, 1993. Springer Verlag.
7. M. Croxford and J. Sutton. Breaking through the V and V bottleneck. In *Proceedings of Ada in Europe*. Springer Verlag, 1995.
8. B. Dutertre and V. Stavridou. Formal Requirements Analysis of an Avionics Control System. *IEEE Transactions on Software Engineering*, 25(5):267–278, May 1997.

9. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.
10. A. Hall. Using Formal Methods to develop an ATC Information System. *IEEE Software*, 12(6):66–76, 1996.
11. M. R. Hansen and Z. Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, pages 283–330, 1997.
12. D. Harel. STATECHARTS: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
13. M. Heimdahl and N. Leveson. Completeness and Consistency in Heirarchical State-based Requirements. *IEEE transactions on Software Engineering*, 22(6):363–377, June 1996.
14. T.A. Henzinger and P-H. Ho. Model Checking Strategies for Linear Hybrid Systems. In *proc. of Workshop on Formalisms for Representing and Reasoning about Time, as part of the Seventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, May 1994.
15. D.C. Karnopp, R.C. Rosenberg, and D. Margolis. *System dynamics - A unified approach (2nd edition).* John Wiley & Sons, New York, 1990.
16. H. Langmaack, W.-P. de Roever, and J. Vytopil, editors. *Proc. of the 3rd. International Conference on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS 863.* Springer Verlag, 1994.
17. M. Morin, S. Nadjm-Tehrani, P. Österling, and E. Sandewall. Real-Time Hierarchical Control. *IEEE Software*, 9(5):51–57, September 1992.
18. S. Nadjm-Tehrani. *Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification.* PhD thesis, Dept. of Computer and Information Science, Linköping University, March 1994. Dissertation No. 338.
19. S. Nadjm-Tehrani. Time-Deterministic Hybrid Transition Systems. In *Hybrid Systems V, Proceedings of the fifth international workshop on hybrid systems, LNCS, To appear.* Springer Verlag, 1998.
20. S. Nadjm-Tehrani and J-E. Strömberg. From Physical modelling to Compositional models of Hybrid Systems. In Langmaack et al. [16], pages 583–604.
21. S. Nadjm-Tehrani and J-E. Strömberg. Proving Dynamic Properties in an Aerospace Application. In *Proc. of the 16th International Symposium on Real-time Systems*, pages 2–10. IEEE Computer Society Press, December 1995.
22. S. Nadjm-Tehrani and J-E. Strömberg. JAS-95 Lite: Modelling and Formal Analysis of Dynamic Properties. Technical Report LITH-IDA-R-96-41, Dept. of Computer and Information Science, Linköping University, December 1996. Currently appears on `http://www.ida.liu.se/~snt/activities.html`.
23. M. R. Nielsen. Support for Duration Calculus Verification. Master's thesis, Dept. of Information Technology, Technical University of Denmark, April 1997.
24. N. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *Proc. 11th International Conference on Automated Deduction, LNCS 607.* Springer Verlag, 1992.
25. Henry M. Paynter. *Analysis and design of engineering systems.* MIT Press, Cambridge, M.A., 1961.
26. P.J.G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, (77):81–97, March 1989.
27. A.P. Ravn. Design of Embedded Real-time Computing Systems. Technical Report ID-TR:1995-170, Dept. of Computer Science, Technical University of Denmark, October 1995.
28. H. Ruess, N. Shankar, and M. Srivas. Modular Verification of SRT division. In *In proceedings of the International Conference on Computer Aided Verification, CAV'96, LNCS 1102*, pages 123–134. Springer Verlag, 1996.
29. J. U. Skakkebaek and N. Shankar. Towards a Duration Calculus Proof Assistant in PVS. In Langmaack et al. [16], pages 660–679.
30. U. Söderman. *Conceptual modelling of mode switching physical systems.* PhD thesis, Linköping University, Linköping, 1995. Dissertation no. 375.
31. J.-E. Strömberg and S. Nadjm-Tehrani. Hybrid Systems Verification Combining Duration Calculus and Bond Graphs. In *the invited session on Hybrid Dynamic Systems, Proc. IFAC-IFIP-IMACS Conference on Control of Industrial Systems*, pages 481–486. IFAC, 1997.

32. J.-E. Strömberg, S. Nadjm-Tehrani, and J. Top. Switched Bond Graphs as Front-end to Formal Verification of Hybrid Systems. In R. Alur, T.A. Henzinger, and E. Sontag, editors, *Proc. of the DIMACS International Workshop on Verification and Control of Hybrid Systems, LNCS 1066*, pages 282–293. Springer Verlag, 1996.
33. J.E. Strömberg. *A mode switching modelling philosophy.* PhD thesis, Linköping University, Linköping, 1994. Dissertation no. 353.
34. J.E. Strömberg, J.L. Top, and U. Söderman. Variable causality in bond graphs caused by discrete effects. In *Proc. First Int. Conf. on Bond Graph Modeling (ICBGM '93)*, number 2 in SCS Simulation Series, volume 25, pages 115–119, San Diego, 1993.
35. J.L. Top. *Conceptual modelling of physical systems.* PhD thesis, University of Twente, Enschede, 1993.
36. J. van Dijk. *On the role of bond graph causality in modelling mechatronic systems.* PhD thesis, University of Twente, Enschede, 1994.

Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach[1]

Aleksandra Tešanović

Jörgen Hansson

Linköping University
Department of Computer Science
Linköping, Sweden
{alete,jorha}@ida.liu.se

Dag Nyström

Christer Norström

Mälardalen University
Department of Computer Engineering
Västerås, Sweden
{dag.nystrom,christer.norstrom}@mdh.se

December 16, 2001

[1]This work is supported by ARTES, a network for real-time and graduate education in Sweden, and CENIT (project 01.07).

Abstract

As complexity and the amount of data are growing in embedded real-time systems, the need for a uniform and efficient way to store data becomes increasingly important, i.e., database functionality is needed to provide support for storage and manipulation of data. However, a database that can be used in an embedded real-time system must fulfill requirements both from an embedded system and from a real-time system, i.e., at the same time the database needs to be an embedded and a real-time database. The real-time database must handle transactions with temporal constraints, as well as maintain consistency as in a conventional database. The main objectives for an embedded database are low memory usage, i.e., small memory footprint, portability to different operating system platforms, efficient resource management, e.g., minimization of the CPU usage, ability to run for a long period of time without human presence, and ability to be tailored for different applications. In addition, development costs must be kept as low as possible, with short time-to-market and a reliable software. In this report we survey embedded and real-time database platforms developed in industrial and research environments. This survey represents the state-of-the-art in the area of embedded databases for embedded real-time systems. The survey enables us to identify a gap between embedded systems, real-time systems and database systems, i.e., embedded databases suitable for real-time systems are sparse. Furthermore, it is observed that there is a need for a more generic embedded database that can be tailored, such that the application designer can get an optimized database for a specific type of application. We consider integration of a modern software engineering technique, component-based software engineering, for developing embedded databases for embedded real-time systems. This merge provides means for building an embedded database platform that can be tailored for different applications, such that it has a small memory footprint, minimum of functionality, and is highly integrated with the embedded real-time system.

Contents

1 Introduction **3**

2 Database systems **9**
2.1 Traditional database systems 9
2.2 Embedded database systems 9
2.2.1 Definitions 9
2.2.2 An industrial case study 10
2.3 Commercial embedded DBMS: a survey 11
2.3.1 Criteria investigated 11
2.3.2 Databases investigated 12
2.3.3 DBMS model 13
2.3.4 Data model 13
2.3.5 Data indexing 15
2.3.6 Memory requirements 18
2.3.7 Storage media 18
2.3.8 Connectivity 19
2.3.9 Operating system platforms 22
2.3.10 Concurrency control 23
2.3.11 Recovery 25
2.3.12 Real-time properties 26
2.4 Current state-of-the-art from research point of view 26
2.4.1 Real-time properties 27
2.4.2 Distribution 30
2.4.3 Transaction workload characteristics 33
2.4.4 Active databases 36
2.5 Observations 39

3 Component-based systems **40**
3.1 Component-based software development 41
3.1.1 Software component 42
3.1.2 Software architecture 43
3.1.3 The future of component-based software engineering: from the component to the composition 44
3.2 Component-based database systems 46
3.2.1 Granularity level of a database component 47
3.2.2 Component vs. monolithic DBMS 48
3.2.3 Components and architectures in different CDBMS models 49
3.2.4 Extensible DBMS 49
3.2.5 Database middleware 52
3.2.6 DBMS service 57
3.2.7 Configurable DBMS 57
3.3 Component based embedded and real-time systems 60

3.3.1 Components and architectures in different component-based embedded (and) real-time systems 61
3.3.2 Extensible systems 62
3.3.3 Middleware systems 62
3.3.4 Configurable systems 64
3.4 A tabular overview 67

4 Summary **76**
4.1 Conclusions 76
4.2 Future work 78

Chapter 1

Introduction

Digital systems can be classified in two categories: general purpose systems and application-specific systems [41]. General purpose systems can be programmed to run a variety of different applications, i.e., they are not designed for any special application. Application-specific systems are designed for a specific application. Application-specific systems can also be part of a larger host system and perform specific functions within the host system [20], and such system are usually referred to as *embedded systems.* An embedded system is implemented partly on software and partly on hardware. When standard microprocessors, microcontrollers or DSP processors are used, specialization of an embedded system for a particular application consists primarily on specialization of software. In this report we focus on such systems. An embedded system is required to be operational during the lifetime of the host system, which may range from a few years, e.g., a low end audio component, to decades , e.g., an avionic system. The nature of embedded systems also requires the computer to interact with the external world (environment). They need to monitor sensors and control actuators for a wide variety of real-world devices. These devices interface to the computer via input and output registers and their operational requirements are device and computer dependent.

Most embedded systems are also real-time systems, i.e., the correctness of the system depends both on the logical result of the computation, and the time when the results are produced [104]. We refer to these systems as *embedded real-time systems*[1] Real-time systems are typically constructed out of concurrent programs, called tasks. The most common type of temporal constraint that a real-time system must satisfy is the completion of task deadlines. Depending on the consequence due to a missed deadline, real-time systems can be classified as hard or soft. In a *hard real-time system* consequences of missing a deadline can be catastrophic, e.g., aircraft control, while in a *soft-real-time system*, missing a deadline does not cause catastrophic damage to the system, but affect performance negatively, e.g., multimedia. Below follows a list of examples where embedded real-time systems can be found.

- Vehicle systems for automobiles, subways, aircrafts, railways, and ships.
- Traffic control for highways, airspace, railway tracks, and shipping lines.
- Process control for power plants and chemical plants.
- Medical systems for radiation therapy and patient monitoring.
- Military uses such as advanced firing weapons, tracking, and command and control.

[1] Some authors use the terms embedded systems and real-time systems interchangeably. We, however, distinguish between embedded and real-time systems, since there are some embedded systems that do not enforce real-time behavior, and there are real-time systems that are not embedded.

- Manufacturing systems with robots.
- Telephone, radio, and satellite communications.
- Multimedia systems that provide text, graphic, audio and video interfaces.
- Household systems for monitoring and controlling appliances.
- Building managers that control such entities as heat, lights, doors, and elevators.

In the last years the deployment of embedded real-time systems has increased dramatically. As can be seen from the examples, these systems are now virtually embedded in every aspect of our lives. At the same time the amount of data that needs to be managed is growing, e.g., embedded real-time systems that are used to control a vehicle, such as a modern car, must keep track of several hundreds of sensor values. As the amount of information managed by embedded real-time systems increases, it becomes increasingly important that data is managed and stored efficiently and in a uniform manner by the system. Current techniques adopted for storing and manipulating data objects in embedded and real-time systems are ad hoc, since they normally manipulate data objects as internal data structures. That is, in embedded real-time systems data management is traditionally built as a part of the overall system. This is a costly development process with respect to design, implementation, and verification of the system. In addition, such techniques do not provide mechanisms that support porting of data to other embedded systems or large central databases.

Database functionality is needed to provide support for storage and manipulation of data. Embedding databases into embedded systems have significant gains: (i) reduction of development costs due to the reuse of database systems; (ii) improvement of quality in the design of embedded systems since the database provides support for consistent and safe manipulation of data, which makes the task of the programmer simpler; and (iv) increased maintainability as the software evolves. Consequently, this improves the overall reliability of the system. Furthermore, embedded databases provide mechanisms that support porting of data to other embedded systems or large central databases.

However, embedded real-time systems put demands on such embedded database that originate from requirements on embedded and real-time systems.

Most embedded systems need to be able to run without human presence, which means that a database in such a system must be able to recover from the failure without external intervention [81]. Also, the resource load the database imposes on the embedded system should be carefully balanced, in particular, memory footprint. For example, in embedded systems used to control a vehicle minimization of the hardware cost is of utmost importance. This usually implies that memory capacity must be kept as low as possible, i.e., databases used in such systems must have small memory footprint. Embedded systems can be implemented in different hardware environments supporting different operating system platforms, which requires the embedded database to be portable to different operating system platforms.

On the other hand, real-time systems put different set of demands on a database system. The data in the database used in real-time systems must be logically consistent, as well as temporally consistent [51]. Temporal consistency of data is needed in order to maintain consistency between the actual state of the environment that is being controlled by the real-time system, and the state reflected by the content of the database. Temporal consistency has two components:

- *Absolute consistency*, between the state of the environment and its reflection in the database.
- *Relative consistency*, among the data used to derive other data.

We use the notation introduced by Ramamritham [51] to give a formal definition of the temporal consistency.

A data element, denoted d, which is temporally constrained, is defined by three attributes:

- value d_{value}, i.e., the current state of data d in the database,
- time-stamp d_{ts}, i.e., the time when the observation relating to d was made, and
- absolute validity interval d_{avi}, i.e., the length of the time interval following d_{ts} during which d is considered to be absolute consistent.

A set of data items used to derive a new data item forms a relative consistency set, denoted R, and each such set is associated with a relative validity interval, R_{rvi}. Data in the database, such that $d \in R$, has a correct state if and only if [51]

1. d_{value} is logically consistent, and
2. d is temporally consistent, both
   - absolute $(t - d_{ts}) \leq d_{avi}$, and
   - relative $\forall d' \in R, |d_{ts} - d'_{ts}| \leq R_{rvi}$.

A transaction, i.e., a sequence of read and write operations on data items, in conventional databases must satisfy following properties: atomicity, consistency, isolation, and durability, called ACID properties. In addition, transactions that process real-time data must satisfy temporal constraints. Some of the temporal constraints on transactions in a real-time database come from the temporal consistency requirement, and some from requirements imposed on the system reaction time (typically, periodicity requirements) [51]. These constraints require time-cognizant transaction processing so that transactions can be processed to meet their deadlines, both with respect to completion of the transaction as well as satisfying the temporal correctness of the data.

An example application

We describe one example of a typical embedded real-time system. The example is typical for a large class industrial process control system, that handles large volume of data, where data have temporal constraints. In order to keep the example as illustrative and as simple as possible we limit our example to a specific application scenario, namely the control of the water level in a water tank (see figure 1.1). Through this example we illustrate demands put on data management in such a system. Our example-system contains a real-time operating system, a database, an I/O management subsystem and a user-interface.

A controller task (PID-regulator) controls the level in the water tank according to the desired level set by the user, i.e., the highest allowed water level. The environment consists of the water level, the alarm state, the setting of the valve and the value of the user interface. The environment state is reflected by the content of the database (denoted a', x', y', r' in figure 1.1). Furthermore, PID variables that reflect internal status of the system are also are stored in the database. The I/O manager (I/O MGNT) is responsible for data exchange between the environment and the database. Thus, the database stores the parameters from the environment and to the environment, and configuration data. Data in the database need to be temporally consistent, both absolute and relative. In this case, absolute validity interval can depend on the water flow. That is, if the the tank in figure 1.1 is very big and the flow is small, absolute validity interval can be greater than in the case where the water-flow is big and the data needs to be sampled more frequently. The relative consistency depicts the difference between the oldest data sample and the youngest sample of data. Hence, if

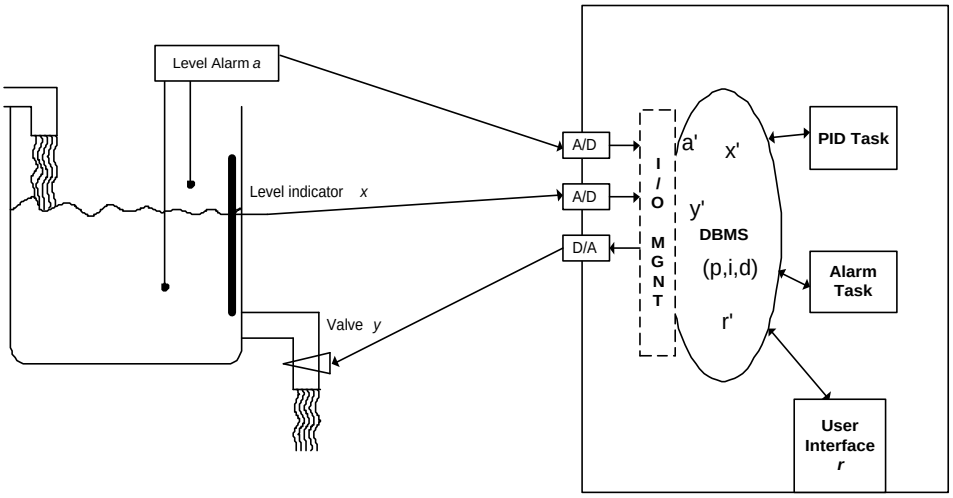


Figure 1.1: *An example system. On the left side a water tank is controlled by the system on the right side. The PID-task controls the flow out by the valve (y) so that the level in tank (x) is the same as the level set by the user (r). An alarm task shuts the system down if the level alarm (a) is activated.*

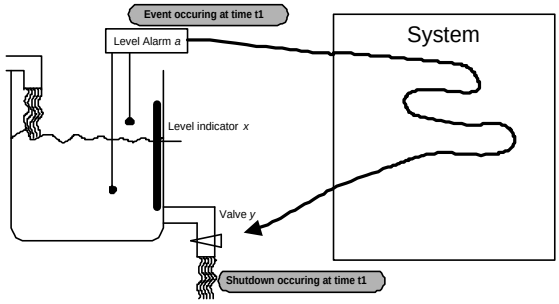


Figure 1.2: *The end-to-end deadline constraint for the alarm system. The emergency shutdown must be completed within a given time* dl_{alarm} *implying that* $t2 - t1 \leq dl_{alarm}$.

the alarm in our system is activated, due to high water level, and *x'* indicates a lover level, these two values are not valid even though they have absolute validity.

For this application scenario, an additional temporal constraint must be satisfied by the database, and that is an end-to-end deadline. This temporal constraint is important because the maximum processing time for the alarm event must be smaller than the end-to-end-deadline. Figure 1.2 shows the whole end-to-end process for an alarm event. When an alarm is detected, an alarm sensor sends the signal to the A/D converter. This signal is read by the I/O manager recording the alarm in the database. The alarm task then analyzes the alarm data and sends a signal back to indicate an emergency shutdown.

In our example, the database can be accessed by the alarm task, the PID task, the user interface, and the I/O manager. Thus, an adequate concurrency control must be ensured in order to serialize transactions coming from these four different database clients.

Let us now assume that the water level in the tank is just below the highest allowed level, i.e., the level when an alarm is triggered. The water flowing into the tank creates ripples on the surface. These ripples could cause the alarm to go on and off with every ripple touching the sensor, and consequently sending bursts of alarms to the system. In this case, one more temporal constraint must be satisfied, a delayed response. The delayed

response is a period of time within which the water level must be higher than the highest allowed level in order to activate the alarm.

As we can see, this simple application scenario puts different requirements on the database. A complete industrial process control system, of which this example is part of, would put a variety of additional requirements on a database, e.g., logging.

Note that requirements placed on the database by the embedded real-time system are to some extent general for all embedded and real-time applications, but at the same time, there are requirements that are specific to an application in question (e.g., delayed response). Thus, an embedded database system must, in a sense, be tailored, i.e., customized for each different application to give an optimized solution. That is, given the resource demands of the embedded real-time system, a database must be tailored to have minimum functionality, i.e., only functionality that a specific application needs.

In recent years, a significant amount of research has focused on how to incorporate database functionality into real-time systems without jeopardizing timeliness, and how to incorporate real-time behavior into embedded systems. However, research for embedded databases used in embedded real-time systems, explicitly addressing the development and design process, and the limited amount of resources in embedded systems is sparse. Hence, the goal of our report is to identify the gap between the following three different systems: real-time systems, embedded systems, and database systems. Further, we investigate how component-based software engineering would provide a feasible approach for bridging this gap by enabling development of a customizable embedded database suitable for embedded real-time systems, in a short time with reduced development costs, and high quality of software.

There are many embedded databases on the market, but, as we show in this report, they vary widely form vendor to vendor. Existing commercial embedded database systems, e.g., Polyhedra [93], RDM and Velocis [70, 71], Pervasive.SQL [91], Berkeley DB [102], and TimesTen [113], have different characteristics and are designed with specific applications in mind. They support different data models, e.g., relational vs object-oriented model, and operating system platforms. Moreover, they have different memory requirements and provide different types of interfaces for users to access data in the database.

Application developers must carefully choose the embedded database their application requires, and find the balance between the functionality an application requires and the functionality that an embedded database offers. Thus, finding the right embedded database, in addition of being a quite time consuming, costly and difficult process, is a process with lot of compromises. Although a significant amount of research in real-time databases has been done in the past years, it has mainly focussed on various schemes for concurrency control, transaction scheduling, and logging and recovery. Research projects that are building real-time database platforms, such as ART-RTDB [118], BeeHive [50], DeeDS [99] and RODAIN [49], mainly address real-time performance, have monolithic structure, and are built for a particular real-time application. Hence, the issue of how to enable development of an embedded database system that can be tailored for different embedded real-time applications arises. The development costs of such database system must be kept low, and the development must ensure good software quality. We show that exploiting component-based software engineering in the database development seem to have a potential, and examine how component-based software engineering can enable database systems to be easily tailored, i.e., optimized, for a particular application. By having well-defined reusable components as building blocks, not only development costs are reduced, but more importantly, costs related to verification and certification are reduced since components will only need to be checked once. Although some major database vendors such as Oracle, Informix, Sybase, and Microsoft have recognized that component-based development offers significant benefits, their component-based solutions are limited in terms of tailorability with, in most cases, inadequate support for development of such component-based database systems. Furthermore, commercial component-based databases do not en-

force real-time behavior and are not, in most cases, suitable for environments with limited resources. We argue that in order to compose a reliable embedded real-time system out of components, component behavior must be predictable. Thus, components must have well defined temporal attributes. However, existing component-based database systems do not enforce real-time behavior. Also, issues related to embedded systems such as low-resource consumption are not addressed at all in these solutions. We show that real-time properties are preserved only in a few of the existing component-based real-time systems. In addition to temporal properties, components in component-based embedded systems must have explicitly addressed memory needs and power consumption requirements. At the end of this report, we outline initial ideas for preserving real-time and embedded properties in components used for building an embedded database platform that can be tailored for different applications.

The report is organized as follows. In chapter 2 we survey commercial embedded and research real-time database systems. We then examine the component-based software engineering paradigm, and its integration with databases for embedded real-time systems in chapter 3. In chapter 4 we discuss a possible scenario for developing an embedded database platform that can be tailored for different applications, and give conclusions.

Chapter 2

Database systems

2.1 Traditional database systems

Databases are used to store data items in a structured way. Data items stored in a database should be persistent, i.e., a data item stored in the database should remain there until either removed or updated. Transactions are most often used to read, write, or update data items. Transactions should guarantee serialization. The so-called database manager is accessed through interfaces, and one database manager can support multiple interfaces like C/C++, SQL, or ActiveX interfaces.

According to [26] most database management systems consist of three levels:

- The internal level, or physical level, deals with the physical storage of the data onto a media. Its interface to the conceptual level abstracts all such information away.
- The conceptual level handles transactions and structures of the data, maintaining serialization and persistence.
- The external level contains interfaces to users, uniforming transactions before they are sent to the conceptual level.

One of the main goals for many traditional database systems are transaction throughput and low average response time [51], while for real-time databases the main goal is to achieve predictability with respect to response times, memory usage and CPU utilization. We can say that when a worst case response time or maximum memory consumption for a database can be guaranteed, the system is predictable. However, there can be different levels of predictability. For a system that can guarantee a certain response time with some defined confidence, is said to have a certain degree of predictability.

2.2 Embedded database systems

2.2.1 Definitions

A device embedded database system is a database system that resides into an embedded system. In contrast, an application-embedded database is hidden inside an application and is not visible to the application user. Application-embedded databases are not addressed further in this survey. This survey focuses only on databases embedded in real-time and embedded systems. The main objectives of a traditional enterprise database system often is throughput, flexibility, scalability, functionality etc., while things like size, resource usage, and processor usage are not as important, since hardware is relatively cheap. In embedded systems these issues are much more important. The main issues for an embedded database system can be summarized as [98, 68, 96]:

- Minimizing the memory footprint: The memory demand for an embedded system are most often, mainly for economical reasons, kept as low as possible. A typical footprint for an embedded database is within the range of some kilobytes to a couple of megabytes.
- Reduction of resource allocations: In an embedded system, the database management system and the application are most often run on the same processor, putting a great demand on the database process to allocate minimum CPU bandwidth to leave as much capacity as possible to the application.
- Support for multiple operating systems: In an enterprise database system, the DBMS is typically run on a dedicated server using a normal operating system. The clients, that could be desktop computers, other servers, or even embedded systems, connect to the server using a network connection. Because a database most often run on the same piece of hardware as the application in an embedded system, and that embedded systems often use specialized operating systems, the database system must support these operating systems.
- High availability: In contrast to a traditional database system, most embedded database systems do not have a system administrator present during run-time. Therefore, an embedded database must be able to run on its own.

Depending on the kind of system the database should reside in, some additional objectives might be more emphasized, while others are less important. For example, Pervasive, which manufactures Pervasive.SQL DBMS system, has identified three different types of embedded systems and has therefore developed different versions of their database to support these systems [90]:

- Pervasive.SQL for smart cards is intended for systems with very limited memory resources, typically a ten kilobytes. This version has traded off sophisticated concurrency control and flexible interfaces for size. Typical applications include banking systems like cash-cards, identification systems, health-care, and mobile phones.
- Pervasive.SQL for embedded systems is designed for small control systems like our example system in figure 1.1. It can also be used as a data pump, which is a system that reads data from a number of sensors, stores the data in a local database, and continuously “pumps” data to a large enterprise database in an unidirectional flow. Important issues here are predictability, with respect to both timing and system resources, as we are approaching a real-time environment. The interfaces are kept rather simple to increase speed and predictability. Since the number of users and the rate of transactions often can be predicted, the need for a complex concurrency control system might not be necessary.
- Pervasive.SQL for mobile systems is used in mobile devices like cellular phones or PDAs, where there is a need for higher degree of concurrency control. Consider that a user browses through e-mails on a PDA while walking into his/hers office, then the synchronizer updates the e-mail list using a wireless communication without interrupting the user. The interfaces support more complex ad-hoc queries than the embedded version. The interfaces are modular and can be included or excluded to minimize memory footprint.

2.2.2 An industrial case study

In 1999, ABB Robotics who develops and manufactures robots for industrial use, wanted to exchange the existing in-house developed configuration storage management system

into a more flexible and generic system, preferably some commercial-of-the- shelf (COTS) embedded database.

An industrial robot is a complex and computer-controlled system, which consists of many sub-systems. In each robot some configuration information about its equipment is stored. Today, this amounts to about 600 kilobytes of data [86]. This data needs to be stored in a structured and organized way, allowing easy access.

The current system, called CFG, is a custom made application and resembles in many ways to a regular database application. However, it is nowadays considered to be too non-flexible and the user-interface is not user friendly enough. Furthermore, the internal structures and the choice of index system have lead to much longer response times as the data size has increased.

ABB Robotics decided to investigate the market for an existing suitable database system that would fulfill their demands. The following requirements were considered to be important[63]:

- Connectivity. It should be possible to access the database both directly from the robot controlling application and from an external application via a network connection. Furthermore the database should support *views* so data could be queried from a certain perspective. It would be preferable if some kind of standard interface, e.g., ODBC, could be used. If possible, the database should be backward compatible with the query language used in the CFG system. The database should also be able to handle simultaneous transactions and queries, and be able to handle concurrency.

- Scalability. The amount of data items in the system must be allowed to grow as the system evolves. Transaction times and database behavior must not change due to increase in data size.

- Security. It should be possible to assign different security levels for different parts of the data in the database. That is, some form of user identification and some security levels must exist.

- Data persistence. The database should be able to recover safely after a system failure. Therefore some form of persistent storage must be supported, and the database should remain internally consistent after recovery.

- Memory requirements. To keep the size of the DBMS system low is a key issue.

2.3 Commercial embedded DBMS: a survey

In this section we discuss and compare a number of commercial embedded database systems. We have selected a handful of systems with different characteristics. These databases are compared with each other with respect to a set of criteria, which represent the most fundamental issues related to embedded databases.

2.3.1 Criteria investigated

- DBMS model, which describes the architecture of the database system. Two DBMS architectures for embedded databases are the client/server model and the embedded library model.

- Data model, which specifies how data in the database is organized. Common models are the relational, the object-oriented and the object relational.

- Data indexing, which describes how the data indexes are constructed.

- Memory requirements, which describes the memory requirements for the system, both data overhead and the memory footprint, which is the size of the database management system.
- Storage media, which specifies the different kinds of storage medias that the database supports.
- Connectivity, which describes the different interfaces the application can use to access the database. Network connectivity is also specified, i.e. the ability to access the database remotely via a network connection.
- Operating system platforms, which specifies the operating systems supported by the database system.
- Concurrency control, which describes how concurrent transactions are handled by the system.
- Recovery, which specifies how backup/restoration is managed by a system in case of failures.
- Real-time properties, which discusses different real-time aspects of the system.

2.3.2 Databases investigated

In this survey we have selected a handful of systems that together represents a somewhat complete picture of the products currently on the market. This list of systems is not to be considered complete in any way, worth mentioning are, for example, the SQL.Anywhere system developed by Sybase [110] and the c-tree Plus system by FairCom [35].

- Pervasive.SQL by Pervasive Software Inc. This database has three different versions for embedded systems: Pervasive.SQL for smart-card, Pervasive.SQL for mobile systems, and Pervasive.SQL for embedded systems. All three versions integrate well with each other and also with their non embedded versions of Pervasive.SQL. Their system view and the fact that they have, compared to most embedded databases, very small memory requirements was one reason for investigating this database [91].
- Polyhedra by Polyhedra Plc. This database was selected for three reasons, first of all it is claimed to be a real-time database, secondly it is a main memory database and third, it has active behavior [93].
- Velocis by Mbrane Ltd. This system is primarily intended for e-Commerce and Web applications, but has some support for embedded operating systems [71].
- RDM by Mbrane Ltd. Like Polyhedra, RDM also claims to be a real-time database. It is however fundamentally different from the Polyhedra system, by, for example, being an embedded library and, thus, does not adopt the client/server model [70].
- Berkeley DB by Sleepycat Software Inc. This database, which also is implemented as a library, was selected for the survey because it is distributed as open source and therefore interesting from a research point of view [102].
- TimesTen by TimesTen Performance Software. This relational database is, like the Polyhedra system a main memory real-time database system [113].

DBMS system	Client/server	Library
Pervasive.SQL	x	
Polyhedra	x	
Velocis	x	
RDM		x
Berkeley DB		x
TimesTen	x	

Table 2.1: DBMS models supported by different embedded database systems.

2.3.3 DBMS model

There are basically two different DBMS models supported (see table 2.1). The first model is the client/server model, where the database server can be considered to be an application running separately from the real-time application, even though they run on the same processor. The DBMS is called using a request/response protocol. The server is accessed either through inter-process communication or some network. The second model is to compile the DBMS together with the system application into one executable system. When a task wants to access the database it only performs function calls to make the request. Transactions execute on behalf of their tasks' threads, even though internal threads in the DBMS might be used. There are advantages and drawbacks with both models. In a traditional enterprise database, the server most often run on a dedicated server machine, allowing the DBMS to use almost 100% of the CPU. However in an embedded client/server system, the application and the server process often share the same processor. This implies that for every transaction at least two context switches must be performed, see figure 2.1. More complex transactions, like an update transaction might require even more context switches. Consider for example an real-time task that executes an update transaction that first reads the data element x, then derives a new value of x from the old value and finally writes it back to the database. This transaction would generate four context switches. Two while fetching the value of x and two while writing x back to the database.

These context-switches adds to the system overhead, and can furthermore make worst case execution time estimations of transactions more complex to predict. The network message passing or inter-process communication between the server and the client also add to the overhead cost. A drawback with an embedded library is that they lack standardized interfaces like ODBC. [68].

The problem with message passing overhead has been reduced in the Velocis system. They allow the process to be compiled together with the application, thus reducing the overhead since shared memory can be used instead.

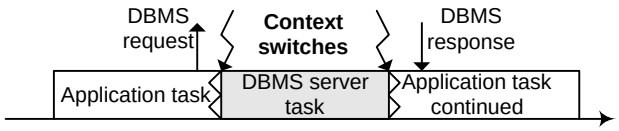


Figure 2.1: *In an embedded client/server DBMS at least two context switches are necessary for each transaction.*

2.3.4 Data model

The data model concerns how data is logically structured. The most common model is the relational model where data is organized in tables with columns and rows. Databases

DBMS system	Relational	Object-oriented	Object-relational	Other
Pervasive.SQL	x			
Polyhedra	x		(x)	
Velocis	x			
RDM	x			
Berkeley DB				x
TimesTen	x			

Table 2.2: Data models supported by different embedded database systems.

implementing relational data model are referred to as relational databases (RDBMS). One advantage with a relational model is that columns in the table can relate to other tables so arbitrary complex logical structures can be created. From this logical structure queries can be used to extract a specific selection of data, i.e. a view. However, one disadvantage with the relational model is added data lookup overhead. This is because that data elements are organized with indexes in which pointers to the data is stored, and to perform the index lookup can take significant amount of time, especially for databases stored on hard drives. This can be a serious problem for databases that resides in time critical applications like our example application in figure 1.1.

The relational model, which was developed by Codd, only supports a few basic data types, e.g., integers, floating point numbers, currency, and fixed length arrays. This is nowadays considered a limitation which has lead to the introduction of databases which supports more complex data types, e.g., Binary Large Objects (BLOBs). BLOB is data, which is treated as a set of binary digits. The database does not know what a BLOB contains and therefore cannot index anything inside of the BLOB.

The object-oriented database (ODMBS) is a different kind of data model, which is highly integrated with object-oriented modeling and programming, The ODBMS is an extension to the semantics of an object-oriented language. An object-oriented database stores objects that can be shared by different applications. For example, a company which deals with e-Commerce has a database containing all customers. The application is written in an object-oriented language and a customer is modeled as an object. This object has methods, like `buy` and `changeAdress`, associated with it. When a new customer arrives, an instance of the class is created and then stored in the database. The instance can then be retrieved at any time. Controlled access is guaranteed due to concurrency control.

A third data model, which has evolved from both the relational and the object-oriented model, incorporates objects in relations, thus is called object-relational databases (ORDBMS).

As shown in table 2.2, all systems in the survey except Berkeley DB are relational. Furthermore the Polyhedra has some object-relational behavior through its Control language described below. However it is not fully object-relational since objects itself cannot be stored in the database. To our knowledge, no pure object-oriented embedded databases exists on the market today. There are however some low requirements databases that are object-oriented, such as the Objectivity/DB [80] system. Furthermore some object-oriented client-libraries exists that can be used in an embedded system, such as the PowerTier [89] and the Poet Object system [92]. These client-libraries connects to an external database server. Since the actual DBMS and the database is located on a non-embedded system we do not consider them embedded databases in this survey.

Most systems have ways to “shortcut” access to data, and therefore bypassing the index lookup routine. Pervasive, for example, can access data using the Btrieve transactional engine that bypasses the relational engine. Mbrane uses a different approach in the RDM system. In many real-time systems data items are accessed in a predefined order (think of a controlling system where some sensor values are read from the database and the result

is written back to the database). By inserting shortcuts between data elements such that they are directly linked in the order they are accessed, very fast accesses can be achieved. As these shortcuts point directly to physical locations in memory, reorganization of the database is much more complex since a large number of pointers can become stale when a single data is dropped or moved.

The Polyhedra DBMS system is fundamentally different compared to the rest of the relational systems in this survey, because of its active behavior. This is achieved through two mechanisms, active queries and by the control language (CL). An active query looks quite like a normal query where some data is retrieved and/or written, but instead the query stays in the database until explicitly aborted. When a change in the data occurs that would alter the result of the query, the application is notified. The CL, which is a fully object-oriented script language that supports encapsulation, information hiding and inheritance, can determine the behavior of data in the database. This means that methods, private or public, can be associated with data performing operations on them without involving the application. In our example application, the CL could be used to derive the valve setting y' from the level indicator x' and the PID parameters, thus removing the need for the PID task. The CL has another important usage, since it can be used in the display manager (DM) to make a graphical interface. The DM, which is implemented as a Polyhedra Client, together with the control language is able to present a graphical view on a local or remote user terminal. This is actually exactly the user interface in our example application, since DM also can take user input and forward that to the database. Active database management will be further discussed in section 2.4.4.

As mentioned above, Berkeley DB is the only non-relational system in this survey. Instead it uses a key-data relationship. One data is associated with a key. There are three ways to search for data, from key, part of key or sequential search. The data can be arbitrary large and of virtually any structure. Since the keys are plain ASCII strings the data can contain other keys so complex relations can be built up. In fact it would be possible to implement a relational engine on top of the Berkeley DB database. This approach claims for a very intimate relationship between the database and the programming language used in the application.

2.3.5 Data indexing

To be able to efficiently search for specific data, an efficient index system should exist. To linearly search through every single key from an unsorted list would not be a good solution since transaction response times would grow as the number of data elements in the database increases. To solve this problem two major approaches are used, tree structures and hashed lists. Both approaches supply similar functionality, but differ somewhat in performance. Two major tree structures are used, B^+-tree indexing, which suits disk based databases, and T-tree indexing, which is primarily used in main-memory databases.

The main issue for B^+-tree indexing is to minimize disk I/O, thus trading disk I/O for added algorithm complexity. B^+-tree indexing sorts the keys in a tree structure in which every node has a predefined number of children, denoted p. A large value of p results in a wide but shallow tree, thus the tree has a large fanout. A small value of p results in the opposite. For $p = 2$ tree is a binary tree. Since all data indexes reside in the leave nodes of the tree, while only the inner nodes are used to navigate to the right leave, the number of visited nodes will be fewer for a shallow tree. This results in fewer nodes that have to be retrieved from disk, thus reducing disk I/O. In figure 2.3.5 we see an example of a B^+-tree . The root directs requests to all keys that are less than six to the left child and keys from six to its middle child. As the value of p increases, the longer time it takes to pass the request to the correct child. Thus, when deciding the fanout degree, the time it takes to fetch a node from disk must be in proportion to the time it takes to locate which child to redirect to. Proposals on real-time concurrency control for B^+-tree indexing have been made [59].

For main-memory databases a different type of tree can be used, the T-tree structure

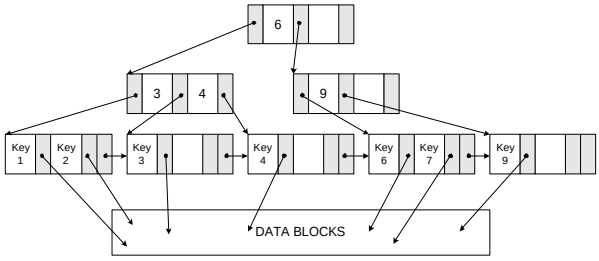


Figure 2.2: *The structure of a B^+-tree with fanout of 3. The inner nodes are only used to index down to the leave node containing the pointer to the correct data. Figure partially from [59].*

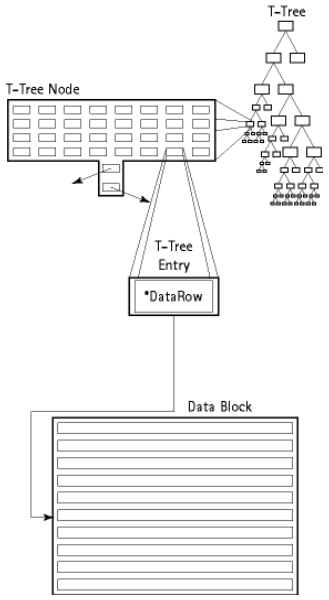


Figure 2.3: *The structure of a T-tree. Each node contains a number of entries that contains the key and a pointer to the corresponding data. All key values that are less then the boundary of this node is directed to the left child, and key values that are above the boundary are directed to the right. Figure from TimesTen White paper.*

DBMS system	B^+-tree	T-tree	Hashing	Other
Pervasive.SQL	x			
Polyhedra			x	
Velocis	n/a	n/a	n/a	n/a
RDM	x			
Berkeley DB	x		x	x
TimesTen	x	x	x	

Table 2.3: Data indexing strategies used by different embedded database systems.

[61]. The T-tree uses a deeper tree structure than the B^+-tree since it is a balanced binary tree, see figure 2.3.5, with a maximum of two children for every node. To locate data, more nodes are normally visited, compared to the B^+-tree , before the correct node is found. This is not a problem for a main-memory database since memory I/O is significantly faster than disk I/O. The primary issue for T-tree indexing is that the algorithms used to traverse a T-tree have a lower complexity and faster execution time than corresponding algorithms for the B-tree. However, it has been pointed out that a T-tree traversal combined with concurrency control perform worse than B-tree indexes due to the fact that more nodes in the T-tree had to be locked to ensure tree integrity during traversal [65] . They also proposed an improvement called T-tail, which reduces the number of costly re-balancing operations needed. Furthermore, they proposed two concurrency algorithms for T-tail and T-tree structures, one optimistic and one pessimistic (for a more detailed discussion about pessimistic and optimistic concurrency control, see section 2.3.10).

The second approach, hashing, uses an hash list in which keys are inserted according to a value derived from the key itself (different keys can be assigned the same value) [62]. Therefore, each entry in the list is a bucket that can contain a predefined number of data keys. If a bucket is full a rehash operation must be performed. The advantage with hashing is that the time to search for certain data is constant independent of the amount of data entries in the database. However, hashing often cause more disk I/O than B-trees. This is because data is often used with locality of reference, i.e., if data element d_1 is used, it is more probable that data element d_2 will be used shortly. In a B-tree both elements would be sorted close to each other, and when d_1 is retrieved from disk, then it is probable that also d_2 will be retrieved. However in a hash list d_1 and d_2 are likely to be in different buckets, causing extra disk I/O if both data are used. Another disadvantage with hashing compared to tree-based indexing is that non-exact queries is time consuming to perform. For example, consider a query for all keys greater than x. After that x has been found, all keys are found to the right of x in a B^+-tree . For a hashed index, all buckets need to be searched to find all matching keys.

We can notice that main-memory databases, namely Polyhedra and TimesTen use hashing (see table 2.3). Additionally, TimesTen supports T-trees. Pervasive, RDM and Berkeley DB use B-tree indexing.

It is also noteworthy that Berkeley DB uses both B^+-tree and hashing. They claim that hashing is suitable for database schemes that are either so small that the index fits into main memory or when the database is so large that B^+-tree indexing will cause disk I/O upon most node fetching due to that the cache can only fit a small fraction of the nodes. Thus making the B^+-tree indexing suitable for database schemes with a size in between these extremes. Furthermore, Berkeley DB supports a third access method, queue, which is used for fast inserts in the tail, and fast retrieval of data from the head of the queue. This approach is suitable for the large class of systems that consume large volumes of data, e.g., state machines.

DBMS system	Memory requirements[1]
Pervasive.SQL for smart cards	8kb
Pervasive.SQL for embedded systems	50kb
Pervasive.SQL for mobile systems	50 - 400kb
Polyhedra	1.5 - 2Mb
Velocis	4Mb
RDM	400 - 500kb
Berkeley DB	175kb
TimesTen	5Mb

Table 2.4: Different memory needs of investigated embedded database systems.

2.3.6 Memory requirements

Memory requirement of the database is an important issue for embedded databases residing in environments with small memory requirements. For mass-produced embedded computer systems like computer nodes in a car, minimizing hardware is usually a significant factor for reducing development costs. There are two interesting properties to consider for embedded databases, first of all the memory footprint size, which is the size of the database without any data elements in it. Second, data overhead is of interest, i.e., the number of bytes required to store a data element apart from the size of the data element itself. An entry in the index list with a pointer to the physical storage address is typical data overhead. Typically client/server solutions seems to require significantly more memory than embedded libraries, with Pervasive.SQL being the exception (see table 2.4). This exception could partially be explained by Pervasive's Btrieve native interface being a very low-level interface (see section 2.3.8), and that much of the functionality is placed in the application instead. In the case of Berkeley DB, there is a similar explanation to its small memory footprint. Additional functionality (not provided by the vendor) can be implemented on top of the database.

Regarding data overhead, Polyhedra has a data overhead of 28 bytes for every record. Pervasive's data overhead is not as easily calculated since it uses paging. Depending on record and page sizes different amount of fragmentation is introduced in the data files. There are however formulas provided by Pervasive for calculating exact record sizes. The other systems investigated in this survey supplies no information about actual data overhead costs.

For systems where low memory usage is considered more important than processing speed, there is an option of compressing the data. Data will then be transparently compressed and decompressed during runtime. This, however, requires free working memory of 16 times the size of the record being compressed/decompressed.

2.3.7 Storage media

Embedded computer systems support different storage medias. Normally, data used on the computer is stored on a hard-drive. Hand-held computer use Flash or other non-volatile memory for storage of both programs and data. Since data in a database needs to be persistent even upon a power loss, some form of stable storage technique must be used. As can be seen from table 2.5, most systems support Flash in addition to a hard-drive. The Polyhedra system also supports storage via a FTP-connection. This implies that the Polyhedra system can run without persistent storage, since data can be loaded via FTP-connection upon system startup.

[1] The values given in the table are the "footprint"-values made available by database vendors.

DBMS system	Hard-drive	Flash	Smart card	FTP
Pervasive.SQL	x	x	x	
Polyhedra	x	x		x
Velocis	x			
RDM	x			
Berkeley DB	x	x		
TimesTen	x			

Table 2.5: Storage medias used by investigated embedded database systems.

2.3.8 Connectivity

Database interfaces are the users way to access the database. An interface normally contains functions for connecting to the server, making queries, performing transactions, and making changes to the database schema. However different interfaces can be specialized on specific types of operations, e.g., SQL is used mainly for schema modifications and queries, while a C/C++ API normally are specialized for transactional access. Figure 2.4 shows a typical interface configuration for an embedded database system.

The most basic interface is the native interface. This interface, which is for a specific programming language, e.g., C or C++, is often used by the application running on the embedded system. In our example in figure 1.1, tasks and I/O management would typically use the native interface to perform their transactions.

Microsoft has developed the Open Database Connectivity (ODBC) interface in the late 80's. This interface uses the query language SQL, and is today one of the largest standards of database interfaces. The advantage with ODBC is that any database that supports ODBC can connect with any application written for ODBC. ODBC is a low-level interface, that requires more implementation per database access then a high-level interface like ActiveX Data Object (ADO) interface, explained below in more detail. The ODBC interface cannot handle non-relational databases very well. Main benefits, beside the interchangeability, is that the performance is generally higher than high-level interfaces, and the interface gives the possibility of detailed tuning.

A newer standard, also developed by Microsoft, is the OLE DB interface which is based on the Microsoft COM technology (for more detailed description of OLE DB's functionality see section 3.2.5). OLE DB is, as ODBC, a low-level interface. The functionality of OLE DB is similar to ODBC, but object-oriented. However, one advantage is the possibility to use OLE DB on non-relational database systems. There are also drivers available to connect the OLE DB interface on top of an ODBC driver. In this case the ODBC driver would act as a server for the application but as a client to the DBMS.

On top of OLE DB, the high-level ADO interface could be used. Since both OLE DB and ADO are based on Microsoft COM, they can be reached from almost any programming language.

For Java applications, the JDBC interface can be used. It builds on ODBC but has been developed with the same philosophy as the OLE DB and ADO interfaces. For databases that do not directly support JDBC, an ODBC driver can be used as intermediate layer, just like OLE DB. Pervasive.SQL uses the Btrieve interface for fast reading and updating transactions. Btrieve is very closely connected to the physical storage of data. In Btrieve data is written as records. A record can be described as a predefined set of data elements, similar to a *struct* in the C programming language. Btrieve is not aware of the data elements in the record, but treats data elements as string of bytes that can be retrieved or stored. An index key is attached to every record. Records can then be accessed in two ways, physical or logical. When a record is located logically, an index containing all keys is used to lookup the location of the data. The indexes are sorted in either ascending or descending order in a B-tree. Some methods to access data, except by keyword, are `get first`, `get last`,

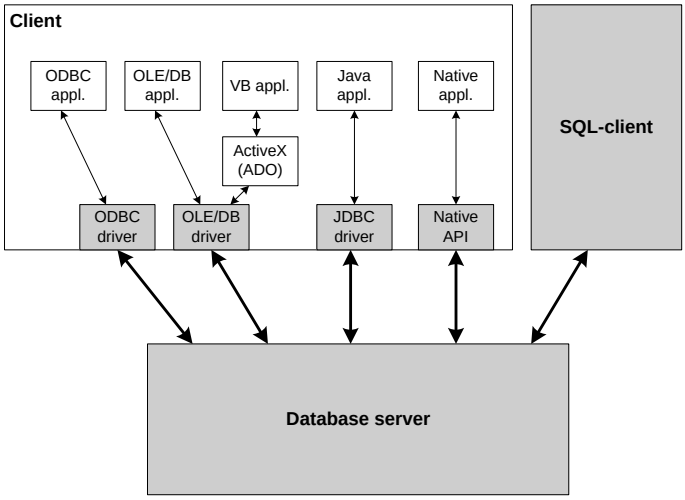


Figure 2.4: *Figure showing the architecture of applications that uses different types of database interfaces. The shaded boxes are components that normally are provided by the database vendor.*

DBMS system	C/C++	ODBC	OLE DB	ADO	JDBC	Java	Other
Pervasive.SQL		x	x	x	x		Btrieve RSI
Polyhedra	x	x	x	x	x		
Velocis	x	x				x	Perl
RDM	x						
Berkeley DB	x	x				x	TCL Perl
TimesTen					x		

Table 2.6: Interfaces supported by different embedded database systems.

DBMS system	Network connectivity
Pervasive.SQL	x
Polyhedra	x
Velocis	x
RDM	x
Berkeley DB	
TimesTen	x

Table 2.7: Network connectivity in investigated embedded database systems.

`get greater than`, etc. However for very fast access, the physical method can be used. Then the data is retrieved using a physical location. This technique is useful when data is accessed in a predefined order, this can be compared to the pointer network used by RDM discussed previously. Physical access is performed using four basic methods, `step first`, `step last`, `step next` and `step previous`.

One limitation with the Btrieve interface is that it is not relational. Pervasive has therefore introduced their RowSet interface (RSI) that combines some functionality from Btrieve and some from SQL. Every record is a row and the data elements in the records can be considered columns in the relation. This interface is available for the smart card version, embedded version, and mobile version only, and is not available for non-embedded versions of Pervasive.SQL.

The Btrieve and RSI interfaces are quite different from the native interface in Polyhedra, which does not require the same understanding of the physical data structure. The Polyhedra C/C++ interface uses SQL, thus operating on a higher level. Basically, three different ways to access the database are provided in the interface: queries, active queries, and transactions. A query performs a retrieval of data from the database immediately, using the SQL `select` command, and the query is thereafter deleted. An active query works the same way as an ordinary query but will not be deleted after the result has been delivered, but will be reactivated as soon as any of the involved data is changed. This will continue until the active query is explicitly deleted. One problem with active queries is that if data changes very fast, the number of activations of the query can be substantial, thus risking to overload the system. To prevent this a minimum inter-arrival time can be set, and thus the query cannot be reactivated until the time specified has elapsed. From a real-time perspective, the activation could then be treated as a periodic event, adding to the predictability of the system.

Using transactions is the only way to make updates to the database, queries are only for retrieval of data. A transaction can consist of a mixture of active update queries and direct SQL statements. Note that a query might update the database if it is placed inside of a transaction. Active update queries insert or update a single data element and also delete on single row. Transactions are, as always, treated as atomic operations and are aborted upon data conflict. To further speed up transaction execution time, SQL procedures can be used, both in queries and transactions. These procedures are compiled once and cannot be changed after that, only executed. This eliminates the compilation process and is very useful for queries that run often. However, procedures do not allow schema changes.

Noteworthy is that TimesTen only supports ODBC and JDBC, and it has no native interface like the rest of the systems. To increase the speed of ODBC connections, an optimized ODBC driver is developed that connects only to TimesTen. Like the Polyhedra DBMS, TimesTen uses precompiled queries, but in addition supports parameterized queries. A parameterized query is a precompiled query that supports arguments passed to it. For example the parameterized query

```
query( X )
    SELECT * from X
```

would return the table specified by `X`.

The Berkeley DB, as mentioned earlier, is not a client/server solution, but is implemented as an embedded library. It is, furthermore, not relational but uses, similar to the Btrieve interface, a key that relates to a record. The similarities to Btrieve go further than that; data can be accessed in two ways, by key or by physical location, just as Btrieve. The methods `get()` and `put()` access data by the key, while cursor methods can be used to get data by physical location in the database. As in Btrieve, the first, last, next, current, and previous record number can be retrieved. A cursor is simply a pointer that points directly to a record. The functionality and method names are similar to each other independent of which of the supported programming language that is used. There are three different modes that the database can run in:

DBMS system	Desktop OS				
	Windows	**UNIX**	**Linux**	**OS/2**	**DOS**
Pervasive.SQL for smart cards					
Pervasive.SQL for embedded systems					
Pervasive.SQL for mobile systems	x				
Polyhedra	x	x	x		
Velocis	x	x	x		
RDM	x	x	x	x	x
Berkeley DB	x	x	x		
TimesTen	x	x	x		

Table 2.8: Desktop operating systems supported by investigated embedded database systems.

- Single user DB. This version only allows one user to access the database at a time, thus, removing the need for concurrency control and transaction support.
- Concurrent DB. This version allows multiple users to access the database simultaneously, but does not support transactions. This database is suitable for systems that has very few updates but multiple read-only transactions.
- Transactional database. This version allows both concurrent users and transactions.

The RDM database, which is implemented as a library, has a C/C++ interface as its only interface. This library is, in contrast to Berkeley DB, relational. The interface is a low-level interface that does not comply with any of the interface standards. As mentioned earlier, one problem with databases that does not use the client/server model is the lack of standard interfaces. However, since RDM is relational, a separate SQL-like interface *dbquery* is developed. It is a superset of the standard library, which makes use of a subset of SQL.

2.3.9 Operating system platforms

Different embedded systems might run on various operating systems due to differences in the hardware environment. Also the nature of the application determines which operating systems that might be most useful. Thus, most embedded databases must be able to run on different operating system platforms.

DBMS system	Real-Time OS						Hand-held OS		Smart card OS		
	VxWorks	***OSE***	***pSOS***	***LynxOS***	***QNX***	***Phar-lap***	***Win CE***	***Palm OS***	***Java Card***	***Mult OS***	***Win^0***
Pervasive.SQL[1]									x	x	
Pervasive.SQL [2]	x				x	x					
Pervasive.SQL [3]							x	x			
Polyhedra	x	x	x	x							
Velocis											
RDM	x				x						
Berkeley DB	x										
TimesTen	x			x							

Table 2.9: Real-time and embedded operating systems supported by investigated embedded database systems.

Tables 2.8 and 2.9 give an overview of different operating systems supported by embedded databases we investigated. There are basically four categories of operating systems that investigated embedded databases support:

- Operating systems traditionally used by desktop computers. In this category you will find the most common operating systems like, Microsoft Windows, different versions of UNIX and Linux.
- Operating systems for hand-held computers. In this category you find Palm OS and Windows CE/PocketPC. These operating systems demand small memory requirements but still have most of the functionality of the ordinary desktop computer operating systems. A good interaction and interoperability between the operating systems on the hand-held computer and a standard desktop is also important. This is recognized by all databases, if you consider the complete Pervasive.SQL family as one database system.
- Real-time operating systems. In this category you find systems like VxWorks and QNX.
- Smart card operating systems. These systems, like Java Card and MultOS, are made for very small environments, typically no more than 32kb. The Java Card operating system is simply an extended Java virtual machine.

Most commercial embedded database systems in this survey support a real-time operating system (see table 2.9). Additionally, Pervasive.SQL supports operating systems for hand-held computers and smart card operating systems, in the Pervasive.SQL for mobile systems version and the Pervasive.SQL for smart card version, respectively.

2.3.10 Concurrency control

Concurrency control (CC) *serializes* transactions, retaining the *ACID properties*. All transactions in a database system must fulfill all four ACID properties. These are:

- Atomic: A transaction is indivisible, either it is run to completion or it is not run at all.
- Consistent: It must not violate logical constraints enforced by the system. For example, a bank transaction must follow the law of conservation of money. This means that after a money transfer, the sum of the receiver and the sender must be unchanged.
- Isolated: A transaction must not interfere with any other concurrent transaction. This is also referred to as serialization of transactions.
- Durable: A transaction is, once committed, written permanently into the database.

If two transactions that have some data elements in common is active at the same time, a data conflict might occur. Then it is up to the concurrency control to detect and resolve this conflict. This is most commonly done with some form of locking mechanism. It substitutes the semaphore guarding a global data in a real-time system. There are two fundamentally different approaches on achieving this serialization, an optimistic and a pessimistic approach.

The most common pessimistic algorithm is two-phase-locking (2PL) algorithm proposed in 1976 by Eswaran et al. [33]. This algorithm consists, as the name indicates, of

[0] Smart card operating system for Windows.
[1] Pervasive.SQL for smart cards.
[2] Pervasive.SQL for embedded systems.
[3] Pervasive.SQL for mobile systems.

DBMS system	Pessimistic CC	Optimistic CC	No CC
Pervasive.SQL	x		
Polyhedra		x	
Velocis	x		
RDM	x		
Berkeley DB	x		x
TimesTen	x		

Table 2.10: Concurrency control strategies used in different embedded database systems.

two phases. In the first phase all locks are collected, no reading or writing to data can be performed before a lock has been obtained. When all locks are collected and the updates have been done, the locks are released in the second phase.

Optimistic concurrency control (OCC) was first proposed by Kung and Robinson [58] in 1981. This strategy takes advantage of the fact that conflicts in general are rather rare. The basic idea is to read and update data without regarding possible conflicts. All updates are, however, done on temporary data. At commit-time a conflict detection is performed and the data is written permanently to the database if no conflict was detected. However the conflict detection (verification phase) and the update phase need to be atomic, implying some form of locking mechanism. Since these two phases take much shorter time than a whole transaction, locks that involves multiple data, or the whole database, can be used. Since it is an optimistic approach, performance degrades when congestion in the system increases.

For real-time databases a number of variants of these algorithms have been proposed that suit these databases better [119, 10]. These algorithms try to find a good balance between missed deadlines and temporally inconsistent transactions. Song and Liu showed that OCC algorithms performed poorly with respect to temporal consistency in real-time systems that consist of periodic activities [103], while they performed very well in systems where transactions had random parameters, e.g., event-driven systems. However, it has been shown that the strategies and the implementation of the locking and abortion algorithms significantly determine the performance of OCC [46]. All databases except the Polyhedra DBMS use pessimistic CC (see table 2.10). Since Polyhedra is an event-driven system, OCC is a natural choice. Furthermore, Polyhedra is a main-memory database with very fast execution of queries, the risk of a conflict is thus reduced.

Pervasive.SQL has two kind of transactions: exclusive and concurrent. An exclusive transaction locks a complete database file for the entire duration of the transaction, allowing only concurrent non-transactional clients to perform read-only operations on the file. A concurrent transaction, however, uses read and write locks with much finer granularity, e.g., page or single data locks.

The Berkeley DB has three configurations: (i) The non-concurrent configuration allows only one thread at a time to access the database, removing the need for concurrency control. (ii) The concurrent version allows concurrent readers and concurrent writers to access the database simultaneously. (iii) The concurrent transactional version allows for full transactional functionality with concurrency control, such as fine grain locking and database atomicity.

Similar to Berkeley DB, TimesTen also has a “no concurrency” option, in which only a single process can access the database. In addition, TimesTen supports two lock sizes: data-store level and record level locking.

DBMS system	Roll-forward	Roll-back	Journalling	No recovery
Pervasive.SQL	x			x
Polyhedra			x	x
Velocis	x			
RDM	x			
Berkeley DB		x		
TimesTen		x		x

Table 2.11: Strategies for recovery used in different embedded database systems.

2.3.11 Recovery

One important issue for databases is persistence, that is data written and committed to the database should remain until it is overwritten or explicitly removed, even if the system fails and have to be restarted. Furthermore, the state of all ongoing transactions must be saved to be able to restore the database to a consistent state upon recovery, since uncommitted data might have been written to the database. There is basically two different strategies for backup restoration: roll-back recovery, and roll-forward recovery, normally called "forward error recovery". During operation continuous backup points occurs where a consistent state of the database is stored on a non-volatile media, like a hard-drive. These backup points are called checkpoints.

In roll-back recovery you simply restart the database using the latest checkpoint, thus guaranteeing a consistent database state. The advantage with this approach is that it does not require a lot of overhead, but the disadvantage is that all changes made to the database after the checkpoint are lost.

When roll-forward recovery is used, checkpoints are stored regularly, but all intermediate events, like writes, commits and aborts are written to a log. In roll-forward recovery, the database is restored to the last checkpoint, and all log entries are performed in the same order as they have been entered into the log. When the database has been restored to the state it was at the time of the failure, uncommitted transactions are roll-backed. This approach requires more calculations at recovery-time, but will restore the database to the state it was in before the failure. Pervasive.SQL offers three different types of recovery mechanisms, as well as the option to ignore recovery (see table 2.11). This can also be selected parts of the database. Ignoring check-pointing for some data can be useful for systems where some data must be persistent while other data can be volatile. Going back to our example in section 1.1, the desired level given by the user interface, and the regulator parameters need to be persistent, while the current reading from the level indicator must not, since by the time the database is recovered the data will probably be stale. The second option is shadow-paging, which simply makes a copy of the database file and makes the changes there, and when the transaction is complete, the data is written back to the original page. A similar approach is the delta-paging, which creates an empty page for each transaction, and only writes the changes (delta-values) to it. At the end of a transaction the data is written back to the original, just as for shadow-paging. With both of these techniques, the database is consistent at all times. The last option is to use logging and roll-forward recovery.

The default configuration for Polyhedra is non-Jornalling, which means that no automatic backup is performed, but the application is responsible for saving the database to non-volatile memory. This is particularly important since this system is a main memory database, and if no backups is taken all data is, of course, lost. When journalling is used, the user can select which tables that should be persistent and the journaller will write an entry in the log when a transaction involving persistent data is committed. There are two approaches when data is persistently written, direct journalling and non-blocking journalling. The direct journalling approach writes data to persistent storage when the load on the DBMS is low. However, the database is blocked during this process and this can cause

problems for systems with time-critical data. The second approach can then be used, and that is to use a separate journalling process responsible for writing entries persistent.

TimesTen supports three different levels of recovery control. The most stringent guarantees transaction atomicity and durability upon recovery, in which case the transaction is not committed until the transaction is written onto disk. The second level of control guarantees transaction atomicity but not durability. Upon commit, the log entry is put into a queue and is later written to disk. Upon recovery the database will be consistent, but committed transactions that have not yet been written to disk might be lost. The lowest level of recovery control is no recovery control. Neither atomicity nor durability is guaranteed. This option might not be as inappropriate as it might seem at a first glance, since TimesTen is a main-memory database, often used in systems with data that would be stale upon recovery, e.g., a process controller.

2.3.12 Real-time properties

Even though none of the commercial database systems in this survey can be classified as a real-time database from a hard real-time systems perspective, most of the systems are successfully used in time-critical systems (to different extent). Ericsson uses Polyhedra in their 3G platform for wireless communication. Delcan Corporation uses Velocis DB in a traffic control system that handles about 1200 traffic lights simultaneously even though this database primarily is used in web and e-Commerce applications. The database systems are simply so fast and efficient and have so many options for fine tuning their performance that the application systems works, even though the DB systems cannot itself guarantee predictability.

A question one could ask is: How would we use one of these systems in a hard real-time system? Are there any ways to minimize the unpredictability to such a degree that a pessimistic estimation would fulfill hard real-time requirements? In our opinion it is. For an event triggered real-time system, Polyhedra would fit well. Let us use our example application again. When new sensor values arrive the I/O management, the database is updated. If there is a change in the alarm data the CL code that is connected to that data is generating an event to trigger the alarm task. Built into the system is also the minimum inter-arrival interval mentioned in the interfaces section. So by using the Polyhedra database to activate tasks, that will then be scheduled by a priority based real-time operating system, a good enough degree of predictability is achieved. To further increase guarantees that critical data will be updated is to use a so called watchdog, that activates a task if it has not been activated for a predefined time. The memory and size predictability would be no problem since they have specified the exact memory overhead for every type of object. However, temporal validity is not addressed with this approach.

For statically scheduled real-time systems the RDM “Network access” could be able to run with predictable response times. This because the order of the data accesses is known a priori and can therefore be linked together in a chain, and the record lookup index is bypassed.

2.4 Current state-of-the-art from research point of view

There exists a number of databases that could be classified as pure real-time databases. However these databases are research project and are not yet on the commercial market. We have selected a number of real-time database systems and compared them with respect to the following criteria:

- Real-time properties. The criteria enables us to discuss real-time aspects of the systems and how they are implemented.

- Distribution. The criteria enables us to talk about different aspects with respect to distributing the database.
- Transactions workload characteristics. The criteria enables us to discuss how the transaction system is implemented.
- Active behavior. The criteria enables us to talk about the active aspects for some of the systems.

The systems selected in this survey represent some of the more recent real-time database platforms developed. These systems are representative of the ongoing research within the field.

STRIP The STanford Real-time Information Processor (STRIP) [4] is a soft real-time database system developed at Stanford University, US. STRIP is built for the UNIX operating system, is distributed and uses streams for data sharing between nodes. It is an active database that uses SQL3-type rules.

DeeDS The Distributed active, real-time Database System (DeeDS) [9] supports both hard and soft real-time transactions. It is developed at the University of Skövde, Sweden. It uses extended ECA rules to achieve an active behavior with real-time properties. It is built to take advantage of a multiprocessor environment.

BeeHive The BeeHive system [106] is a virtual database developed at the University of Virginia, Charlottesville, US, in which the data in the database can be located in multiple locations and forms. The database supports four different interfaces namely, a real-time interface, a quality of service interface, a fault-tolerant interface, and a security interface.

REACH The REACH system [121], developed at the Technical University of Darmstadt, Germany, is an active object-oriented database implemented on top of the object-oriented database openOODB. Their goal is to archive active behavior in an OO database using ECA rules. A benchmarking mode is supported to measure execution times of critical transactions. The REACH system is intended for soft real-time systems.

RODAIN The RODAIN system [112] developed at the University of Helsinki, Finland, is a firm real-time database system that primarily is intended for telecommunication. It is designed for high degree of availability and fault-tolerance. It is tailored to fit the characteristics of telecommunication transactions identified as short queries and updates, and long massive updates.

ARTS-RTDB The ARTS-RTDB system [53], developed at the University of Virginia, Charlottesville, US, supports both soft and hard real-time transactions. It uses imprecise computing to ensure timeliness of transactions. It is built on top of the ARTS real-time operating system [114].

2.4.1 Real-time properties

STRIP

The STRIP [4] system is intended for soft real-time systems in which the ultimate goal is to make as many transactions as possible commit before their deadlines. It is developed for the UNIX operating system which might seem odd, but since STRIP is intended to run in open systems UNIX was selected. Even though UNIX is not a real-time operating system it has, when used in the right way, turned out to be a good operating system for soft

real-time systems since a real-time scheduling emulator can be placed on top of the UNIX scheduler which sets the priorities according to some real-time scheduling algorithm, a real-time behavior can be achieved. However, there have been some problems that needed to be overcome while implementing STRIP that is related to real-time scheduling in UNIX. For example to force a UNIX schedule not to adjust the priorities of processes as they are running. Non-real-time operating systems often have mechanisms to dynamically adjust the priorities of processes during run time to add to throughput and fairness in the system. By using Posix UNIX, a new class of processes whose priorities are not adjustable by the scheduler was provided. In [5] versions of the earliest deadline and least slack algorithms was emulated on top of a scheduler in a traditional operating system, and the results showed that earliest deadline using absolute deadline and least slack for relative deadline performed equal or better than their real-time counterparts, while the emulated earliest deadline for relative deadline missed more deadlines than the original EDF algorithm for low systems load. However, during high load the emulated algorithm outperformed the original EDF. In STRIP EDF, highest value first, highest density value first or a custom scheduling algorithm can be used to schedule transactions. To minimize unpredictability, the STRIP system is a main-memory database, thus removing disk I/O.

DeeDS

DeeDS [9] is intended for both hard and soft real-time systems. It is built for the OSE delta real-time operating system developed by ENEA DATA [30]. This distributed hard real-time operating system is designed for both embedded uniprocessor and multiprocessor systems. If used in a multiprocessor environment the CPUs are loosely coupled. A more detailed description of OSE delta can be found in [43]. The DeeDS system consists of two parts, one part that handles non critical system services and one that handles the critical. All critical systems services are executed on a dedicated processor to simplify overhead cost and increase the concurrency in the system. The DeeDS system is, as the STRIP system, a main memory database.

BeeHive

BeeHive utilizes the concept of data-deadline, forced-wait and the data deadline based scheduling algorithms [117]. These concepts assure that transactions are kept temporally consistent by assigning a more stringent deadline to a transaction based on the maximum allowed age of the data elements that are used in the transaction, thus the name data deadline. The key concept of forced wait is to, if a transaction cannot complete before its data deadline, postpone it until data elements get updated, thus giving the transaction a later data deadline. These transactions can then be scheduled using, for example, the earliest data-deadline first (EDDF) or data deadline based least slack first (DDLSF) algorithms [117].

For real-time data storage a four level memory hierarchy is assumed: main memory, non-volatile RAM (e.g., Flash), persistent disk storage, and archival storage (e.g., tape storage) [106]. The main memory is used to store data that is currently in use by the system. The non-volatile RAM is used as a disk-cache for data or logs that are not yet written to disk. Furthermore, system data structures like lock tables and rule bases can be stored in non-volatile RAM. The disk is used to store the database, just like any disk-based database. Finally, the tape storage is used to make backups of the system. By using these four levels of memory management a higher degree of predictability can be achieved than for entirely disk-based databases, since the behavior can be somewhat like a main-memory database even though it is a disk-based system.

REACH

REACH is built on top of Texas Instruments openOODB system, which is an open database system. By an open system we refer to a system that can be controlled, modified, and partly extended by developers and researcher [116]. The OpenOODB system is by itself not a real-time database system, but an object-oriented database. Some efforts has been done in the REACH project to aid the user with respect to system predictability [18] such as milestones, contingency plans, a benchmarking tool and a trace tool.

Milestones are used to monitor the progress of transactions. If a milestone is missed, the transaction can be aborted and a contingency plan is released instead. This contingency plan should be implemented to handle a missed deadline situation by either degrading the system in a safe way, or produce a good-enough result before the original transactions deadline. One could say that a contingency plan in cooperation with milestones works as a watch-dog that is activated when a deadline is about to be missed.

The REACH system has a benchmarking tool that can be used to measure execution times for method invocations and event triggering. The REACH trace tool can trace the execution order in the system. If the benchmarking tool is used together with the trace tool, system behavior and timeliness could be determined.

RODAIN

The RODAIN system is a real-time database system intended for telecommunication applications. The telecommunication environment is a dynamic and complex environment that deals with both temporal data and non-temporal data. A database in such a system must, support both soft and firm real-time transactions [112]. However, the believe is that hard real-time databases will not be used in telecommunication in the near future since they generally are to too expensive to develop or use to suit the market [112].

One of the key issues for RODAIN is availability. Therefore a fault-tolerant system is necessary. The RODAIN system is designed to have two nodes, thus giving full database replication. This is especially important since the database is a main-memory system. It uses a primary and a mirror node. The primary database sends logs to the mirror database which in turn acknowledges all calls. The communication between the nodes is assumed to be reliable and have a bounded message transfer delay. A watchdog monitor keeps track of any failing subsystem and can instantly swap the primary node and the mirror node in case of a failure, see figure 2.5. The failed node always recover as a mirror node and loads the database image from permanent storage. Furthermore, the User Request Interpreter system (URIS) can keep track of all ongoing transaction and take appropriate actions if a transaction fails. Since both the primary and the mirror node has a URIS no active transactions will be lost if a failure in the primary system leads to a node swap between the primary and the mirror. Figure 2.5 shows the architecture of RODAIN. The subsystem OODBMS handles all traditional database activities, like concurrency control, physical storage, schema management etc. There are three interfaces to the database, the traditional user interface, referred to as the Applications interface, the DBMS nodes interface, and the Mirror interface. Noteworthy is the distinction between mirror nodes and distributed DBMS nodes. The mirror nodes purpose is to ensure a fault-tolerant running, while the distribution nodes are used for normal database distribution.

ARTS-RTDB

The relational database ARTS-RTDB [53] incorporates an interesting feature called imprecise computing. If a query, when its deadline expires, is not finished, the result so far can be returned to the client if the result can be considered meaningful. Take a query that calculates the average value of hundreds of values in a relation column. If an adequate amount of values has been calculated at the time of deadline, the result could be considered meaningful but imprecise and it is therefore returned to the client anyway.

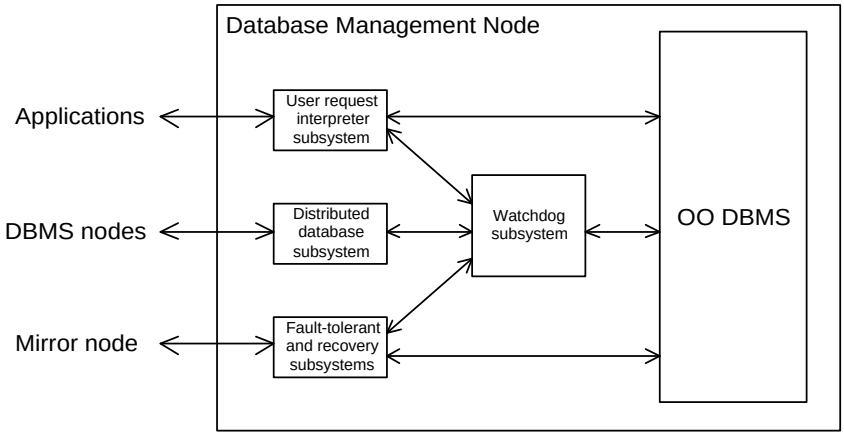


Figure 2.5: *RODAIN DBMS node. Figure from [112]*

ARTS-RTDB is built on top of the distributed real-time operating system ARTS, developed by Carnegie Mellon University [114]. ARTS schedules tasks according to a time-varying value function, which specifies both criticality and semantics importance. It does support both soft and hard real-time tasks.

In ARTS-RTDB the most critical data operations has been identified to be the INSERT, DELETE, UPDATE and SELECT operations. Efforts has therefore been made to optimize the system to increase the efficiency for those four operations. According to [53] many real-time applications almost only use these operations at run-time.

2.4.2 Distribution

To increase concurrency in the system, distributed databases can be used. Distributed in a sense that the database copies reside on different computer nodes in a network. If the data is fully replicated over all nodes, applications can access any node to retrieve a data element. Unfortunately, maintaining different database nodes consistent with each other is not an easy task, especially if timeliness is crucial. One might have to trade consistency for timeliness. A mechanism that continuously tries to keep the database nodes as consistent with each other as possible is needed. Since the system is distributed all updates to a database node must be propagated via message passing, or similar, thus adding significantly to the database overhead because of the added amount of synchronization transactions in the system. One of the most critical moments for a transaction deployed in a distributed database is the point of commit. At this point all the involved nodes must agree upon committing the transaction or not.

One of the most commonly used commit protocols for distributed databases is the two-phase commit protocol [39]. As the name of the algorithm suggests the algorithm consists of two phases, the prepare-phase and the commit-phase. In the prepare phase, the coordinator for the transaction sends a `prepare` message to all other nodes and then waits for all answers. The nodes then answers with a `ready` message if they can accept the commit. If the coordinator receives `ready` from all nodes a `commit` message is broadcasted, otherwise the transaction is aborted. The coordinator then finally waits for a `finished` message from all nodes to confirm the commit. Hereby is consistency guaranteed between nodes in the database.

DeeDS

The DeeDS [9] system has a less strict criteria for consistency, in order to enforce timeliness. They guarantee that each node has a local consistency, while the distributed database might be inconsistent due to different views of the system on the different nodes. This approach might be suitable for systems that mostly rely on data that is gathered locally, but sometimes uses data imported from other subsystems. Consider an engine that has all engine data, e.g., ignition control, engine speed and fuel injection, stored in a local node of a distributed database. The timeliness of these local data items are essential in order to run the engine. To further increase the efficiency of the engine, remotely gathered data, like data from the transmission box, with less critical timing requirements can be distributed to the local node.

STRIP

The concept of streams communicating between nodes in a distributed system has been recognized in STRIP [4]. Nodes can stream views or tables to each other on a regular basis. The user can select if whole tables/views or delta tables, which only reflects changes to the tables/views, should be streamed. Furthermore it is selectable if the data should be streamed, periodically when some data has reached a predefined age or only after an explicit instruction to stream. In figure 2.6 we can see the process architecture of STRIP. The middle part of the figure shows the execution process, the transaction queue, the result queue and the request process. This part makes the query layer of the database and can be compared to an ordinary database that processes transactions and queries from the application. Remote application addresses queries or transactions to the requesting process, which in turn places the transaction in the transaction queue. However, local applications can access the transaction queue directly via a client library, thus minimizing overhead. Each execution processes can execute a transaction from the transaction queue or an update transaction from one of the update queues. The update queues are fed from the incoming update streams. When a table/view needs to be sent to an outgoing stream, one of the execution processes enqueues it to the update queue read by the export process. It has been shown that when the update frequency is low, the import and export processes can run at a high priority without significantly affecting transaction response times but when the frequency increases the need for a different update scheduling policy is necessary [4], see section 2.4.3.

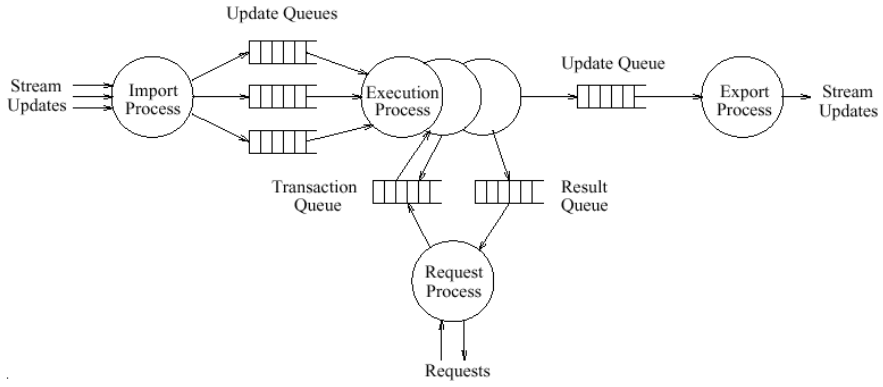


Figure 2.6: *The process architecture of the STRIP system. Figure from [4].*

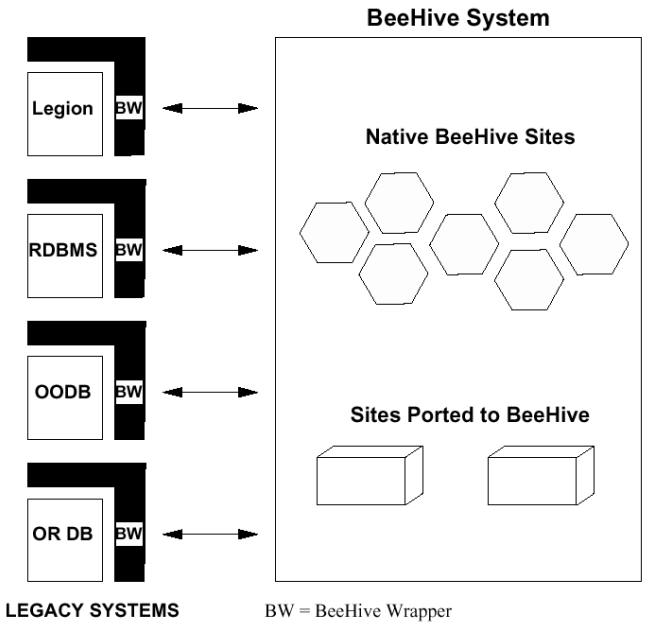


Figure 2.7: *The BeeHive virtual database. Figure from [106]*

BeeHive

Unlike most databases BeeHive is a virtual database, in a sense that in addition to its own data storage facility it can use external databases and incorporate them as one database. These databases can be located either locally or on a network, e.g., the Internet.

In [40], a virtual database is defined as a database that organizes data scattered through the Internet into a queriable database. Consider a web service that compares prices of products on the Internet. When you enter the database you can search for a specific product, and the service will return to you a list of Internet retailers and to what price they are selling the product. The database system consists of four parts [40]:

- Wrappers. A wrapper is placed between a web-page or database and the virtual database. In the web-page case, the wrapper converts the textual content of the web-page into a relational format understandable by the virtual database. When using a wrapper in the database case, the wrapper converts the data received from the database into the virtual database format. Such a wrapper can be implemented using a high level description language, e.g., Java applets.
- Extractors. The relations received from the wrapper most often contains unstructured textual data in which the interesting data needs to be extracted.
- Mappers. The mapper maps extracted relations into a common database schema.
- Publisher. The publisher is used to present the database to the user, it has similar functionality as a traditional database.

BeeHive can communicate with other virtual databases, web-browsers or independent databases using wrappers. These wrappers are Java applets and may use the JDBC standard, see figure 2.7.

RODAIN

RODAIN system supports data distribution. The nodes can be anything between fully replicated to completely disjoint. Their belief is that only a few requests in telecommunications need access to more than one database node and that the request distributions among the databases in the system can be arranged to be almost uniform [112].

ARTS-RTDB

ARTS-RTDB has been extended to support distribution. The database nodes use a shared file which contains information that binds all relations to the server responsible for a particular relation. Since the file is shared between the nodes, it is to be treated as a shared resource and must therefore be accessed using a semaphore. It is believed that if relations are uniformly distributed between the nodes and if no hot-spot relations exist, an increase in performance will be seen. A hot-spot relation is a relation that many transactions use, and such a relation can lead to a performance bottleneck in the system.

2.4.3 Transaction workload characteristics

DeeDS

DeeDS supports both sporadic (event-triggered) and periodic transactions. There are two classes of transactions: critical (hard transactions) and non-critical (soft transactions). To ensure that a deadline is kept for a critical transaction, milestone monitoring and contingency plans are used. A milestone can be seen as a deadline for a part of the transaction. If a milestone is passed, it is clear that the transaction will not make its deadline, a contingency plan can be activated and the transaction is aborted. A contingency plan should, if implemented correctly, deal with the consequences of a transaction aborting. One way to compute less accurate result and present it in time, similar to imprecise computing used by ARTS-RTDB (see section 2.4.1). Milestones and contingency plans are discussed further in [32]. The timing requirements for a transaction is passed to the system as a parameterized value function, this approach reduces the computational cost and the storage requirements of the system.

The scheduling of transactions is made online in two steps. First, a sufficient schedule is produced. This schedule meets the minimum requirements, for example all hard deadlines and 90 percent of the soft deadlines are met. Secondly, the event monitors worst-case execution time is subtracted from the remaining allowed time for the scheduler during this time slot, and this time is used to refine and optimize the transaction schedule [73].

STRIP

STRIP supports firm deadlines on transactions [7], thus when a transaction misses its deadline it will be aborted. There are two classes of transactions in STRIP, low value and high value transactions. When transactions are scheduled for execution is determined by transaction class, value density, and choice of scheduling algorithm. The value density is the ratio between the transactions value and the remaining processing time. The scheduling of transactions is in competition with the scheduling of updates, therefore updates also have the same properties as transactions with respect to classes, values, and value density. STRIP has four scheduling algorithms, Updates First (UF), Transactions First (TF), Split Updates (SU), and Apply Updates on Demand (OD).

- Updates First schedules all updates before transactions, regardless of the value density of the transactions. It is selectable if a running transaction is preempted by an update of if it should be allowed to commit before the update is executed. For a system with a high load of updates this policy can cause long and unpredictable execution times for transactions. However, for systems where consistency between nodes

and where temporal data consistency is more important than transaction throughput, this can be a suitable algorithm. Further, this algorithm is also suitable for systems where updates are prioritized over transactions. Consider a database running stock exchange transactions. A change in price for a stock must be performed before any pending transactions in order to get the correct value of the transaction.

- Transactions First is the opposite of update first. Transactions are always scheduled in favor of updates. However a transactions can not preempt a running update. This because updates most often have short execution time compared to transactions. This algorithm suits systems that values throughput of transactions higher than temporal consistency between the nodes. Transactions first might be useful in a industrial controlling system. If, in an overloaded situation, the most recent version of a data element is not available, the system might gain in performance from being able to run its control algorithms using an older value.

- Split Updates make use of the different classes of updates. It schedules updates and transactions according to the following order: high value updates, transactions, and low value updates. This algorithm combines the two previous algorithms and would suit a system where one part of the data elements is crucial with respect to temporal consistency, at the same time as transaction throughput is important.

- Apply Updates on Demand execute transactions before updates, but with the difference that when a transaction encounters stale data, the update queue is scanned for an update that is waiting to update the stale data element. This approach would enable a high throughput of transactions with a high degree of temporal consistency. However, calculating the execution time for a transaction is more difficult since the worst case is that all data elements used in the transaction has pending updates. Applying updates on demand resembles about the ideas of triggered updates [8].

Since transaction deadlines are firm, thus transactions are valueless after their deadline has expired, a mechanism called feasible deadlines can be used for transactions. The mechanism aborts transactions that has no chance of committing before their deadline.

BeeHive

The fact that the database is virtual and may consist of many different kinds of underlying databases is transparent. The user accesses the database through at least one of the four interfaces provided by BeeHive: FT API, RT API, Security API, and QoS API (see figure 2.8). FT API is the fault-tolerant interface. Transaction created with this interface will have some protection against processor failures and transient faults due to power glitches, software bugs, race conditions, and timing faults. RT API is the real-time interface, which enables the user to specify time constraints and typical real-time parameters along with transactions. The security interface can be used by application that for instance demands authorization for users and applications. Furthermore encryption can be supported. The Quality of Service (QoS) interface is primarily used for multimedia transactions. The user can specify transactions demands on quality of service.

When a transaction arrives from any interface, it is sent to the service mapper, which transforms it to a common transaction format used for all transactions in the system, regardless of its type, e.g., real-time transaction. It is then passed on to the resource planner, which determines which resources that will be needed. Furthermore, it is passed to the admission controller, which in cooperation with the resource planner, decides if the system can provide a satisfactory service for the transaction. If that is the case, it is sent to the resource allocation module, which globally optimizes the use of resources in the system. Finally the transaction is sent to the correct resource, e.g., a network or the operating system etc.

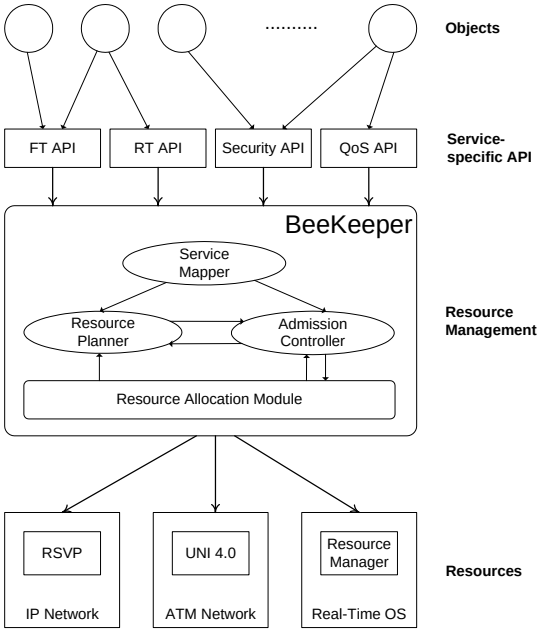


Figure 2.8: *The resource manager in BeeHive. Figure from [106]*

RODAIN

Five different areas for databases in telecommunication is identified [95]:

1. Retrieval of persistent customer data.
2. Modifications of persistent customer data.
3. Authorization of customers, e.g., PIN codes.
4. Sequential log writing.
5. Mass calling and Tele voting. This can be performed in large blocks.

>From these five areas, three different types of transactions can be derived [78]: short simple queries, simple updates, and long massive updates. Short simple queries are used when retrieving customer data and authorizing customers.Simple updates are used when modifying customer data and writing logs. Long massive updates are used when tele voting and mass calling is performed.

The concurrency control in RODAIN supports different kinds of serialization protocols. One protocol is the τ -serialization in which a transaction may be allowed to read old data as long as the update is not older than a predefined time [112].

Apart from a transactions deadline, an isolation level can also be assigned to a transaction. A transaction running on a low isolation level accepts that transactions running on a higher isolation level are accessing locked objects. However, it cannot access objects belonging to a transaction with a high degree of isolation.

ARTS-RTDB

The RTDB uses a pessimistic concurrency control, strict two phase locking with priority abort [118]. This means that as soon as a higher prioritized transactions wants a lock that is owned by a transaction with a low priority, the low level transaction is aborted. To avoid

the costly process of rolling back the aborted transaction, all data writing is performed on copies. At the point of commit, the transaction asks the lock-manager if a commit can be allowed. If this is the case, the transaction invokes a subsystem that writes all data into the database.

2.4.4 Active databases

Active databases can perform actions not explicitly requested from the application or environment. Consider a data element z, which is derived from two other data elements, x and y. The traditional way to keep z updated would be to periodically poll the database to determine if x or y have been altered. If that is the case, a new value of z would be calculated and the result written back to the database. Periodic polling is often performed at a high cost with respect to computational cost and will increase the complexity of the application task schedule. A different approach would be to introduce a trigger that would cause an event as soon as either x or y has been changed. The event would in its turn start a transaction or similar that would update z. This functionality would be useful for other purposes than to update derived data elements. It could, for example, be used to enforce boundaries of data elements. If a data element, e.g., data that represents a desired water level in our example application in figure 1.1, has a maximum and minimum allowed value than this restriction can be applied in the database instead of in the application. An attempt to set an invalid value could result in some kind of action, e.g., an error exception or a correction of the value to the closest valid value. By applying this restriction in the database it is abstracted away from the application, thus reducing the risk of programming errors or inconsistency between transactions that use this data element.

One way to handle such active behavior is through the use of ECA rules [34], where ECA is an abbreviation of Event-Condition-Action-rules, see figure 2.9. The events in an

```
ON <event E>
   IF <condition C>
      THEN <action A>
```

Figure 2.9: *The structure of an ECA rule.*

active database need to be processed by an event manager, which sends them to the rule manager. The rule manager in its turn locates all ECA rules that are affected by the newly arrived event and checks their conditions. For those ECA rules whose conditions are true, the action is activated. There are, however, three distinct approaches on when and how to execute the actions [32]:

- Immediate: When an event arrives, the active transaction is suspended and condition evaluation and possibly action execution is done immediately. Upon rule completion, the transaction is resumed. This approach affects the response time of the transaction.
- Deferred: When an event arrives, the active transaction is run to completion and then condition evaluation and possibly action execution is done. This approach does not affect the response time of the triggering transaction.
- Decoupled: When using this approach, condition evaluation and possibly action execution are performed in one or several separate transactions. This has the advantage that these updates will be logged even if the triggering transaction aborts.

One disadvantage with active databases is that the predictability, with respect to response times, is decreased in the system. This is because of the execution of the rules. For immediate or deferred rules the response time of the triggering transaction is prolonged because of the execution of rules prior to its point of commit. To be able to calculate a

response time for such a transaction, the execution time of all possible rules that the transaction can invoke. These rules can in their turn activate rules, so called cascading, which can create arbitrary complex calculations. When decoupled rules are used, transactions can generate new transactions in the system. These transactions and their priorities, response times, and possible rule invocations must be taken in account when calculating transaction response time, resulting in very pessimistic worst-case-execution times.

Some extensions to ECA rules which incorporates time constraints have been studied in [51]. For a more detailed description about ECA rules, active databases and active real-time databases see [32, 101]

STRIP

The STRIP rule system checks for events at each transaction, T, commit-time and for all events whose condition is true a new transaction T' is created that executes the execution clause of the rule. Thus, the STRIP system uses a decoupled approach. T' is activated for execution after the triggering transaction has committed. By default T' is released immediately after commit of the triggering transaction, but it can be delayed using the optional `after` statement, see figure 2.10.

```
WHEN <eventlist>
   [IF <condition>]
   THEN
      EXECUTE <action>
      [AFTER <time-value>]
```

Figure 2.10: *A simplified structure of a STRIP-rule. The optional `AFTER` -clause enables for batching of several rule-executions in order to decrease computational costs.*

The `after` statement enables all rule execution clauses, activated during the time window between rule evaluation and the execution clause specified by the `after` statement, to be batched together. If multiple updates on a single data element exists in the batch, only the ones necessary to derive the resulting value of that data element is executed, thus saving computational cost [6]. To have the execution clause in a separate transaction simplifies rule processing. The triggering transaction can commit regardless of the execution of the executions clauses, since the condition clause of the rules cannot alter the database state in anyway. Possible side-effects of the rules are then dealt with in a separate transaction. One disadvantage with decoupled execution clauses is that the condition results and transition data are deleted when the triggering transaction commits, thus, data is not available for the the triggered transaction. This has been solved in STRIP by the `bind as` statement. This statement binds a created table to a relation that can be accessed by the execution clause.

DeeDS

Since DeeDS supports hard real-time systems the rule processing must be designed to retain as much predictability with respect to timeliness as possible. This is achieved by restricting rule processing. Cascading, for example, is limited so that unbounded rule triggering is avoided when a rule is executed. The user can define a maximum level of the cascade depth. If the cascading goes deeper than that an exception is raised. Furthermore, condition evaluation is restricted to logical expressions on events and object parameters, and method invocations. DeeDS system an extended ECA rules [73] which also allows for specifying timing constraints to rules, e.g., deadlines.

Event monitoring has also been designed for predictability. Both synchronous and asynchronous monitoring is available. In synchronous mode the event monitor is invoked

	Sequential	Parallel
Independent	T_r independent of Tt	
Dependent	T_r cannot *start* until Tt *commits*, otherwise it is *discarded*.	T_r cannot *complete* until Tt *commits*, otherwise it is *aborted*.
Exclusive	T_r cannot *start* until Tt *aborts*, otherwise it is *discarded*.	T_r cannot *complete* until Tt *aborts*, otherwise it is *aborted*.

Table 2.12: *Table showing all decoupled execution modes supported by the DeeDS system. Tt is the triggering transaction and T_r is the rule transaction. Table from [31].*

on time-triggered basis and results have shown that throughput is higher than if the asynchronous (event-triggered) event monitoring is used, but at the cost of longer minimum event delays. When synchronous mode is used, event bursts will not affect the system in an unpredictable way [73]. In DeeDS, the immediate and deferred coupling modes are not allowed. Instead when a rule is triggered and the condition states that the rule should be executed, a new transaction is created, which is immediately suspended. If a contingency plan exists for this rule, a contingency transaction is also created and suspended. All combinations of decoupled execution is supported in DeeDS [31], see table 2.12. Depending on which mode that is selected, the scheduler activates the rule transactions according to the mode. The exclusive mode is used to schedule contingency plans.

REACH

REACH allows ECA rules to be triggered in all modes, immediate, deferred and decoupled. Decoupled rules might be dependent or independent, furthermore they can be executed in parallel, sequential, or exclusive mode. The exclusive mode is intended for contingency plans. The ECA rules used in REACH have no real-time extensions.

There are however some restrictions on when to use the different coupling modes. Rules trigger by a single method event can be executed in any coupling mode, while a rule triggered by a composite event only cannot be run in immediate mode. This is because if the events that make up composite event originate in multiple transactions, no identification of the spawning transactions is possible [121]. Rules that are invoked by temporal events (time-triggered) can only be run in an independent decoupled mode, since they are not triggered by a transaction [19]j.

To simplify the creation of the the rule-base a tool, GRANT, has been developed. This graphical tool, prompts the user for necessary information, provides certain default choices, creates C++ language structures and maps rules to C functions stored in a shared library [18].

The event manager in REACH consumes events according to two policies: chronological or most recent. The chronological approach executes all events in the order they arrive while the most recent strategy only executes the most recent instance of every event. What policy to use is dependent of the semantics of the application. Consider the example with the stock market database again. If, in an overloaded situation, the database is not able to immediately execute all stock price updates, only the most recent update of a stock is interesting, since this update contains the current price of this stock. For a system like this, the most recent approach would be sufficient, while in a telephone billing system it might not. In a system like that all updates to the database consist of a value to add to the existing record, therefore most recent is not sufficient here.

2.5 Observations

The databases presented represent a set of systems that basically have the same purpose, i.e., to efficiently store information in embedded systems or real-time systems. They mainly consist of the same subsystems, e.g., interfaces, index system and concurrency control system. But yet the systems behave so differently. Some systems that are primarily event driven, and on the other hand systems that might suit well for static scheduling. Some systems are relational while other are object-oriented. Some systems are intended for a specific application, e.g., telecommunication, process control or multimedia application.

Furthermore, an interesting observation can be made about these systems. The research platforms focus more on functionality while the commercial systems are more interested in user friendliness, adaptability, and standards compliance. While the research platforms have detailed specification on new internal mechanisms that improve performance under certain circumstances, like new concurrency controls or scheduling policies, the commercial systems supports multiple interfaces to ease integration with the application and supports standards like ODBC, OLE/DB and JDBC. Looking back at the criteria mentioned by ABB in section 2.2.2, we can see that most of the criteria are fulfilled by the commercial products. To be able to combine these criteria with the technology delivered by the research platforms would be a contribution.

One important issue that arises is: What embedded database system should one choose for a specific type of application? The task of choosing the best database system can be a long and costly process and probably not without compromises. A different approach would be to have a more generic database that can be customized, possibly with aid of a tool, to fit a specific application. This will be further discussed in the next chapter.

Chapter 3

Component-based systems

Business enterprises and society in general are becoming increasingly dependent on computer systems. Almost every application domain is faced with the challenge of ensuring that data in the system is managed in a uniform manner and stored efficiently by the system, e.g., telecommunications, e-commerce, business enterprise, vehicle industry, embedded and mobile systems. As a result, database systems are used in a vide variety of application areas, and are required to provide efficient management of heterogeneous data of all shapes and sizes.

Database industry is today, more than ever, faced with requirements for rapid application development and deployment, delivery of data in the right form to the right place at the right time and, most importantly, minimized complexity for both end users and developers. Rapid development with low costs, high quality and reliability, that minimize system complexity can be achieved if component-based software development paradigm is introduced in database development. This has been recognized and exploited by some major database vendors, e.g., Oracle, IBM, Informix and Sybase have all developed extensible database systems, which allow adding non-standard features and functionality to the extensible database system. However, existing component-based database solutions, mainly focus on solving one problem, e.g., the Microsoft Universal Data Access architecture provides uniform manipulation of data from different data stores, the extensible Oracle8*i* server enables manipulation of non-standard data types, etc.

It has been recognized that embedded and real-time systems could benefit from component-based development as well. Generally, by having well-defined reusable components as building blocks, not only development costs are reduced, but more importantly, verification and certification of components will only need to be done once. However, to efficiently reuse components for composing different systems, good composition rules for the system assembly must be provided, ensuring reliability of the composed system. Existing component-based embedded and real-time systems are mostly focused on ways of preserving real-time behavior of the system and/or enabling use of components in low-resource systems. A database system that can be tailored for different applications such that it provides necessary functionality but requires minimum memory footprint, and is highly integrated with the system would most likely be preferable solution for databases in embedded real-time systems. However, databases for real-time and embedded systems supporting fast development through reuse of components, do not exist, which makes these application domains deprived of the benefits that component-based development could bring. Thus, in this chapter we identify and contrast major issues in two component-based systems, database and embedded real-time, and inspect possibilities of combining these two areas.

Component-based systems can be analyzed only if certain knowledge of general component-based software engineering and its concepts such as components, architecture, and reuse are understood. Therefore, basic concepts of component-based software engi-

neering are introduced in section 3.1, and they are necessary in order to identify issues in different component-based systems and to obtain uniform criteria for surveying existing component-based solutions in both systems. In section 3.2 we review components and component-based solutions in database systems. This is followed by real-time components and solutions in embedded real-time systems in section 3.3. Section 3.4 concludes this chapter with a tabular overview of investigated component-based systems and their characteristics.

3.1 Component-based software development

The need for transition from monolithic to open and flexible systems has emerged due to problems in traditional software development, such as high development costs, inadequate support for long-term maintenance and system evolution, and often unsatisfactory quality of software [17]. Component-based software engineering (CBSE) is an emerging development paradigm that enables this transition by allowing systems to be assembled from a pre-defined set of components explicitly developed for multiple usage. Developing systems out of existing components offers many advantages to developers and users. In component-based systems [17, 24, 25, 36]:

- Development costs are significantly decreased because systems are built by simply plugging in existing components.
- System evolution is eased because system built on CBSE concepts is open to changes and extensions, i.e., components with new functionality can be plugged into an existing system.
- Quality of software is increased since it is assumed that components are previously tested in different contexts and have validated behavior at their interfaces. Hence, validation efforts in these systems have to primarily concentrate on validation of the architectural design.
- Time-to-market is shortened since systems do not have to be developed from scratch.
- Maintenance costs are reduced since components are designed to be carried through different applications and changes in a component are, therefore, beneficial to multiple systems.

As a result, efficiency in the development for the software vendor is improved and flexibility of delivered product is enhanced for the user [60].

Component-based development also raises many challenging problems, such as [60]:

- Building good reusable components. This is not an easy task and a significant effort must be used to produce a component that can be used in different software systems. In particular, components must be tested and verified to be eligible for reuse.
- Composing a reliable system out of components. A system built out of components is in risk of being unreliable if inadequate components are used for the system assembly. The same problem arises when a new component needs to be integrated into an existing system.
- Verification of reusable components. Components are developed to be reused in many different systems, which makes the component verification a significant challenge. For every component use, the developer of a new component-based system must be able to verify the component, i.e., determine if the particular component meets the needs of the system under construction.

- Dynamic and on-line configuration of components. Components can be upgraded and introduced at run-time; this affects the configuration of the complete system and it is important to keep track of changes introduced in the system.

3.1.1 Software component

Software components are the core of CBSE. However, different definitions and interpretations of a component exist. In a software architecture, a component is considered to be a unit of composition with explicitly specified interfaces and quality attributes, e.g., performance, real-time, reliability [17]. In systems where COM is used for a component framework, a component is generally assumed to be a self-contained, configurable binary package with precisely defined standardized interfaces [77]. A component can be also viewed as a software artifact that models and implements a well-defined set of functions, and has well-defined component interface (which do not have to be standard interfaces) and component implementation [28].

Hence, there is no common definition of a component for every component-based system. The definition of a component clearly depends on the implementation, architectural assumptions, and the way the component is to be reused in the system. However, all component-based systems have one common fact, and that is: *components are for composition* [111].

While frameworks and standards for components today primarily focus on CORBA, COM or JavaBeans, the need for component-based development in has been identified in the area of operating systems. The aim is to facilitate OS evolution without endangering legacy applications and provide better support of distributed applications [74].

Common for all types of components, independent of their definition, is that they communicate with its environment through well-defined interfaces, e.g., in COM and CORBA interfaces are defined in an interface definition language (IDL), Microsoft IDL and CORBA IDL. Components can have more than one interface. For example, a component may have three types of interfaces: provided, required, and configuration interface [17]. Provided and required interfaces are intended for the interaction with other components, whereas configuration interfaces are intended for use by the user of the component, i.e., software engineer (developer) who is constructing an application out of reusable components. Each interface provided by a component is, upon instantiation of the component, bound to one or more interfaces required by other components. The component providing an interface may service multiple components, i.e., there can be one-to-many relation between provided and required interfaces. When using components in an application there might be syntactic mismatch between provided and required interfaces, even when the semantics of the interfaces match. This requires adaptation of one or both of the components or an adapting connector to be used between components to perform the translation between components (see figure 3.1).

Independently of application area, a software component is normally considered to have black box properties [36, 28]: each component sees only interfaces to other components; internal state and attributes of the component are strongly encapsulated.

Every component implements some field of functionality, i.e., a domain [17]. Domains can be hierarchically decomposed into lower-level domains, e.g., the domain of communication protocols can be decomposed into several layers of protocol domains as in the OSI model. This means that components can also be organized hierarchically, i.e., a component can be composed out of subcomponents. In this context, two conflicting forces need to be balanced when designing a component. First, small components cover small domains and are likely to be reused, as it is likely that such component would not contain large parts of functionality unneeded by the system. Second, large components give more leverage than small components when reused, since choosing the large component for the software system would reduce the cost associated with the effort required to find the component, analyze its suitability for a certain software product, etc [17]. Hence, when designing a

Performance	The systems capacity of handling data or events.
Reliability	The probability of system working correctly over a given period of time.
Safety	The property of the system that will not endanger human life or the environment.
Temporal constraints	The real-time attributes such as deadlines, jitter, response time, worst case execution time, etc.
Security	The ability of software system to resist malicious intended actions.
Availability	The probability of a system functioning correctly at any given time.

Table 3.1: Functional quality attributes

component, a designer should find the balance between these two conflicting forces, as well as actual demands of the system in the area of component application.

3.1.2 Software architecture

Every software system has an architecture, although it may not be explicitly modeled [72]. The software architecture represents a high level of abstraction where a system is described as a collection of interacting components [3]. A commonly used definition of a software architecture is [11]:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationship among them.

Thus, the software architecture enables decomposition of the system into well-defined components and their interconnections, and consequently provides means for building complex software systems [72]. Design of the software architecture is the first step in the design process of a software product, right after the specification of the system's requirements. Hence, it allows early system testing, that is, as pointed out by Bosch [17] assessment of design and quality attributes of a system. Quality attributes of a system are those that are relevant from the software engineering perspective, e.g., maintainability, reusability, and those that represent quality of the system in operation, e.g., performance, reliability, robustness, fault-tolerance. These quality attributes of a system could be further classified as functional (see table 3.1) and non-functional quality attributes (see table 3.2), and used for architectural analysis [115]. The software architectural design process results in compo-

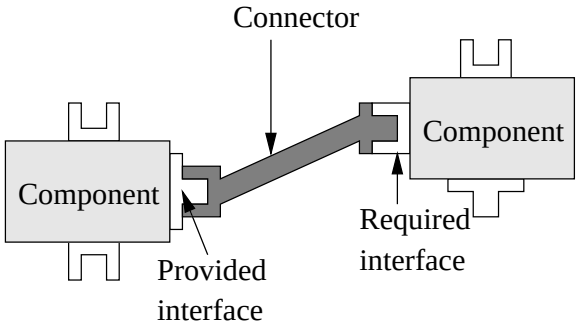


Figure 3.1: Components and interfaces

Testability	The ability to easily prove system correctness by testing.
Reusability	The extent to which architecture can be reused.
Portability	The ability to move software system to a different hardware and/or software platform.
Maintainability	The ability of a system to undergo evolution and repair.
Modifiability	Sensitivity of the system to changes in one or several components.

Table 3.2: Non-functional quality attributes

nent quality requirements, as well as several constraints and design rules component must obey, e.g., means of communication [17].Hence, developing reusable components depends on the software architecture, because the software architecture to a large extent defines the way a component is to be developed, its functionality and required quality [17]. Thus, the software architecture represents the effective basis for reuse of components [88]. Moreover, there are indications that software evolution and reuse is more likely to receive higher payoff if architectures and designs can be reused and can guide low level component reuse [67, 76].

In the system development the issue of handling conflicts in quality requirements during architectural design must be explicitly addressed. That is because a solution for improving one quality attribute affects other quality attributes negatively, e.g., reusability and performance are considered to be contradicting, as are fault-tolerance and real-time computing. E.g., consider a software system fine-tuned to meet extreme performance requirements. In such a system, it could be costly to add new functionality (components), since adding new parts could result in degraded performance of the upgraded system. In such a scenario, additional efforts would have to be made to ensure that the system meets the initial performance requirements.

3.1.3 The future of component-based software engineering: from the component to the composition

So far, we have examined software components and software architectures, as they are the core of the component-based software development. However, the research in the component-based software engineering community increasingly emphasizes composition of the system as the way to enable development of reliable systems, and the way to improve reuse of components. In this section we give introduction to new techniques dealing with the problem of system composition. Figure 3.2 provides hierarchical classification of component-based systems [69].

Component-based systems on the first level, e.g., CORBA, COM, JavaBeans, represent the first generation of component-based systems, and are referred to as “classical” component-based systems. Frameworks and standards for components of today in industry primarily focus on classical component-based systems. Components are black boxes and communicate through standard interfaces, providing standard services to clients, i.e., the component is standardized. Standardization eases adding or exchanging components in the software system, and improves reuse of components. However, classical component-based systems lack rules for the system composition, i.e., composition recipe.

The next level represents architecture systems, e.g., RAPIDE [66], UNICON [120]. These systems provide an architectural description language (ADL). ADL is used to specify architecture of the software system. In an architecture system, components encapsulate application-specific functionality, and are also black boxes. Components communicate through connectors [3]. A connector is a specific module that encapsulates communication between application-specific components. This gives significant advancement in the composition as compared to classical component-based systems, since communication and the

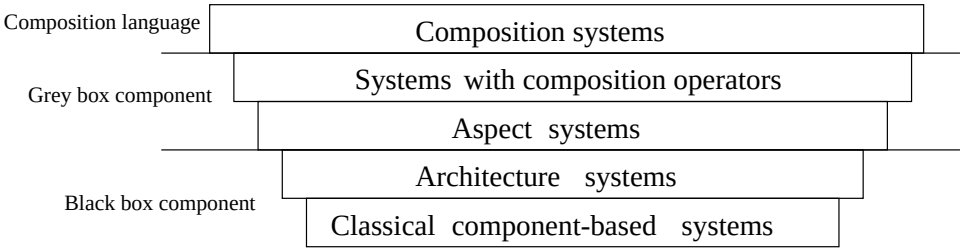


Figure 3.2: Classes of component-based systems

architecture can be varied independently of each other. Thus, architecture systems separate three major aspects of the software system: architecture, communication, and application-specific functionality. One important benefit of architecture system is possibility of early system testing. Tests of the architecture can be performed with "dummy" components leading to the system validation in the early stage of development. This also enables the developer to reason about the software system on an abstract level. Classical component-based systems, adopted in the industry, can be viewed as a subset of architecture systems (which are not yet adopted by the industry), as they are in fact simple architecture systems with fixed communication.

The third level represents aspect systems, based on aspect-oriented programming (AOP) paradigm [52]. Aspect systems separate more aspects of the software system than architecture systems. Beside architecture, application, and communication, aspects of the system can be separated further: representation of data, control-flow, and memory management. Temporal constraints can also be viewed as an aspect of the software system, implying that a real-time system could be developed using AOP [13]. Only recently, several projects sponsored by DARPA (Defense Advanced Research Projects Agency), have been established with an aim to investigate possibilities of reliable composition of embedded real-time systems using AOP [94]. Aspects are separated from the core component and are recombined automatically, through process called weaving. In AOP, a core component is considered to be a unit of system's functional decomposition, i.e., application-specific functionality [52]. Weavers are special compilers which combine aspects with core components at so-called joint points statically, i.e., at compile time, and dynamically at run-time. Weaving brakes the core component (at joint points) and cross-cuts aspects into the component, and the weaving process results in an integrated component-based system. Core components are no longer black boxes, rather they are grey boxes as they are cross-cut with the aspects. However, aspect weavers can be viewed as black boxes, since they are written for a specific combination of aspects and core components, and for each new combination of aspects and core components a new aspect weaver needs to be written. As compared to architecture systems, aspect systems are more general and allow separation of various additional aspects, thus, architecture systems can be viewed as the subset of the class of aspect systems. Having different aspects improves the reuse, since various aspects can be combined (reused) with different core components. Drawback of aspect systems is that they are build on special languages for aspects, requiring system developers to learn these languages.

At the fourth level are systems that provide composition operators by which components can be composed. Composition operators are comparable to component-based weaver, i.e., a weaver that is no longer a black box, but is also composable out of components, and can be, in a sense, re-composed for every combination of aspects and components, further improving the reuse. Subject-oriented programming (SOP) [84], an example of systems with composition operators, provide composition operators for classes, such as merge (merges on two views of a class), equate (merges two definition of classes into one),

etc. SOP is a powerful technique for compositional system development, since it provides a simple set of operators for weaving aspects or views, and SOP programs support the process of system composition. However, SOP focuses on composition and does not provide any special component. Components in SOP are C++ classes.

Finally, at the last level are systems that contain full-fledged composition language, and are called composition systems. A composition language should contain basic composition operators to compose, glue, adopt, combine, extend and merge components. The composition language should also be tailorable, i.e., component-based, and provide support for composing (different) systems, in the large. Invasive software composition [69] is one approach that aims to provide a language for the system composition. Components in the invasive composition technique are more general. The component may consist of a set of arbitrary program elements, and is called a box. Boxes are connected to the environment through very general connection points, called hooks. Composition of the system is encapsulated in composition operators (composers), which transform the component with hooks into a component with code. The process of system composition using composers is more general than aspect weaving and composition operators, since invasive composition allows composition operators to be collected in libraries and to be invoked by the composition programs (recipes) in a composition language. Composers can be realized in any programming or specification language. Invasive composition supports software architecture, separation of aspects, and provides composition receipts, allowing production of families of variant systems [69]. Reuse is improved, as compared to systems in the lower levels, since composition recipes can also be reused, leading to easy reuse of components and architectures. An example of the system that supports invasive composition is COMPOST [23]. However, COMPOST appears to be more suitable for complex software systems, rather than systems that have limited amount of resources, since resource demands for the components and the system are not addressed. Further, COMPOST is not developed to operate at runtime. Also, the system is not general enough, since it only supports Java source-to-source transformations.

The given overview illustrates current efforts to enable efficient system composition. In particular, techniques at the last three layers in figure 3.2 focus on the composition process. Component-based systems we investigate in more detail in the following sections do not cover all classes of component-based systems shown in figure 3.2. Systems we investigate represent current componentization efforts in the database and the embedded real-time community, and are mainly subsets of component-based systems in the first two layers in figure 3.2.

3.2 Component-based database systems

A database system consists of a software called database management system (DBMS) and one or more databases managed by the DBMS [100]. Essential part of the database system is the DBMS.

In this section we focus on databases systems that can be partially or completely assembled from a pre-defined set of components, i.e., that exploit component-based development. We refer to these systems as component-based database systems. First, the main granularity level of a component used for the composition of a database system is established. Then, in sections that follow, the classification and detail analysis of existing component-based database systems is given. Of a particular interest is to show how and to which degree different component-based database systems allow customization, i.e., to which degree a designer can tailor the database system for a particular application.

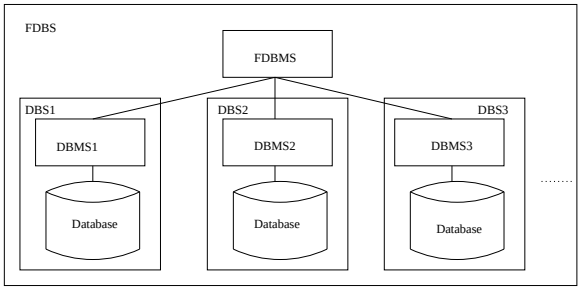


Figure 3.3: Components in FDBS

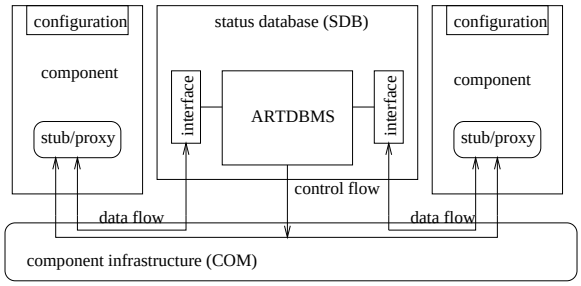


Figure 3.4: Component-based system with ARTDBS as a component

3.2.1 Granularity level of a database component

In general, two main granularity levels of a database component in component-based systems exist:

- DBMS (database system) as a component, and
- part of the DBMS as a component.

The former component type is not exactly a component-based solution of a database system, rather DBMS used as a component in larger component-based systems. For example, a database system may be viewed as a component in a federated database system (FDBS), which is a collection of cooperating but autonomous database systems [100]. Database systems are controlled and managed by a software called federated database management system (FDBMS) (see figure 3.3). Hence, the database system is not made out of components but is rather a component of a larger component-based federated database system. In this respect, CORBA can be used to achieve interoperability among DBMSs in a multidatabase approach. CORBA enables the basic infrastructure that can be used to provide database interoperability [29]. When using CORBA as a middleware in a multidatabase system, DBMS itself is not customizable; rather DBMS is a component (CORBA object) registered to the CORBA infrastructure. Another example of a database system used as a component is an active real-time database system (ARTDBMS) that can be used as a component in an architecture with standardized components [77] (see figure 3.4). In this solution, infrastructure of the component technology (in this case COM) is used for separating the exchange of control information between components and data transfer via the status database system. The status database system is implemented as an active real-time database system and contains all the information about the current status of an application that is used by at least two different components. This architecture supports application composition out of components, and aims to enable adding or removing components without additional programming. A component in this system is assumed to be a self-contained,

configurable binary software package with precisely defined standardized interfaces that can be combined with other components. Time consistency of saved data, in addition to the logical consistency, is ensured in the composed system by having a real-time database. However, this type of component-based solution is applicable for tailoring component-based systems where time consistency of data is needed and not for tailoring a database system. A database system in this solution is a monolithic system, and changes to that type of systems are not easily done.

It is our opinion that more emphasis and detailed analysis is needed for the second category of database components, i.e., a component being a part of the DBMS. DBMS made out of components would indeed be an actual component-based solution of a database system. We motivate this further in the following section.

3.2.2 Component vs. monolithic DBMS

Database systems are today used in various application areas, such as telecommunications [1], e-commerce [37, 2], embedded systems [81], real-time systems [44], etc. Expansion of database systems to new application domains unavoidably results in new requirements a database system must fulfill [28]. Consider Internet applications where data is accessed from a variety of sources such as audio, video, text, image data, etc. Therefore, many web-based applications require a database that can manage vide variety of different types of data. Also, new application-specific data types have emerged that are not supported by traditional databases. Examples of non-standard data types are temporal data, spatial data, multimedia data, etc. Hence, DBMSs are required to provide support for modeling and storing these new, non-standard, data types [83]. Broad variety of data from different data sources makes heterogeneous data management a very important challenge, since a large number of applications require a database that provides them with the uniform and homogeneous view of the entire system [21]. Business computing environment poses additional set of requirements on database systems, such as rapid application development, and minimized complexity both for end users and developers [82]. In addition, it is common that a company needs to produce a database system for a bigger spectra of different application areas. Such scenarios require cost-effective solutions of DBMSs. During the development of a DBMS, it is very hard to exactly predict in which direction system will evolve, i.e., which additional features the database system will have to provide in the future. Thus, a DBMS must be developed such that it is open to new, future, uses. Having a database that can be extended with new features or modified during its life-time is beneficial both for users and database vendors.

Meeting new requirements implies that traditional, monolithic DBMS must be extended to include new functionality. However, adding functionality to a monolithic DBMS has several drawbacks [28]:

- Parts in the monolithic DBMS are connected to one another and dependent on one another to a such a degree that changes made in one part of the DBMS structure would lead to a domino effect in other parts of the monolithic system, i.e., extensions and modifications are not easily done.
- Adding new functionality in the monolithic system would result in increased complexity of a DBMS and in turn, increased maintenance costs of the system.
- Applications would have to pay performance and cost penalty for using unneeded functionality.
- System evolution would be more complex, since a DBMS vendor might not have the resources or expertise to perform such extensions in a reasonable period of time.

One approach to solve these problems could be componentization of a DBMS [28]. A component DBMS (CDBMS), also called a component-based database system, allows new

functionality to be added in a modular manner, that is, system can be easily modified or extended by adding or replacing new components. Components and their interconnections should be well-defined in the CDBMS architecture. The architecture defines places in the system where components can be added, thus, specifying and restricting the ways in which DBMS can be customized. Ideally, applications would no longer have to pay performance and cost penalty for using unneeded functionality because unnecessary components do not have to be added to a system. However, it has to be noted that components needed by the application might contain some unnecessary functionality themselves. Also, complexity of the system and maintenance cost would be reduced if the system could be composed only out of components (functionality) application needs. Finally, evolution of such a system would be simplified, since new components can be plugged into the system without having to recompile (or shut down) the entire system. Of course, this implies that new components have previously been tested and have a verified behavior.

Note that component-based database systems are faced with the same challenges as a component-based technology in general, such as performance degradation of a component-based system as compared to a monolithic system, complex development process of a reusable component, etc.

3.2.3 Components and architectures in different CDBMS models

Four different categories of CDBMSs have been identified in [28]:

- Extensible DBMS. The purpose of systems falling into this category is to extend existing DBMS with non-standard functionality, e.g., Oracle8*i* [83], Informix Universal Server with its DataBlade technology [47], Sybase Adaptive Server [82] and DB2 Universal Database [21].
- Database middleware. The purpose of systems falling into this category is to integrate existing data stores into a database system and provide users and applications with a uniform view of the entire system, e.g., Garlic [22] and DiscoveryLink [42], and OLE DB [75], .
- DBMS service. The purpose of systems falling into this category is to provide database functionality in standardized form unbundled into services, e.g., CORBAService [85].
- Configurable DBMS. The purpose of systems falling into this category is to enable composition of a non-standard DBMS out of reusable components, e.g., KIDS [38].

Sections that follow focus on characteristics of systems in each of these four different categories.

3.2.4 Extensible DBMS

The core system in this group is formed from fully functional DBMSs that implement all standard DBMS functionality [28]. Non-standard features and functionality not yet supported can be plugged into this DBMS (see figure 3.5). Components in this category of CDBMSs are families of base and abstract data types or implementations of some DBMS function such as new index structures. The architecture defines a number of plugs that components can use and formulates expectations concerning interfaces that component must meet in order to be integrated successfully.

Oracle8*i*

Oracle8*i* allows developers to create their own application-domain-specific data types [83]. Capabilities of the Oracle data server can be extended by means of data cartridges which,

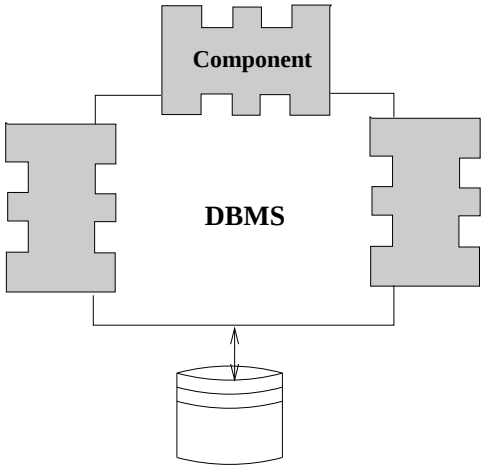


Figure 3.5: The principle of systems falling into an extensible DBMS category

in this case, represent components in the Oracle8*i* architecture. Data cartridges are stored in an appropriate library for reusability. A data cartridge consists of one or more domain-specific types and can be integrated with the server. For example, a spatial data cartridge may provide comprehensive functionality for a geographical domain such as being able to store spatial data, perform proximity/overlap comparisons on such data, and also integrate spatial data with the server by providing the ability to index spatial data. In addition to the spatial cartridge, text, image and video data cartridge can be integrated in the architecture. Data cartridges can be integrated into a system through extensibility interfaces. There are three types of these interfaces: DBMS and data cartridge interfaces, used for communication between components and the DBMS, and service interfaces used by the developers of a component. In the Oracle8*i* architecture, data cartridges can provide their own implementation of database services such as storage service, query processing service, services for indexing, query optimization, etc. These services are extended by the data cartridge implementation, and are called database extensibility services (see figure 3.6). Extensibility of a database service can be illustrated with the example of the spatial data cartridge, which provides the ability to index spatial data, and consequently extends indexing service of the server. Configuration support is provided for the development, packaging and deployment of data cartridges, as illustrated in figure 3.7. The Oracle Designer family of products has modeling and code generation tools which enable development of data cartridges. The Cartridge Packager module, part of the Oracle Enterprise Manager family, assists developer in packaging data cartridges into installable units. Components (data cartridges) are then installed into the server by end users using the Oracle Installer.

The architecture of the Oracle8*i* is fixed and defines the places where extensions can be made, i.e., components added, as well as interfaces to these components. Designer is allowed to customize database server only by plugging-in new components, i.e., the system has low degree of tailorability. Provided configuration support is adequate, since the system already has a fixed architecture and pre-defined extensions, and that extensions are allowed only in well-defined places of the architecture. This type of system emphasizes on satisfying only one requirement - handling non-standard data types. If a system needs to provide another (non-standard) service beside managing abstract data types, or if some internal part of the infrastructure needs to be changed, we are faced with the same difficulties as when using monolithic systems (see section 3.2.2). Also, this type of system can not easily be integrated in all application domains, e.g., real-time system, since there is no analysis support for checking temporal behavior.

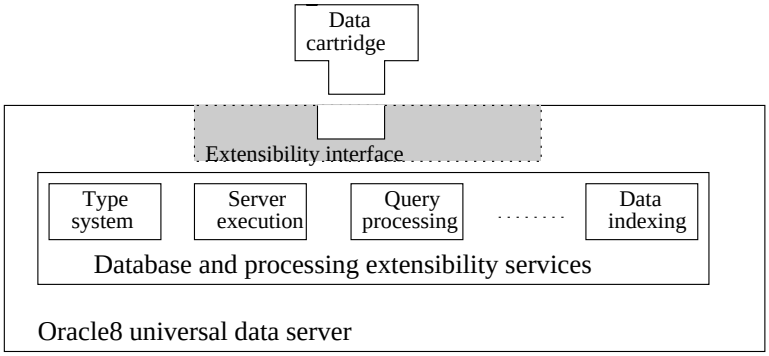


Figure 3.6: The Oracle extensibility architecture

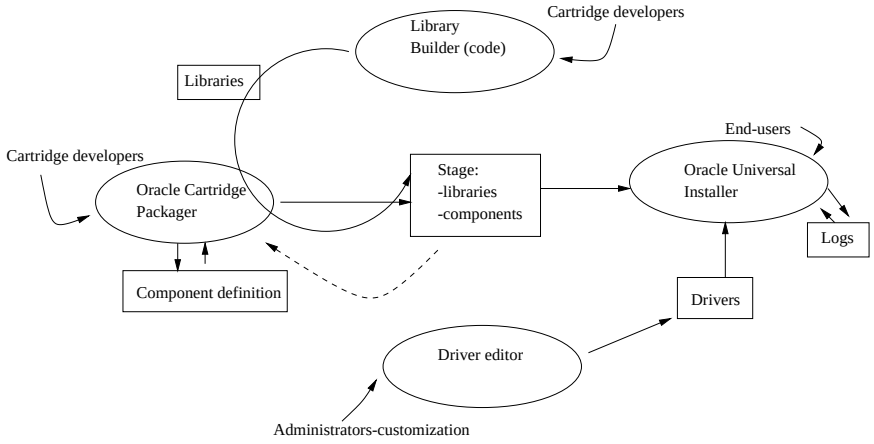


Figure 3.7: Packaging and installing cartridges

The concept of the Informix DataBlade technology, DB2 Universal Database, and Sybase Adaptive Server is the same as the above described concept of the extensible Oracle8*i* server architecture for manipulation of non-standard data types. Moreover, conclusions made for the Oracle8*i* with respect to configuration tools, tailorability for satisfying only one requirement (handling non-standard data types), and lack of real-time properties and analysis tools, are applicable for all systems that fall in this category, i.e., extensible DBMS. Therefore, analysis of the Informix DataBlade technology, DB2 Universal Database, and Sybase Adaptive Server is kept very short.

Informix DataBlade technology

DataBlade modules are standard software modules that can be plugged into the Informix Universal Server database to extend its capability [47]. DataBlade modules are components in the Informix Universal Server. These components are designed specifically to enable users to store, retrieve, update, and manipulate any domain-specific type of data. Informix allows development of DataBlade modules in Java, C, C++, J++ or stored procedure language (SPL) by providing the application programming interface (API) for those languages. Same as Oracle, Informix has provided low degree of tailoring, since the database can only be extended with standardized components that enable manipulation of non-standard data types. Configuration support is provided for development and installation of DataBlade modules, e.g., BladeSmith, BladePack, and BladeManager.

DB2 Universal Database

As previously described database servers, DB2 Universal Database [21, 27] also allows extensions in the architecture to provide support for comprehensive management of application-specific data types. Application-specific data types and new index structures for that data types are provided by DB2 Relational Extenders, reusable components in the DB2 Universal Database architecture.There are DB2 Relation Extender for text (text extender), image (image extender), audio and video (extender). Each extender provides the appropriate functions for creating, updating, deleting, and searching through data stored in its data type. An extender developers kit with wizards for generating and registering extenders provides support for the development and integration of new extenders in DB2 Universal Database.

Sybase Adaptive Server

Similar to other database systems in this category, the Sybase Adaptive Server [82] enables extensions in its architecture, called Sybase's Adaptive Component Architecture (ACA), to enable manipulation of application-specific data types. Components that enable manipulation of these data types are called Speciality Data Stores, e.g., speciality data stores for text, time series and geospatial data. The Sybase Adaptive Server differs from other database systems in the extensible DBMS category in that it provides a support for standard components in the distributed computing environment, as the name of the ACA architecture indicates. Through open (Java) interfaces Sybase's Adaptive Component Architecture (ACA) provides mechanisms for communication with other database servers. In addition of providing the interoperability of database server, Sybase enables interoperability with other standardized components in the network, such as JavaBeans.

3.2.5 Database middleware

The aim of systems falling into this category is to integrate existing data stores, into a common DBMS-style framework [28]. A CDBMS acting as a middleware between different

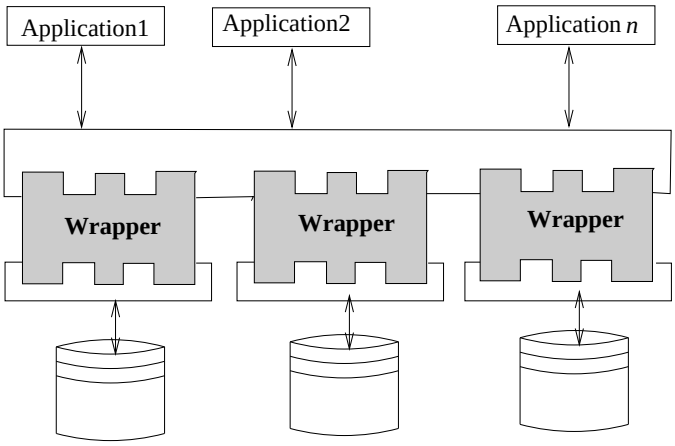


Figure 3.8: Principle of systems falling into a DBMS middleware category

data stores and the application of integration provides users and applications with a uniform and integrated view of the entire system (see figure 3.8). In this model the architecture introduces a common intermediate format into which the local data formats can be translated. Components in in this category of CDMSs perform this kind of translation. Common interfaces and protocols define how the database middleware system and the components interact. Components, also known as wrappers, are located between the DBMS and each data source. Each component mediates between the data source and the DBMS, i.e., it wraps the data source.

Garlic

Garlic [22] is an IBM research project that aims to integrate and unify data managed by multiple, disparate, data sources. The primarily goal of the Garlic project is to build heterogeneous multimedia information system capable of integrating data from a broad range of data sources. Each data source (also known as a repository) has its own data model, schema, programming interface and query capability. Data models for these sources vary widely, e.g., relational, object-oriented, a simple file-system, and a specialized molecular search data model. The Garlic data model is based on the Object Database Management Group (ODMG) standard. Garlic can provide access to a data source only if appropriate wrapper for that data source exists. Thus, associated with each data source is a wrapper, a reusable component in the Garlic architecture (see figure 3.9). In addition to data sources containing legacy data, Garlic allows users to create their own data source, i.e., data source for Garlic complex objects. Wrappers for new data sources can be integrated into an existing Garlic database without disturbing legacy applications, other wrappers and Garlic applications. Garlic applications interact with query services and run-time system through Garlic's object query language and a C++ API. The query processor develops plans to efficiently decompose queries that span multiple data sources into pieces that individual data sources can handle. As shown in figure 3.10 wrapper provides four major services to the Garlic system [97]:

1. A wrapper models the contents of its data source as Garlic objects, and allows Garlic to retrieve references to these objects. The Garlic Data Language (GDL), a variant of the ODMG's Object Description Language (ODMG-ODL), is used to describe the content of a data source. Interfaces that describe the behavior of objects in a data source (repository) are known collectively as a repository schema. Repositories are registered as parts of the Garlic database and their individual repository schemas are

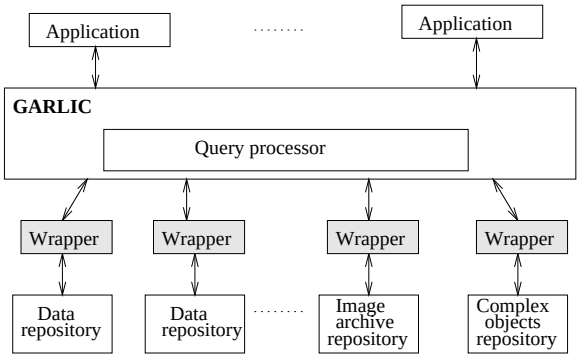


Figure 3.9: The Garlic architecture

merged into the global schema that is presented to Garlic users.

2. A wrapper allows Garlic to invoke methods on objects and retrieve their attributes. Method invocation can be generated by Garlic's execution engine, or by a Garlic application that has obtained a reference (either as a result of a query or by looking up root object by name).

3. A wrapper participates in query planing when a Garlic query processor ranges over multiple objects in repository. The goal of the query planning is to develop alternative plans for answering a query, and then to choose the most efficient one.

4. A wrapper's finale service is to participate in plan translation and query execution. A Garlic query plan is presented as a tree of operators, such as FILTER, PROJECT, JOIN, etc. The optimized plan must be translated into a form suitable for execution. During every execution the wrapper completes the work which was assigned to it in the query planing phase. Operators are mapped into iterators, and each wrapper provides a specialized `Iterator` subclass that controls execution of the work described by one of its plans.

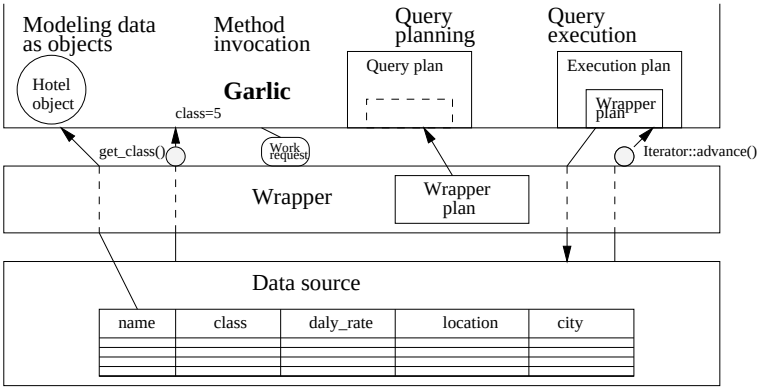


Figure 3.10: Services provided by the wrapper

Having defined services that a wrapper provides to the Garlic architecture, simplifies the development process of the wrapper. A designer of the wrapper needs to perform specific task to ensure that a wrapper provides a certain service. After defining the wrapper services, i.e., performing tasks specific to each service, the wrapper designer has a final task

of packaging written pieces in a complete wrapper. To keep the cost of communication between the Garlic and a wrapper low, the wrapper code is packaged as a dynamically loadable library and resides in the same address space. Every wrapper includes interface files that contain one or more interface definitions written in GDL, environment files that contain name/value pairs to encode repository-specific information for use by the wrapper, and libraries that contain dynamically loadable code to implement schema registration, method invocation, and the query interface. Decomposing a wrapper into interface files, libraries, and environment files gives the designer of a wrapper additional flexibility when designing a wrapper for a particular repository or family of repositories. For example, the wrapper for relational databases packages generic method dispatch, query planing code, and query execution code as sharable library. For each repository, an interface file describes the objects in the corresponding database, and an environment file encodes the name of the database to which the wrapper must connect, the names of the roots exported by the repository and the tables to which they are bound, etc. Wrapper designers are provided with a library of schema registration tools, query-plan-construction routines, a facility to gather and store statistics, and other useful routines in order to automate writing wrappers as much as possible.

To conclude, the well-defined Garlic wrapper architecture with wrapper services and tasks for a wrapper designer associated with each service, enables easy and fast (cost-effective) development of wrappers. The architecture also provides stability in a sense that new wrappers can be easily installed into an existing system without effecting the other wrappers or the system itself. The Garlic system is suitable for large distributed database systems. In terms of tailorability, one can observe that the Garlic wrapper architecture provides a moderate degree of customization, and can be tailored for different applications, provided that appropriate wrappers, needed by the application exist, or are developed.

However, the Garlic cannot be used with ease in every application domain, e.g., a real-time domain, where correctness of a system depends on the time when results are produced. The Garlic wrapper architecture unifies access to different data source, but it is not clear if such access can be guaranteed in a timely predictable manner.

DiscoveryLink

DiscoveryLink [42] is a commercial middleware DBMS, completely based on the Garlic technology, and incorporated into a IBM's DB2 database server. Discovery Link integrates life science data from heterogeneous data sources . As in Garlic, components, i.e., wrappers, in the DiscoveryLink are C++ programs, packaged as shared libraries. Wrappers can be created dynamically, allowing a number of data sources to to grow (and shrink) on the fly. Most of the wrappers in DiscoveryLink are required to have only minimum functionality, in particular, the wrapper must be able to return the tuples of any relation that it exports. The wrapper architecture and the notion of having reusable components that can be added or exchanged makes DiscoveryLink an unique among commercial database middleware systems that provide query across multiple relational sources, e.g., DataJoiner[1] and Sybase[2].

Since DiscoveryLink completely relies on the Garlic principles, but is tailored for a specific application domain, i.e., life science data, analysis performed for the Garlic is sufficient to get an inside of the DiscoveryLink and its characteristics (see previous description of the Garlic wrapper architecture).

OLE DB

OLE DB [14, 15] is designed as a Microsoft Component Object Model (COM) interface. COM provides a framework for integrating components. This framework supports inter-

[1] http://www.software.ibm.com/data/datjoiner
[2] http://www.sybase.com

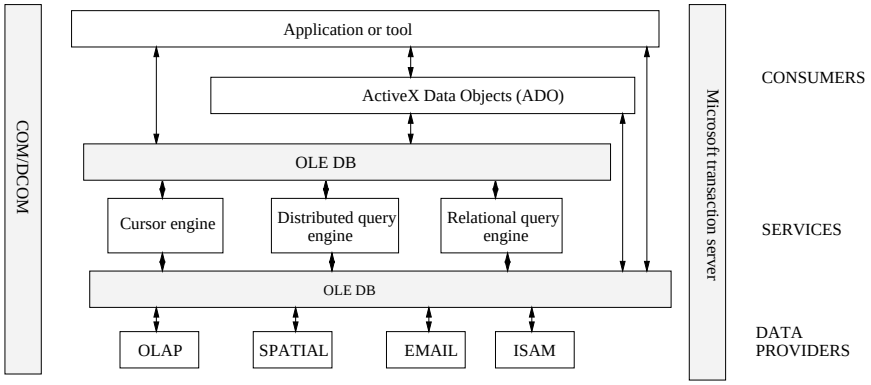


Figure 3.11: The Universal Data Access (UDA) architecture

operability and reusability of distributed objects by allowing developers to build systems by assembling reusable components from different vendors which communicate via COM. COM defines an application programming interface (API) to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact. However, in order to interact, components must adhere to a binary structure specified by Microsoft. As long as components adhere to this binary structure, components written in different languages can inter-operate. COM supports concepts such as aggregation, encapsulation, interface factoring, inheritance, etc.

OLE DB is a specification for a set of data access interfaces designed to enable a variety of data stores to work together [75]. OLE DB provides a way for any type of data store to expose its data in a standard and tabular form, thus unifying data access and manipulation. In Microsft's infrastructure for component-based computing, a component is thought of as [75]

> ...the combination of both process and data into a secure, reusable object...

and as a result, both consumers and providers of data are treated as components. A data consumer can be any piece of the system or the application code that needs access to a broad range of data. In contrast, data providers are reusable components that represent data source, such as Microsoft ODBC, Microsoft SQL server, Oracle, Microsoft Access, which are all standard OLE DB providers. Thus, OLE DB enables building component-based solutions by linking data providers and data consumers through providing services that add functionality to existing OLE DB data and where the services are treated as components in the system (see figure 3.11). The architecture in figure 3.11 is called the Universal Data Access (UDA) architecture. It is possible to develop new, custom, data providers that reuse existing data providers as the underlying component or a component building block of more complex (data provider) component. This enables developers to expose custom views of data and functionality without rewriting the entire data store or the access methods. Note that only reusable components in this system are data providers. These components can be viewed as wrappers in the UDA architecture. Note that architectural requirements get more complex, as compared to the systems in the extensible category. In contrast to the Garlic wrapper architecture, OLE DB allows the UDA architecture to have more variable parts and more customization of the system is allowed. The UDA architecture provides a common format into which all local data formats can be translated as well as standardized interfaces that enable communication between the applications and different data stores, i.e., OLE DB interface.

Although OLE DB provides unified access to data and enables developers to build their own data providers, there is no common implementation on either the provider or consumer

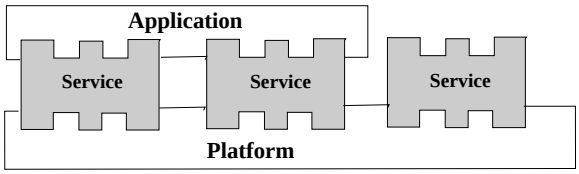


Figure 3.12: The principle of systems falling into a DBMS service category

side of the interface [16]. Compatibility is provided only through the specification and developers must follow the specification exactly to make interoperable components, i.e., adequate configuration support for this is not yet provided. To make up for inadequate configuration support, Microsoft has made available, in Microsoft's Software Developer's Kit (SDK), tests that validate conformance of the specification. However, analysis of the composed system is missing.

OLE DB is not applicable for a real-time domain, since temporal behavior is not of an importance. Additionally, OLE DB is limited with respect to software platforms, since it can only be used in Microsoft software environments.

3.2.6 DBMS service

In this type of a CDBMS, the DBMS and related tasks are unbundled into services [28], and as a result the monolithic DBMS is transformed into a set of stand-alone services (see figure 3.12). Applications no longer operate on full fledged DBMSs, but instead use those services as needed. Services are defined in a common model or language and are implemented using a common platform in order to render the service implementations exchangeable and freely combinable. Systems in this category have components that are database services and their implementations. CORBA with its CORBAservices could be an example of a system falling into this category. Although there are no DBMS service systems that are implemented solutions, the objectives of this category can be illustrated through a CORBAservices example-system.

CORBAservices

One single DBMS could be obtained by gluing together CORBAservices which are relevant for databases, such as transaction service, backup and recovery service, concurrency service. CORBAservices are implemented on the top of the Object Request Broker (ORB). Service interfaces are defined using the Interface Definition Language (IDL) [29]. In this scenario component would be one of the database relevant CORBAservices. This would mean that applications could choose from a set of stand-alone services those services, i.e., components, which they need. However, this approach is (still) not viable because it requires writing significant amount of glue code. In addition, performance overhead could be problem due to the inability of an ORB to efficiently deal with fine-granularity objects [85]. Also, an adequate value-added framework that allows development of components and use of these components in other applications is still missing. Configuration as well as analysis tools to support this process are missing.

3.2.7 Configurable DBMS

One step further away from the DBMS service category of CDBMSs, where the set of services have been standardized and fixed, is a configurable DBMS category that allows new DBMS parts to be developed and integrated into a DBMS (see figure 3.13) [28]. Here, com-

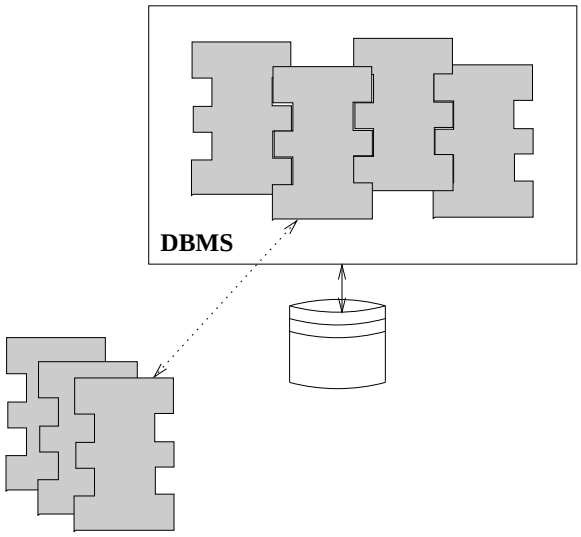


Figure 3.13: The principle of systems falling into a configurable DBMS category

ponents are DBMS subsystems defined in the underlying architecture which is no longer fixed.

KIDS

KIDS (Kernel-ased Implementation of Database management Systems) [38] approach in constructing CDBMSs is an interesting research project from the University of Zürich, since it offers a methodical, open, and cost-effective construction of a DBMS. The approach offers high level of reusability, where virtually any result obtained in a previous system construction is reused (designs, architectures, specifications, etc.). The architecture is well-defined, and once created it can be saved in a library and reused. The DBMS architecture is based on the broker/service model, i.e., it consists of brokers, services, and responsibilities. A service represents a specific task to be provided by the DBMS, and each service is provided by at least one component in the system and can be requested by raising events. Services are provided by reactive processing elements called brokers (brokers are comparable to objects in object-oriented programming). Components in KIDS are DBMS subsystems that are collections of brokers. Brokers are responsible for a related set of tasks, e.g., object management, transaction management and integrity management. For example, the transaction management subsystem can consist of a transaction broker, scheduler, log management broker and recovery broker. Transaction broker is responsible for starting and terminating transactions, i.e., services such as begin transaction, commit and abort. The scheduler is responsible for serializing concurrent transactions, i.e., validate service which checks whether the database access is legitimate with the respect to the execution of concurrent transactions. A structural view of the KIDS architecture is shown in figure 3.14. The DBMS architecture consists of two layers. The first layer is the object server component, which supports the storage and retrieval of storage objects. The object server provides a fixed interface, which hides implementation details and operating system characteristics from upper layers. The object server component is reused in its entirety, and belongs to the fixed part of the DBMS architecture (this is because the object server implements functionality needed by any DBMS). The second layer is variable to a large extent, and can be decomposed into various subsystems. In the initial decomposition of KIDS, three major subsystems exist:

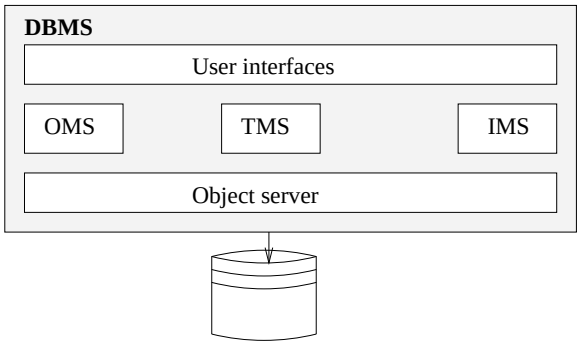


Figure 3.14: The KIDS subsystem architecture

- The object management subsystem (OMS), which implements the mapping from data model objects into storage objects, retrieval of data model objects, and meta data management.
- The transaction management subsystem (TSM), which implements the concept of a transaction, including concurrency control, recovery, and logging.
- The integrity management subsystem (IMS), which implements the (DBMS-specific) notion of semantic integrity, and is responsible for checking whether database state transitions result in consistent states.

Each of these subsystems is implemented using dedicated languages and reuse techniques. These three subsystems (OMS, TMS, and IMS) implement basic database functionality. Additional functionality can be provided by adding new subsystems in the second layer of the KIDS architecture, i.e., expanding decomposition of this layer to more than three subsystems.

KIDS is composed out of components (subsystems) which are carefully defined in an underlying architecture. The construction process of the CDBMS is well-defined, and starts after a requirement analysis of the desired DBMS, for a specific domain, has been performed and relevant aspects, i.e., functionality that DBMS needs to provide, have been determined (see figure 3.15). The process of DBMS construction continues in phases:

- The selection of an architecture skeleton from the library, or a development of a new architecture skeleton, i.e., a (possibly still incomplete) collection of partially defined subsystems.
- The subsystem development, which consists of subsystem design, implementation, and subsystem integration. At this phase the process branches in a sense of parallelism for a design and implementation of each subsystem. Integration of subsystems into previously defined architectural skeleton is then performed. Since it might not always be possible to integrate the subsystem after the implementation of all subsystems has been completed, it is possible to perform integration as far as necessary and continue with subsystem development, i.e., this phase is iterative and incremental.
- Testing and optimization are in the phase that follows the subsystem development. The KIDS construction process focuses on the development of subsystems and their integration. Analysis of the system is not investigated in greater details.

KIDS allows new components to be developed and integrated into the system, thus, enables tailoring of a DBMS for different applications. Expanding the initial set of components in the KIDS architecture with the functionality (components) needed by a particular

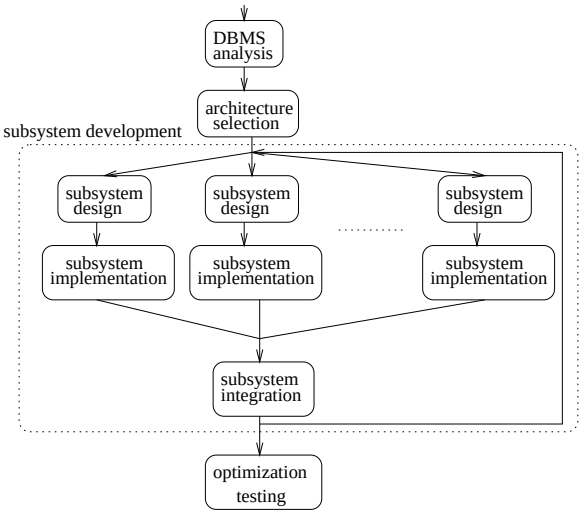


Figure 3.15: Phases of the DBMS construction process

application, one could be able to design "plain" object-oriented DBMS, a DBMS video-server, or a real-time plant control DBMS. Of course, in the proposed initial design of KIDS, real-time properties of the system or components are not considered. For a real-time application, proposed construction process can be used if a component (subsystem) is constructed such that its timing behavior is known, and the consumption of memory by each component is taken into account. Also, KIDS offers good basis for the early system testing; in the first phase of the construction process, at the architecture level of the system.

Defined process of a DBMS construction, reusability of components and architectures, and high degree of componentization (tailorability) of a system differentiate this CDBMS from all others.

3.3 Component based embedded and real-time systems

Embedded real-time systems are being used widely in today's modern society. Thus, agile and low-cost software development for embedded real-time systems has become critically important. Successful deployment of embedded real-time systems depends on low development costs, high degree of tailorability and quickness to market. Thus, the use of the component-based software development paradigm for constructing and tailoring embedded real-time systems has premise. We elaborate this in more detail by listing some of the major benefits of using component-based software engineering for development of embedded real-time systems [109, 64]:

- Rapid development and deployment of the system. Component-based software development aims to reduce the amount of new code that must be written each time new application is being developed. Many software components, if properly designed and verified, can be reused in different embedded real-time applications. Embedded real-time systems built out of components may be readily adopted and tuned for new environments than monolithic systems.

- Changing software "on-the-fly". Embedded real-time sensor-based control systems may be designed to have software resources that can change on the fly, e.g., controllers, device drivers. This feature is desirable in autonomous and intelligent control applications, as well.

- Evolutionary design. In a component-based system software components can be replaced or added in the system. This is appropriate for complex embedded real-time systems that require continuous hardware and software upgrades in response to technological advancements, environmental change, or alteration of system goals.

- Product lines. It is common that manufacturers create several different, but still similar, embedded real-time products. Having reusable components stored in a library would enable the entire product line to use same software components from the library when developing products. This eliminates the need to develop and maintain separate software for each different product.

- Increased reliability. It is easier to develop, test and optimize individual components of limited functionality, than when the same functionality is embedded within a large monolithic system. In addition, individual components can be specified and verified more easily, and provided that the rules for the composition of a system are well-defined, safety-critical applications can be composed from such components.

- Fine-tuning. Having components in the system that can be replaced offers considerable flexibility for fine-tuning of an embedded real-time application, e.g., components can be introduced that enable switching between static and dynamic scheduling algorithms.

- Reduced system complexity. Components are designed to have different functionality. Choosing components from the library that provide functionality needed by the system should reduce system's complexity. This is true since it is likely that a component (as compared to the monolithic system) will not contain large parts of functionality unneeded by the system.

In the remainder of this section we give analysis of existing component-based embedded real-time systems. However, some of the component-based systems analyzed are not embedded real-time, but are only embedded or only real-time systems. To obtain a good understanding of how component-based software development can be successfully integrated in the area of embedded real-time systems, it is necessary to investigate both real-time and embedded system, in addition to those system that can be classified as embedded real-time (definitions of embedded, real-time, and embedded real time systems are given in chapter 1). In order to keep the notation as simple as possible and, at the same time, easily understandable, when referring to multiple systems that can be classified either as real-time or as embedded or as embedded real-time, we use the notation embedded (and) real-time systems.

3.3.1 Components and architectures in different component-based embedded (and) real-time systems

We have identified three distinct types of component-based embedded (and) real-time systems:

- Extensible systems. An example of this type of the system is SPIN [12], an extensible microkernel. Extensions in the system are possible by plugging components, which provide non-standard features or functionality, into an existing system. Extensions are allowed only in well-defined places of the system architecture.

- Middleware systems. These are characterized by their aim of providing efficient management of resources in dynamic heterogeneous environments, e.g., 2K [54] is a distributed operating system that is specifically developed for management of resources in a distributed environment, which consists of a different software and hardware platforms.

- Configurable systems. An architecture of a configurable system allows new components to be developed and integrated into the system. Components in such systems are true building parts of the system. A variety of configurable systems exists, e.g., VEST [105], Ensemble [64], the approach to system development with real-time components introduced by Isović [48], and systems based on the port-based object (PBO) model [108].

3.3.2 Extensible systems

SPIN

SPIN [12, 87] is an extensible operating system that allows applications to define customized services. That is, SPIN can be extended to provide so-called application-specific operating system services [12]. An application-specific service is one that precisely satisfies the functional and performance requirements of an application, e.g., multimedia applications impose special demands on the scheduling, communication and memory allocation policies of an operating system. SPIN provides a set of core services that manage memory and processor resources, such as device access, dynamic linking, and events. All other services, such as user-space threads and virtual memory are provided as extensions. Thus, SPIN provides possibility to customize, i.e, tailor, the system according to the needs of a specific application. However, the system has a low degree of tailorability, since the architecture of SPIN can only be extended with components, called extensions.

A reusable component, an extension, is a code sequence that can be installed dynamically into the operating system kernel by, or on behalf of the application. Thus, an application can dynamically add code to an executing system to provide a new service. The application can also replace or augment old code to exchange existing services, i.e., components can be added, replaced, or modified in the system. E.g., an application may provide a new in-kernel file system, replace an existing paging policy, or add compression to network protocols. Extensions are written in Modula-3, a modular, ALGOL-like programming language. The mechanism that integrates extensions (components) with the core system are events, i.e., communication in SPIN is event-based. An event is a message that is raised to announce a change in the state of the system or a request for a service. Events are registered in an event handler, which is a procedure that receives an event. An event dispatcher oversees event-based communication. The dispatcher is responsible for enabling services such as conditional execution, multicast, asynchrony, and access control. An extension installs a handler with the event thorough a central dispatcher that routes events to handlers. Event-based communication, used in the SPIN architecture, allows considerable flexibility of the system composition. All relationships between the core system and components are subject to changes by simply changing set of event handlers associated with any given event.

The correctness of the composed system depends only on the language's safety and encapsulation mechanisms; specifically interfaces, type safety, and automatic storage management. Analysis of the composed system is not performed, since it is assumed that the configuration support provided within the Modula-3 language is enough to guarantee the correct and safe system. SPIN allows applications to implement their own scheduling policies. Provided the right extension for real-time scheduling policy this operating system can be used for soft real-time applications such as multimedia applications.

3.3.3 Middleware systems

2K

2K [56, 54] is an operating system specifically developed for manipulation of resources in a distributed heterogeneous environment (different software systems on different hardware platforms). 2K is based on a network-centric model and CORBA component infrastructure.

In the network centric model all entities, i.e., user applications (labeled as 2K applications), software components and devices, exist in the network and are represented as CORBA objects (see figure 3.16). When a particular service is instantiated, the entities that constitute that service are assembled. Software components (CORBA objects) communicate through IDL interfaces. As shown in figure 3.16, 2K middleware architecture is realized using standard CORBA services such as naming, trading, security and persistence, and extending the CORBA service model with additional services, such as QoS-aware management, automatic configuration, and code distribution. The 2K automatic configuration service

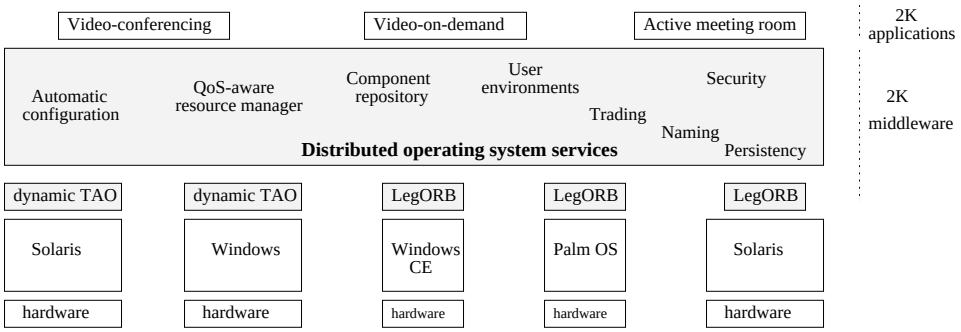


Figure 3.16: The 2K middleware architecture

supports loading components into run-time system. CORBA object, called ComponentConfigurator, in this system stores inter-component dependency, which must be checked before components can be successfully installed into a system by the automatic configuration service. An Internet browser, for example, could specify that it depends upon components implementing an X-Windows system, a local file service, the TCP/IP protocol, and the Java virtual machine version 1.0.2 or later. The automatic configuration service and ComponentConfigurator enable automated installation and configuration of new components, provided that the system has access to the requirements for installing and running a software component, i.e, inter-component dependencies.

Integration of components into the middleware is done through a component called dynamic TAO (The Adaptive Communication Environment ORB). The dynamic TAO is a CORBA compliant reflective ORB as it allows inspection and reconfiguration of its internal engine [55]. This is important since the middleware must be configurable and able to adopt to dynamic changes in resource availability and in the software and hardware infrastructure.

However, the dynamic TAO component has a memory footprint greater than few megabytes, what makes it inappropriate for use in environments with limited resources. A variant to the dynamic TAO, a LegORB component, is developed by the 2K group such that it has a small footprint, and is appropriate for embedded environments with limited resources, e.g., 6 Kbytes on the PalmPilot running on PlamOS. Dynamic TAO and LegORB components are not reusable, but are the key enabler for reuse of other CORBA objects, i.e., components in the network. Even though this system provides automated installation and configuration of new components, it does not specify the development of new components. The development of new components is done only based on CORBA component model specifications. Software components in this system can be reused. However, those are components which already exist in the network, and are reused in the sense that they can be dynamically installed into the system whenever some application needs a specific component. Also, it is is assumed that inter-component dependencies provide good basis for the system integration, and guarantee correct system behavior (other guarantees of the system behavior, obtained by appropriate analysis, do not exist).

Embedded system requirements are meet by the LegORB component, suitable for low-resource (embedded) systems. As figure 3.16 shows, 2K provides services for the appli-

cations such as video-on-demand and video-conferencing, which can be viewed as soft real-time applications. Considering that 2K uses CORBA component infrastructure with real-time CORBA extensions, i.e., TAO ORB, implies that hard real-time applications can also be supported by the 2K middleware architecture.

3.3.4 Configurable systems

Ensemble

Ensemble is a high performance network protocol architecture designed to support group membership and communication protocols [64]. Ensemble does not enforce real-time behavior, but is nevertheless interesting because of the configurable architecture and the way it addresses the problem of configuration and analysis of the system. Ensemble includes library of over sixty micro-protocol components which can be stacked, i.e, formed into a protocol, in a variety of ways to meet communication demands of an application. Each component has a common event-driven Ensemble micro-protocol interface, and uses message-passing way of communication. Ensemble's micro-protocols implement basic sliding window protocols, and functionality such as fragmentation and re-assembly, flow control, signing and encryption, group membership, message ordering, etc. The Ensemble system provides an algorithm for calculating the stack, i.e., composing protocol out of micro-protocols, given the set of properties that an application requires. This algorithm encodes knowledge of protocol designers and appears to work quite well, but does not assure generation of a correct stack (the methodology for checking correctness is not automated yet). Thus, Ensemble can be efficiently customized for different protocols, i.e, has a high level of tailorability. In addition, Ensemble gives the possibility of formal optimization of the composed protocol. This is done in Nuprl [64], and also appears to give good results as far as the optimization of a protocol for a particular application goes.

System development with real-time components

Isović et al. [48] defined the development method specifically targeted towards real-time systems regardless of its complexity (it is suitable for the complex as well as for more simple systems). The development method they propose is an extension of the method used for developing real-time systems in Swedish car industry [79]. In this development method it is assumed that:

- The component library contains binaries of COTS, components-of-the-shelf, and description of them, e.g., memory consumption, identification, environment assumptions (processor family on which component operates), and functional description.
- The component library also contains dependencies to other components.
- Components are mapped to tasks, or multiple tasks for more complex components.
- Component communication is done through shared memory, and interfaces are called ports.

The development process for real-time systems is divided in several stages as shown in figure 3.17, and starts with a system specification as an input to the top-level design. At the top-level design stage decomposition of a system into components is performed. A designer browses through the library and designs the system having in mind possible component candidates. At the detailed design stage temporal attributes are assigned to components: period, release times, precedence constraints, deadline, mutual exclusion, and time-budget (a component is required to complete its execution within its time-budget). At the following stage checks are performed to determine if selected components are appropriate for the system, or, if the adaptation of components is required, or new components need to

be developed. Also, at this stage, component interfaces are checked to see if input ports are connected and if their type matches, and that way system integration is performed. If this stage shows that selected components are not appropriate for the system under construction, then a new component needs to be developed. Developed components are placed in libraries for future reuse. The detail design stage, the scheduling/interface check stage, and the top-level design stage can be repeated until proper components for system integration are found (or designed). When the system finally meets the requirements from the specification, the temporal behavior of components must be tested on the target platform to verify if they meet temporal constrains defined in the design phase, i.e., verification of worst case execution time is performed. The method described provides a high degree of

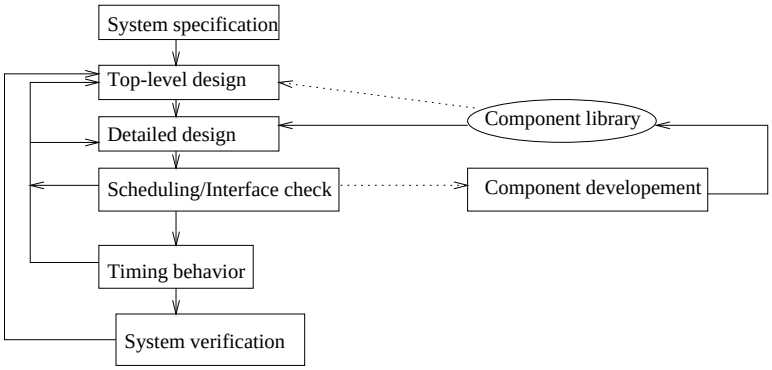


Figure 3.17: Design model for real-time components

tailorability for the developer, since the architecture of the system is not fixed, and components in the system can be exchanged during the design of the system in order to obtaining the most suitable components for that particular system. The developer does have some support while choosing the appropriate component for reuse, given the check stages of the design and dependency checks among component that are stored in libraries. However, such process is not automated.

This method supports analysis of temporal behavior of components (focus is on WCET checks), but how the temporal behavior of the entire system is checked is not clear. Also, this approach is constrained by the assumption that the components in library can be COTS and that temporal behavior of components is not known beforehand. This allows temporal constraints of the component to be defined (or better to say predicted) only at the design time of the system.

VEST

While previously described design method is targeted towards systems that, in most cases, need to be composed out of COTS, and is suitable for more complex real-time systems, VEST [105] aims to enable the construction of the OS-like portion of an embedded real-time system with strengthen resource needs. The development process is fairly good defined and a need for proper configuration and analysis tools is recognized in VEST. The development process offers more flexibility than one presented by Isović et. al. [48], since components in the library are passive and real-time attributes of components are known. System development starts with the design of the infrastructure, which can be saved in a library and used again (see figure 3.18). The infrastructure consists of micro-components such as interrupt handlers, inidirection tables, dispatchers, plug and unplug primitives, proxies for state mapping, etc. The infrastructure represents a framework for composing a system out of components. Configuration tools permits the user to create an embedded

real-time system by composing components into a system, i.e, mapping passive components into run-time structures (tasks). After system is composed, dependency checks are invoked to establish certain properties of the composed system. If the properties are satisfying and the system does not need to be refined, the user can invoke analysis tools to perform real-time, as well as reliability analysis. As can be seen, VEST offers high degree of tailorability for the designer, i.e., a specific system can be composed out of appropriate components as well as infrastructure from the component library. Note that components in

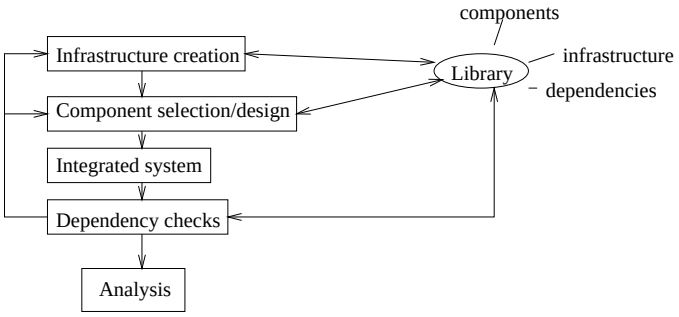


Figure 3.18: Embedded system development in VEST

VEST are passive (collection of code fragments, functions and objects), and are mapped into run-time structures (tasks). Each component can be composed out of subcomponents. For example, task management component can be made of components such as create task, delete task, and set task priority. Components have real-time properties such as worst case execution time, deadline, and precedence and exclusion constraints, which enable real-time analysis of the composed system. In addition to temporal properties, each component has explicit memory needs and power consumption requirements, needed for efficient use in an embedded system. Selecting and designing the appropriate component(s) is fairly complex process, since both real-time and non real-time aspects of a component must be considered and appropriate configuration support has to be available. Dependency checks proposed in VEST are one good way of providing configuration support and the strength of the VEST approach. Due to its complexity dependency checks are broken into 4 types:

- factual: component-by-component dependency checks (worst case execution time, memory, importance, deadline, etc.),
- inter-component: pairwise component checks (interface requirements, version compatibility, is a component included in another, etc.),
- aspects: checks that include issues which affect the performance or semantic of components (real-time, concurrency synchronization and reliability issues), and
- general: checks of global properties of the system (the system should not experience deadlocks, hierarchical locking rules must be followed, etc.).

Having well defined dependency checks is very important since it minimizes possible errors in the system composition. Interface problems in VEST are only identified but are not addressed at all; thus it is not obvious how components can be interconnected. Also, analysis of the system is proposed, but some more concrete solutions are not presented. Finally, it is unclear if the VEST is an implemented solution.

PBO model

A component-based system based on the port-based object (PBO) model can be classified as configurable, and is suitable for development of embedded real-time control software

system [108]. Components from the component library, in addition to newly created ones, can be used for the system assembly. A component is the PBO that is implemented as an independent concurrent process. Components are interconnected through ports, and communicate through shared memory. The PBO defines module specific code, including input and output ports, configuration constants (for adopting components for different applications), the type of the process (periodic and aperiodic), and temporal parameters such as deadline, frequency, and priority. Support for composing a system out of components is limited to general guidelines that are given to the designer for composing a system out of PBO components, and is not automated at all. This approach to componentization is somewhat unique since it gives methods for creating a framework that handles the communication, synchronization and scheduling of each component. Any C programming environment can be used to create components with minimal increase in performance or memory usage. Creating code using PBO methodology is an "inside out" programming paradigm as compared to a traditional coding of real-time processes. The PBO method provides consistent structure for every process and OS system services, such as communication, synchronization, scheduling. Only when necessary, OS calls PBO's method to execute application code. Analysis of the composed system is not considered.

3.4 A tabular overview

This chapter concludes with a tabular summary of investigated component-based systems and their characteristics. Tables 3.3 and 3.4 provide an additional instrument for comparing and analyzing component-based database and component-based embedded real-time systems.
The following symbols are used in the table:

x	—	feature is supported in/true for the system, and
x/p	—	feature is partially supported in/true for the system, i.e, the system fulfills the feature to a moderate extent.

Below follows a description of the criteria.

A. Type of the system We investigated integration of the component-based software engineering for the system development in the following areas:

1. database,
2. embedded,
3. real-time, and
4. embedded real-time.

This criteria illustrates lack of component-based solutions in the area of embedded real-time database systems. As can be seen from table 3.3 there are few component-based systems (also referred as platforms) that can be classified as embedded real-time, and all component-based systems are either embedded and real-time systems, or database systems. There does not exist a component-based database system that can be classified as embedded real-time.

[1]The system aims to support the feature, but only the draft of how the feature should be supported by the system exists, i.e., there is no concrete implementation.

Platforms		DBMS platforms											Embedded (and) real-time platforms						
Characteristics of platforms		[29]	[77]	Oracle8i[83]	DataBlade[47]	UDB2[21]	Sybase[82]	Garlic[97]	DiscoveryLink[42]	OLE DB[75]	CORBAService[85]	KIDS[38]	SPIN[12]	2K[54]	Ensemble[64]	[48]	VEST[105]	PBO [108]	
A. Type of the system	1) database	x	x	x	x	x	x	x	x	x	x	x							
	2) embedded												x		x				
	3) real-time																		
	4) embedded real-time													x		x	x	x	x
B. Status	1) research platform							x			x	x	x	x	x	x	x	x	x
	2) commercial product	x	x	x	x	x	x		x	x									
C. Component granularity	1) system	x	x																
	2) part of the system			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
D. Category of the system	1) extensible			x	x	x	x						x						
	2) middleware							x	x	x				x					
	3) service										x								
	4) configurable											x			x	x	x	x	x
E. Component type	1) domain-specific data type or a new index			x	x	x	x												
	2) wrapper							x	x	x									
	3) service										x		x						
	4) DBMS subsystem											x							
	5) CORBA object																		
	7) microprotocol	x	x											x					
	8) binary														x				
	9) passive															x			
	10) PBO																		x

Table 3.3: Characteristics of component-based database and embedded (and) real-time systems

Platforms		DBMS platforms												Embedded (and) real-time platforms					
Characteristics of platforms		[29]	[77]	Oracle8i[83]	DataBlade[47]	UDB2[21]	Sybase[82]	Garlic[97]	DiscoveryLink[42]	OLE DB[75]	CORBAService[85]	KIDS[38]	SPIN[12]	2K[54]	Ensemble[64]	[48]	VEST[105]	PBO [108]
F.	Real-time properties	1) not preserved	x	x	x	x	x	x	x	x	x	x	x/p	x	x		x	x/p
		2) preserved														x	x^1	
G.	Interface/ communication	1) standardized	x	x						x	x			x				
		2) system specific			x	x	x	x	x			x	x		x			x
		3) ports/unbuffered														x		
H.	Configuration tools	1) not available	x	x							x			x			x^1	x/p
		3) available			x	x	x	x/p	x/p			x	x/p		x	x/p		
I.	Analysis tools	1) not available	x	x	x	x	x	x	x	x	x		x					x
		3) available										x/p		x/p	x/p	x	x/p^1	
J.	Reusability	1) component	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x^1	x
		2) architecture										x					x^1	
K.	Tailoring ability	1) none	x	x										x				
		2) low			x	x	x						x					
		3) moderate						x	x	x	x				x	x	x^1	
		4) high										x						x

Table 3.4: Characteristics of component-based database and embedded (and) real-time systems

B. Status Analyzed component-based platforms are:

1. research platforms, and
2. commercial products.

As can be seen from the table 3.3, most component-based database systems are commercial products, while embedded (and) real-time platforms are all research projects. It is important to notice that database industry has embraced component-based software paradigm and that the need for componentization is increasingly important, since almost all major database vendors provide some component-based solution of their database servers. Also, it is clear that real-time and embedded issues are not integrated in the commercial component-based development, thus, implying that there are still a lot of open research questions that need to be answered, before component-based embedded real-time systems are be commericalized.

C. Component granularity There are two major granularity levels of components:

1. system as a component, and
2. part of the system as a component.

It is noticeable that a database component can be anything from a database system, a component of large granularity, to lightweight components that are composing parts of the DBMS. In contrast, most embedded real-time systems have lightweight components, i.e., parts of the system, and are building an operating system-like portion of embedded (and) real-time systems. Exception is 2K where component (CORBA object) is of larger granularity, and can be an embedded system itself.

D. Category of the system If we emphasize that a database as a component cannot represent a real component-based solution of a database system, then our view is narrowed to a component DBMS (CDBMS) and parts or extensions of the database system as a component. Same can be stated for embedded (and) real-time systems. Hence, investigated component-based systems are classified as follows:

1. extensible,
2. middleware,
3. service, and
4. configurable.

Note that service is the category of CDBMSs, and not of embedded (and) real-time systems.

E. Component type A component in a component-based database system and a component-based embedded (and) real-time system is one of the following :

1. domain-specific data type or a new index,
2. wrapper,
3. service,
4. DBMS subsystem,
5. CORBA object,
6. microprotocol,
7. binary component,

8. passive component, and
9. PBO (port-based object).

The first four component types are typically components found in database systems. We refer to these as database components. The last five types of components are mostly found in embedded (and) real-time systems.

CDBMSs mainly consist of components that implement a certain database functionality (or extend existing functionality), usually mapped into services. Thus, a database component provides certain database service or function. For example, in Oracle8*i* the spatial data cartridge component implements non-standard DBMS functionality such as spatial index. Also, in KIDS, the DBMS subsystem component, transaction management, can provide several related services such as transaction, serialization and validation services, and in CORBAservices one component is one database service.

Components in embedded (and) real-time systems are more diverse. In some systems components are not explicitly defined, and can only be classified as passive components (VEST) or as binary components (in development methodology introduced by Isović et al.[48]). Systems such as SPIN and Ensemble have more specific components, extensions for application-specific operating system services (SPIN) and microprotocols (Ensemble). Only 2K has standardized components, CORBA objects.

Note that almost every system has its own notion and a definition of a component, which suites the system's purpose and requirements.

F. Real-time properties Component-based database and embedded (and) real-time systems may:

1. not preserve, and
2. preserve

real time properties. Component-based database systems do not enforce real-time behavior (see table 3.4). In addition, issues related to embedded systems such as low-resource consumption are not addressed at all. Accent in a database component is on providing a certain database functionality. In contrast, component (or components) in existing component-based embedded real-time systems is usually assumed to be mapped to a task, i.e., passive components [105], binary components [48] and PBO components [108] are all mapped to a task. This comes naturally in the real-time research community because a task, i.e., concurrent process, is the smallest schedulable unit in a real-time system. Therefore, analysis of real-time components in these solutions addresses the problem of managing temporal attributes at a component level by considering them as task attributes [48, 105, 57]:

- worst-case execution time (WCET) - the longest possible time it takes to complete a task,
- release time - the time at which all data that are required to begin executing are available,
- deadline - the time by which a task must complete its execution,
- precedence constraints and mutual exclusion - specify if a task needs to precede or exclude other tasks, and
- period - if a task is periodic.

Components in the PBO model [108] have only two temporal attributes, frequency and deadline. Components, in the development method for component-based real-time systems introduced by Isović et al. [48], are assumed to have all of the above listed temporal

attributes, and these attributes are predicted in the design phase of the system development. VEST includes full list of temporal attributes in its components and expands that list with attributes essential for correct behavior of components in an embedded environment:

- power consumption requirements, and
- memory consumption requirements.

This makes the VEST approach the most flexible embedded real-time approach, since component attributes include real-time and embedded needs, and are known for every component in the library (this was not the case in the development method for component-based real-time systems introduced by Isović et al. [48]).

A component in an embedded real-time system must have well defined temporal properties, i.e., component behavior must be predictable, in order to be able to compose a reliable embedded real-time system. Hence, to be able to develop a database suitable for an embedded real-time system out of components, such components would have to have well-defined temporal properties and resource requirements, e.g., memory and power consumption, issues not addressed in current database components. Thus, we conclude that a database component used in an embedded real-time system would have to be a real-time component, and as such, mapped to a task in order to ensure predictability of the component and of the composed embedded real-time system, i.e., we need to know, for example, worst-case execution time for a database component to be able to ensure predictability of the component, and, in turn, of the database system composed out of such components.

G. Interfaces / communication The component communicates with its environment (other components) through well-defined interfaces. Generally, existing component-based database and embedded (and) real-time solutions use:

1. standard interfaces,
2. system specific, and
3. unbuffered interfaces (ports), in which case they use communication through shared memory.

For example, standardized interfaces defined in the IDL are used in CORBAservices and in 2K. Also, OLE DB interface is used in the Microsoft's Universal Data Access architecture. Interfaces developed within the system (we refer to them as system specific) are used in other systems, e.g., Oracle8*i* has extensibility interfaces and KIDS has component-specific interface. Systems that aim to allow more flexible composition use event-based communication, e.g., KIDS, SPIN, Ensemble, and 2K. Inter-component communication in database systems and embedded real-time systems have different goals. Interfaces in embedded real-time systems must be such that inter-component communication can be performed in a timely predictable manner. There are two possible ways of a real-time component communication:

- Buffered communication. The communication is done through message passing, e.g., [64].
- Unbuffered communication. Unbuffered data is accessed through shared memory, e.g., [107, 48].

Note that most component-based database systems use buffered communication, since predictability of communication is not of importance in such systems. Systems enforcing real-time behavior use unbuffered communication (exception is VEST where interfaces of components are not defined). This is because of several disadvantages that are identified in buffered communication [45, 48]:

- Sending and receiving messages incur significant overhead.
- Tasks waiting for data might block for an undetermined amount of time.
- Crucial messages can get lost as a result of the buffer overflow if tasks do not execute at the same frequency.
- Sending messages in control systems, which have many feedback loops, creates risk for deadlock.
- The upper bound on the number of produced/consumed messages must be determined to enable guarantee of temporal properties.

Generally, a real-time system that uses buffered communication is difficult to analyze due to dependencies among tasks. Unbuffered communication eliminates direct dependencies between tasks, since they only need to bind to a single element in the shared memory. Communication through shared memory incurs fewer overheads as compared to a message-passing system. Also, it is easier to check system's temporal behavior if unbuffered communication is used [48]. Hence, unbuffered style of communication is preferred style of communication in embedded real-time systems. It is suggested that interfaces in (hard) real-time systems should be unbuffered [48]. Unbuffered interfaces are called ports.

H. Configuration tools The development process must be well defined to enable efficient system assembly out of existing components from the component library or newly created ones. Adequate and automated configuration support must exist to help system designer with this process, e.g., rules for composing a system out of components, support for selection of an appropriate component from the component library, and support for the development of new components. However, in some systems configuration tools are not available. Hence, we identify that configuration tools in a component-based database and embedded (and) real-time system are:

1. not available, and
2. available.

Observe (table 3.4) that most extensible CDBMSs have available configuration support. Since extensible systems already have fixed architecture and pre-defined extensions, provided configuration tools are sufficient to enable development and integration of components into an extensible system. The Garlic middleware technology provides a simple and fairly easy development of new components (wrappers), but these components are with simple functionality and structure. The situation gets more complicated if we consider OLE DB middleware, where not only components, data providers, can be added or exchanged in the architecture, but the architecture can also be extended to use customizable data providers. Hence, adequate configuration support is needed to ensure interoperability of different data providers. However, when developing a data provider component the developer is only encouraged to follow specification exactly in order to ensure interoperability with other data providers, i.e, configuration support available but is not enough. OLE DB is also limited in terms of use, it can only be used in Microsoft's computing environments.

On the other hand, extensible and middleware embedded (and) real-time systems do not provide good configuration support. 2K, for example, does not have reusable components in its architecture and provides automated configuration (and reuse) of components from different systems. The correctness of integrated 2K middleware is only assumed (based on checks of inter-component dependencies). Also, SPIN offers very little configuration support, since correctness, configuration, and integration of components in the core system is based on features of the extension language in which components are developed. The rules for composition of a system are not defined in 2K and SPIN, and these systems can be viewed as the first generation of a component-based embedded (and) real-time systems.

In general, demands on development support in configurable systems are high. KIDS has met these demands to some extent, with a well-defined development process. In most configurable embedded real-time systems, some configuration support is also provided. For example, PBO model gives good guidelines to help designer when composing system out of components.The development method for component-based real-time systems introduced by Isović et al. [48] provides configuration support for choosing appropriate component for system composition, i.e., checks of temporal properties and interfaces of components are performed to ensure that the component from the library is suitable for the system under development. In VEST, the necessity of having good configuration tools is recognized. Composition rules are defined through four types of dependency checks. Note that the development introduced by Isović et al. [48] and 2K have only inter-component dependency checks, which are just one out of four types of checks proposed in VEST. This makes the VEST approach the most appropriate one with respect to the correct system composition, because the more dependencies are checked in the system, within components, the probability of errors in the composed system, i.e., compositional errors, is minimized.

I. Analysis tools Since the reliability of the composed system depends on the level of correctness of the component, analysis tools are needed to verify the behavior of the component and the composed system. In particular, real-time systems must meet their temporal constraints, and adequate analysis to ensure that a system has meet temporal constraints is required. Thus, analysis tools are:

1. not available, and
2. available.

The problem of analysis of the composed component-based database system is rather straightforward, in most cases, analysis of the composed system is unavailable (see table 3.4). Importance of having good analysis of the composed system is recognized in KIDS, but is not pursued beyond that, i.e., analysis tools are not provided. Component-based embedded (and) real-time systems do not provide analysis of the composed system as well. That is true for SPIN, 2K, and systems based on the PBO model. VEST introduces notion of reliability and real-time analysis of the system, but does not give more detailed description of such analysis. In the development method introduced by Isović et al. [48] checks of the WCET of components are performed.

G. Reusability Systems are composed out of reusable components from the library. Architecture of the system can be reused in the system's development as well. Thus, in component-based database and embedded (and) real-time systems parts that can be reused are:

1. component, and
2. architecture.

As can be seen from the table 3.4, the part that can be reused in all systems is the component. KIDS and VEST can also reuse the architecture, as opposed to other systems. If we consider that the literature [67, 76] points to reusability of architecture as a way to receive a higher pay-off, KIDS and VEST have significantly higher degree of reusability as compared to others. The fact that an architecture of a system can be reused is also important from a different aspect, early system testing. For example, even though it does not have any analysis tools, KIDS offers good basis for the early system testing, in the first phase of the construction process, at the architecture level of the system development.

H. Tailoring ability The benefit of using component-based development in database systems is customization of the database for different applications. There are four degrees of tailorability in component-based database and embedded (and) real-time systems:

1. none,
2. low,
3. moderate, and
4. high.

It is noticeable that extensible systems have low tailorability, middleware moderate, while configurable systems have high tailorability (see table 3.4). Since the goal is to provide an optimized database for a specific application with low development costs and short time-to-market, it is safe to say that configurable systems are the most suitable in this respect. In particular, VEST and KIDS, since they allow architecture to be saved and reused. At the same time, the methodology introduced in KIDS and VEST should enable tailoring of one generic system for a variety of different applications.

Chapter 4

Summary

This final chapter starts with a summary of the main issues we have identified in the report with respect to current state-of-the-art in the area of embedded databases for embedded real-time systems (section 4.1). In the last section (section 4.2) we identify challenges for future work.

4.1 Conclusions

Embedded systems, real-time systems, and database systems are research areas that have been actively studied. However, research on embedded databases for embedded real-time systems, explicitly addressing the development and design process, is sparse. A database that can be used in an embedded real-time system must handle transactions with temporal constraints, and must, at the same time, be suitable for embedded systems with limited amount of resources, i.e., the database should have small footprint, be portable to different operating system platforms, have efficient resource management, and be able to recover from a failure without external intervention.

As we have shown in this report, there are a variety of different embedded databases on the market. However, they vary significantly in their characteristics. Differences in products are typically the way data are organized in the database (data model), the architecture of the database (DBMS model), and memory usage. Commercial embedded databases also provide different interfaces for applications to access the database, and support different operating system platforms. Application developers must choose carefully the embedded database their application requires. This is a difficult, time consuming and costly process, with a lot of compromises. One solution could be to have a more generic embedded database platform that can be tailored and optimized such that it is suitable for different applications. Existing real-time database research platforms are unsuitable in this respect, since they are mainly monolithic systems, and, as such, they are not easily tailored for new applications having different or additional requirements.

Satisfying requirements put on the embedded database in an embedded real-time system calls for a new way of designing and developing an embedded database for application specific system. For this purpose we have examined component-based software engineering paradigm, since it has been successfully applied in conventional non real-time environments.

Building parts any component-based system are components. However, traditional component-based systems, e.g., systems based on COM or CORBA component framework, normally view an entire database system as one component. We emphasize that the component-based database system is the database system, which building parts are components, and refer to such building parts as database components. Components in a database system usually implement certain database functionality (or extend existing functional-

ity), and these are normally mapped into services. In contrast, components in embedded real-time systems are developed to have well-defined temporal properties, e.g., worst-case execution time, deadline and release time, and are mapped one task or multiple tasks. We have shown that a real-time database component, i.e., a database component with temporal constraints that represents the intersecting type of the database and real-time component, does not exist. Existing component-based database systems do not enforce real-time behavior, and issues related to embedded systems such as low-resource consumption are not addressed at all in these solutions. However, in order to compose a reliable embedded real-time system out of components, each component's behavior must be predictable. This is the reason why components in existing component-based real-time systems are usually assumed to be mapped to a task with well-defined temporal properties. To ensure predictability and reliability of a database in an embedded real-time system, such database would have to be composed out of components, with well-defined and known temporal properties of a component must be well-defined and known. This, in turn, would require such a component to be mapped to a task.

We have observed that the architecture of component-based systems vary from fixed (extensible systems), to completely configurable. In the extensible systems, which have fixed architecture, extensions are allowed only in well-defined places of the architecture. In the configurable systems components can be freely added or exchanged in the architecture. Configurable systems allow significant amount of flexibility to the developer and the user, in comparison to other systems. Hence, configurable systems represent the preferred type of component-based systems, as they allow the embedded database to be tailored for new applications with different requirements. Since configurable systems offer the highest degree of tailorability, and are the most flexible ones, we focus on these systems; in particular KIDS (component-based database system) and VEST (component-based embedded real-time system). Higher pay-off and quickness to market can be achieved if the architecture can be stored in a library and reused, as well as components (this feature can also be found in configurable systems such as KIDS and VEST). Moreover, to have an architectural abstraction of a system would enable early system testing. Temporal analysis of systems and components could be done in an early stage of the system development, thus, reducing development costs, and enhancing reliability and predictability of the system. For example, in KIDS the architecture is reusable, and even though it does not have any analysis tools, KIDS offers good basis for the early system testing; in the first phase of the construction process, at the architecture level of the system.

To ensure minimum errors in the system composition, the most efficient existing approach is to have a large number of dependency checks, as in VEST, where four types of dependency checks are introduced, e.g., factual, inter-component, aspects and general. Note that some systems, such as 2K, have only one type of dependency checks, i.e., inter-component dependency checks. Other systems do not have dependency checks at all, which makes them more vulnerable for compositional errors.

When composing an embedded database system out of components, the issue of inter-component communication should be carefully handled. Components in embedded real-time systems must communicate in a timely predictable manner, and for that reason most embedded real-time systems use unbuffered communication. In contrast, most existing component-based database systems have buffered communication (message passing), and are not concerned with predictability of inter-component communication. Furthermore, most component-based database systems use standardized interfaces, e.g., IDL, OLE DB, which makes them suitable for easier exchange and addition of new components (they must conform to a standard), while embedded real-time systems mostly have system-specific interfaces, i.e., interfaces defined within the system.

We have studied components and their definitions in component-based software engineering in particular, as well as in component-based database and embedded real-time systems. We found that every component-based system has its own notion and a definition of a component. Thus, the component is to a large extent an arbitrary notion, and

is practically re-defined and re-invented in each system. However, there are at least three common requirements that a component must satisfy in any component-based system (i) to be a composing part of the system; (ii) to be reusable with well-defined conditions and ways of reuse; and (iii) to have well-defined interfaces for inter-connection with other components. Furthermore, rules for composition of a system must be defined, and (automated) tools to assist the developer must be available. Although most of the component-based database and embedded real-time systems focus on components, their solutions lack good composition rules, and automated development support.

4.2 Future work

It is evident that research for embedded databases that explicitly addresses (i) development and design process, and (ii) limited amount of resources, inherits challenges from the traditional real-time systems, embedded systems, and component-based software engineering.

Key challenges in component-based software engineering include:

- developing a component that can be reused as many times as possible, and would conform to a standard,
- determining how a component should be verified to obtain appropriate information in order to determine suitability of a component for a specific system,
- defining rules for for the composition of a reliable system, which satisfies specification requirements,
- analyzing the behavior of the composed system, as well as ensuring trade-off analysis between conflicting requirements in the early design stage, and
- providing adequate configuration support for development process, in particular determining what is doable by tools and what needs to be left to the designer.

The above research challenges in a component-based software engineering are further augmented for embedded real-time systems. The additional challenges include:

- defining a component with predictable behavior,
- determining how inter-component communication should be performed, and what are appropriate interfaces for real-time components,
- managing resources such as memory usage and required processing time for different operations of a component, possibly on different hardware platforms and operating system platforms, and
- determining what are the appropriate analysis tools for analysis of system resource demands, and analysis of the timing properties of the composed system.

Embedding a database in a real-time system also brings a set of challenges:

- determining what type of the DBMS architecture is appropriate (library vs client-server) to ensure predictability in a real-time system,
- determining the most appropriate data model for the embedded database, and
- determining which type of interfaces, through which user is accessing a database, is the most suitable.

Finally, fundamental research questions when developing an embedded database for an embedded real-time system using component-based software engineering include:

- defining a real-time database component, i.e., functionality and services that a component should provide, and temporal constraints a component should satisfy, and
- determining which component interfaces are appropriate, and how inter-component communication should be performed (message passing vs shared memory).
- integrating the composed database into an embedded real-time system.

Our research is focused on providing an experimental research platform for building embedded databases for embedded real-time systems. At a high-level, the platform consists of two parts. First, we intend to develop a component library, which holds a set of methods, that can be used when building an embedded database. Initially, we will develop a set of components that deal with concurrency control, scheduling, and main-memory techniques. At the next step, we develop tools that, based on the application requirements, will support the designer when building an embedded database using these components. More importantly, we want to develop application tools and techniques that: (i) support the designer in the composition and tailoring of an embedded database for a specific system using the developed components, where the application requirements are given as an input; (ii) support the designer when analyzing the total system resource demand of the composed embedded database system; and (iii) help the designer by recommending components and methods if multiple components can be used, based on the application requirements. Further, such a tool will help the designer to make trade-off analysis between conflicting requirements early in the design phase.

The components should carry property descriptions of themselves with information about (i) what service and functionality the component provides; (ii) what quality of service guarantees facilitated by the component, e.g., techniques may be applicable to soft real-time applications but not hard real-time applications; (iii) their system resource demand, e.g, memory usage and required processing time for different operations and services provided by a component, possibly on different hardware platforms in order to simplify interoperability; and (iv) composition rules, i.e., which specifies how, and with which components, a component can be combined.

Our research should give better understanding of the specification of components to be used in an embedded and real-time setting. This includes functionality provided by the component, the resource demand required by a component when executed on different platforms, and rules for specifying how components can be combined and how the overall system can be correctly verified given that each component has been verified.

Bibliography

[1] *Proceedings of International Workshop, Database in Telecommunications*, number 1819 in Lecture Notes in Computer Science, Edinburgh, Scotland, UK, September 1999. Springer-Verlag.

[2] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active views for electronic commerce. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 138–149, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann. ISBN 1-55860-615-7.

[3] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993.

[4] B. Adelberg, B. Kao, and H. Garcia-Molina. Overview of the STanford Real-Time Information Processor (STRIP). *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(1), 1996.

[5] Brad Adelberg, Hector Garcia-Molina, and Ben Kao. Emulating soft real-time scheduling using traditional operating system schedulers. In *Proceedings of IEEE Real-Time System Symposium*, 1994.

[6] Brad Adelberg, Hector Garcia-Molina, and Jennifer Widom. The STRIP rule system for efficiently maintaining derived data. In *SIGMOD*, pages 147–158, 1997.

[7] B.S. Adelberg. *STRIP: A Soft Real-Time Main Memory Database for Open Systems*. PhD thesis, Stanford University, 1997.

[8] Q.N Ahmed and S.V. Vrbsky. Triggered Updates for Temporal Consistency in Real-Time Databases. *Real-Time Systems*, 19(3):209–243, November 2000.

[9] S. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. Deeds towards a distributed and active realtime database system, 1996.

[10] F. Baothman, A.K Sarje, and R.C Joshi. On Optimistic concurrency control for RT-DBS. *IEEE Region 10 International Conference on Global Connectivity in Energy, Computer, Communication and Control*, 1998.

[11] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice*. SEI Series in Software Engineering. Addison Wesley, 1998.

[12] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN - an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA, USA, February 1994.

[13] L. Blair and G. Blair. A tool suite to support aspect-oriented specification. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP '99*, pages 7–10, Lisbon, Portugal, June 1999. A position paper.

[14] J. A. Blakeley. OLE DB: A component DBMS architecture. In *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, pages 203–204, New Orleans, Louisiana, USA, March 1996. IEEE Computer Society Press.

[15] J. A. Blakeley. Universal data access with OLE DB. In *Proceedings of the 42nd IEEE International Computer Conference (COMPCON)*, pages 2–7, San Jose California, February 1997. IEEE Computer Society Press.

[16] J. A. Blakeley and M. J. Pizzo. *Component Database Systems*, chapter Enabling Component Databases with OLE DB. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[17] J. Bosch. *Design and Use of Software Architectures*. ACM Press in collaboration with Addison-Wesley, 2000.

[18] A. P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH active OODBMS. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*.

[19] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 117–128. IEEE Computer Society Press, 1995.

[20] R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1):5–50, 1996.

[21] M. J. Carey, L. M. Haas, J. Kleewein, and B. Reinwald. Data access interoperability in the IBM database family. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Interoperability*, 21(3):4–11, 1998.

[22] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas II, J. H. Williams, and E. L. Wimmers. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proceedings of the 5th International Workshop on Research Issues in Data Engineering: Distributed Object Management (RIDE-DOM)*, pages 124–131, Taipei, Taiwan, March 1995. IEEE Computer Society. ISBN 0-8186-7056-8.

[23] Documentation available at: http://i44w3.info.uni-karlsruhe.de/ compost/, Jun 2001. COMPOST is currently under development at University of Karlsruhe, Germany, and University of Linköping, Sweden.

[24] I. Crnkovic and M. Larsson. A case study: Demands on component-based development. In *Proceedings of 22th International Conference of Software Engineering*, pages 23–31, Limeric, Ireland, June 2000. ACM.

[25] I. Crnkovic, M. Larsson, and F. Lüders. State of the practice: Component-based software engineering course. In *Proceedings of 3rd International Workshop of Component-Based Software Engineering*. IEEE Computer Society, January 2000.

[26] C.J Date. *An Introduction to Database Systems*. Addison-Wesley, 2000.

[27] J. R. Davis. Creating an extensible, object-relational data management environment: IBM's DB2 Universal Database. Database Associated International, InfoIT Services, November 1996. Available at http://www.dbaint.com/pdf/db2obj.pdf.

[28] K. R. Dittrich and A. Geppert. *Component Database Systems*, chapter Component Database Systems: Introduction, Foundations, and Overview. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[29] A. Dogac, C. Dengi, and M. T. Öszu. Distributed object computing platform. *Communications of the ACM*, 41(9):95–103, 1998.

[30] ENEA Data. OSE Real-time system. http://www.enea.se.

[31] J. Eriksson. Specifying and managing rules in an active real-time database system. *Licenciate thesis*, 1998.

[32] J. Eriksson. Real-time and active databases: A survey. In *Lecture notes in Computer Science*, 2000.

[33] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *The communications of the ACM*, 19, 1976.

[34] S. Chakravarthy et al. HiPAC: A research project in active, time-constrained database management. Technical Report XAIT-8902, Xerox Advanced Information Technology, 1989.

[35] FairCom Corp. c-tree Plus. http://www.faircom.com.

[36] W. Fleisch. Applying use cases for the requirements validation of component-based real-time software. In *Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 75–84, Saint-Malo, France, May 1999. IEEE.

[37] A. Gal. Data management in ecommerce (tutorial session): the good, the bad, and the ugly. In *Proceedings of the 2000 ACM SIGMOD on Management of Data*, volume 34, page 587, Dallas, TX, USA, May 2000. ACM.

[38] A. Geppert, S. Scherrer, and K. R. Dittrich. KIDS: Construction of database management systems based on reuse. Technical Report ifi-97.01, Department of Computer Science, University of Zurich, September 1997.

[39] J. Gray. Notes on database operating systems. In *Lecture notes in Computer Science*, 1978.

[40] Ashish Gupta, Venky Harinarayan, and Anand Rajaraman. Virtual database technology. In *Proceedings of 14th International Conference on Data Engineering*, pages 297–301, 1998.

[41] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[42] L. M. Haas, P. Kodali, J. E. Rice, P. M. Schwarz, and W. C. Swope. Integrating life sciences data-with a little Garlic. In *Proceedings of the IEEE International Symposium on Bio-Informatics and Biomedical Engineering (BIBE)*, pages 5–12. IEEE, 2000.

[43] J. Hansson. Dynamic real-time scheduling in ose delta, 1994.

[44] J. R. Haritsa and K. Ramamritham. Real-time database systems in the new millennium. *Real-Time Systems*, 19(19):205–208, November 2000. The International Journal of Time-Critical Computing Systems.

[45] M. Hassani and D. B. Stewart. A mechanism for communication in dynamically reconfigurable embedded systems. In *Proceedings of High Assurance Software Engineering (HASE) Workshop*, pages 215–220, Washington DC, August 1997.

[46] J. Huang, J.A Stankovic, K. Ramamritham, and D. Towsley. Experimental Evaluation of Real-time Optimistic Concurrency Control Schemes. In *Proceedings of VLDB*, Barcelona, Spain, September 1991.

[47] Developing DataBlade modules for Informix-Universal Server. Informix DataBlade Technology. Informix Corporation, 22 March 2001. Available at http://www.informix.com/datablades/.

[48] D. Isovic, M. Lindgren, and I. Crnkovic. System development with real-time components. In *Proceedings of ECOOP Workshop - Pervasive Component-Based Systems*, France, June 2000.

[49] J. Lindström and T. Niklander and P. Porkka and K. Raatikainen. A Distributed Real-Time Main-Memory Database for Telecommunication. In *Lecture Notes in Computer Science; 1819*, 1999.

[50] J. Stankovic and S. Son and J. Liebeherr. BeeHive: Global multimedia database support for dependable, real-time applications. In *Second Workshop on Active Real-Time Databases*, 1997.

[51] K. Ramamritham. Real-Time Databases. *Proceedings of International Journal of distributed and Parallel Databases*, 1996.

[52] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[53] Y-K Kim, M.R. Lehr, D.W. George, and S.H. Song. A database server for distributed real-time systems: Issues and experiences. In *In proceedings of the Second IEEE Workshop on Parallel and Distributed RealTime Systems*, 1994.

[54] F. Kon, R. H. Campbell, F. J. Ballesteros, M. D. Mickunas, and K. Nahrsted. 2K: A distributed operating system for dynamic heterogeneous environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, August 2000.

[55] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhales, and R. H. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in Lecture Notes in Computer Science, pages 121–143, New York, USA, April 2000. Springer.

[56] F. Kon, A. Singhai, R. H. Campbell, and D. Carvalho. 2K: A reflective, component-based operating system for rapidly changing environments. In *Proceedings of the ECOOP Workshop on Reflective Object Oriented Programming and Systems*, volume 1543 of *Lecture Notes in Computer Science*, Brussels, Belgium, July 1998. Springer.

[57] C. M. Krishna and K. G. Shin. *Real-time Systems*. McGraw-Hill Series in Computer Science. The McGraw-Hill Companies, Inc., 1997.

[58] H.T. Kung and J.T Robinsson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6, 1981.

[59] Tei-Wei Kuo, Chih-Hung Wei, and Kam-Yiu Lam. Real-Time Data Access Control on B-Tree Index Structures. In *Proceedings of the 15th International Conference on Data Engineering*, 1999.

[60] M. Larsson and I. Crnkovic. New challenges for configuration management. In *Proceedings of System Configuration Management, 9th International Symposium (SCM-9)*, volume 1675 of *Lecture Notes in Computer Science*, pages 232–243, Toulouse, France, August 1999. Springer.

[61] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Kyoto*, 1986.

[62] W. Litwin. Linear hashing: A new tool for file and table addressing, 1977.

[63] Bohua Liu. Embedded real-time configuration database for an industrial robot. Master's thesis, Linkoping University, 2000.

[64] X. Liu, C. Kreitz, R. Renesse, J.Hickey, M. Hayden, K. Birman, and R. Constable. Building reliable, high-performance communication systems from components. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, volume 34, pages 80–92, December 1999. Published as Operating Systems Review.

[65] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-tree or b-tree: Main memory database index structure revisited. *11th Australasian Database Conference*, 2000.

[66] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995. Special Issue on Software Architecture.

[67] C. H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman. An approach to software architecture analysis for evolution and reusability. In *Proceedings of CASCON*, Toronto, ON, November 1997. IBM Center for Advanced Studies.

[68] M.A Olson. Selecting and implementing an Embedded Database System. In *IEEE Computer*, 2000.

[69] Uwe Aßman. Invasive software composition with program transformation. To be published.

[70] MBrane Ltd. RDM Database Manager. http://www.mbrane.com.

[71] MBrane Ltd. Velocis Database Manager. http://www.mbrane.com.

[72] N. Medvedovic and R. N. Taylor. Separating fact from fiction in software architecture. In *Proceedings of the Third International Workshop on Software Architecture*, pages 10–108. ACM Press, 1999.

[73] J. Mellin, J. Hansson, and S. Andler. Refining timing constraints of application in deeds. In *In S. H. Son A. Bestavros, K-J Lin, editor, Real-Time Database Systems: Issues and Applications*, pages 325–343, 1997.

[74] B. Meyer and C. Mingins. Component-based development: From buzz to spark. *IEEE Computer*, 32(7):35–37, July 1999. Guest Editors' Introduction.

[75] Universal data access through OLE DB. OLE DB Technical Materials. OLE DB White Papers, 12 April 2001. Available at http://www.microsoft.com/data/techmat.htm.

[76] R. T. Monroe and D. Garlan. Style-based reuse for software architectures. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 84–93. IEEE Computer Society Press, April 1996.

[77] A. Münnich, M. Birkhold, G. Färber, and P. Woitschach. Towards an architecture for reactive systems using an active real-time database and standardized components. In *Proceedings of International Database Engineering and Application Symposium (IDEAS)*, pages 351–359, Montreal, Canada, August 1999. IEEE Computer Society Press.

[78] T. Niklander and K. Raatikainen. RODAIN: A Highly Available Real-Time Main-Memory Database System. In *In proceedings of the IEEE International Computer Performance and Dependability Symposium*, page 271, 1998.

[79] Christer Norström, Mikael Gustafsson, Kristian Sandström, Jukka Mäki-Turja, and Nils-Erik Bånkestad. Findings from introducing state-of-the-art real-time techniques in vehicle industry. In *Industrial session of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[80] Objectivity inc. Objectivity/DB. http://www.objectivity.com.

[81] M. A. Olson. Selecting and implementing an embedded database system. *IEEE Computers*, 33(9):27–34, September 2000.

[82] S. Olson, R. Pledereder, P. Shaw, and D. Yach. The Sybase architecture for extensible data management. *Data Engineering Bulletin*, 21(3):12–24, 1998.

[83] All your data: The Oracle extensibility architecture. Oracle Technical White Paper. Oracle Corporation. Redwood Shores, CA, February 1999.

[84] H. Ossher and P. Tarr. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on object-oriented programming systems, languages, and applications*, pages 411–428, Washington, USA, September 26 - October 1 1993. ACM Press.

[85] M. T. Özsu and B. Yao. *Component Database Systems*, chapter Building Component Database Systems Using CORBA. Data Management Systems. Morgan Kaufmann Publishers, 2000.

[86] Thomas Padron-McCarthy. Master thesis proposal: Configuration database for an industrial robot. http://www.ida.liu.se/labs/edslab/master_proposals/abb-robotics-english.html, Febr 1999.

[87] P. Pardyak and B. N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Operating Systems Review, Special Issue, pages 201–212, Seattle WA, USA, October 1996. ACM and USENIX Association. ISBN 1-880446-82-0.

[88] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering*, volume 17, 1992.

[89] Persistence software inc. PowerTier. http://www.persistence.com.

[90] Pervasive Software Inc. Pervasive.SQL 2000 Reviewers guide. Technical white paper.

[91] Pervasive Software Inc. Pervasive.SQL Database Manager. http://www.pervasive.com.

[92] POET software corp. POET Object. http://www.poet.com.

[93] Polyhedra Plc. Polyhedra database manager. http://www.polyhedra.com.

[94] DARPA ITO projects. Program composition for embedded systems. http://www.darpa.mil/ito/research/pces/index.html, 7 August 2001.

[95] K. Raatikainen and J. Taina. Design issues in database systems for telecommunication services. In *In proceedings of IFIP-TC6 Working conference on Intelligent Networks, Copenhagen, Denmark*, pages 71–81, 1995.

[96] Raima Corporation. Databases in real-time and embedded systems. Technical white paper.

[97] M. T. Roth and P. M. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 266–275, Athens, Greece, August 1997. Morgan Kaufmann. ISBN 1-55860-470-7. Available at http://dblp.uni-trier.de.

[98] S. Ortiz Jr. Embedded databases come out of hiding. In *IEEE Computer*, 2000.

[99] S.F Andler and J. Hansson and J. Eriksson and J.Mellin and M. Berndtsson and N.Elftring. DeeDS Towards a Distributed and Active Real-Time Database System. *ACM Sigmon Record*, Volume 25, 1996.

[100] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[101] R. Sivasankaran, J. Stankovic, D Towsley, B. Purimetla, and K. Ramamritham. Priority assignment in real-time active databases. *The VLDB Journal*, Volume 5, 1996.

[102] Sleepycat Software Inc. Berkeley DB. http://www.sleepycat.com.

[103] X. Song and J. Liu. Maintaining temporal consistency: Pessimistic vs. optimistic concurrency control. *IEEE Transactions on Knowledge and data engineering*, 7, October 1995.

[104] J Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21, Oct 1988.

[105] J. Stankovic. VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems. Technical Report CS-2000-19, Department of Computer Science, University of Virginia, Charlottesville, VA, May 2000.

[106] John A. Stankovic, Sang Hyuk Son, and Jorg Liebeherr. Beehive: Global multimedia database support for dependable, real-time applications. In *Active and Real-Time Database Systems*, pages 51–69, 1997.

[107] D. B. Stewart, M. W. Gertz, and P. K. Khosla. Software assembly for real-time applications based on a distributed shared memory model. In *Proceedings of the 1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop*, pages 214–224, Silver Spring, MD, July 1994.

[108] D. S. Stewart. Designing software components for real-time applications. In *Proceedings of Embedded System Conference*, San Jose, CA, September 2000. Class 408, 428.

[109] D. S. Stewart and G. Arora. Dynamically reconfigurable embedded software - does it make sense? In *Proceedings of IEEE Intl. Conf. on Engineering of Complex Computer Systems (ICECCS) and Real-Time Application Workshop (RTAW)*, pages 217–220, Montreal, Canada, October 1996.

[110] Sybase Inc. SQL.Anywhere. http://www.sybase.com.

[111] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[112] J. Taina and K. Raatikainen. Rodain: A real-time object-oriented database system for telecommunications. In *In proceedings of the DART'96 workshop*, pages 12–15, 1996.

[113] TimesTen Performance Software. TimesTen DB. http://www.timesten.com.

[114] H. Tokuda and C. Mercer. ARTS: A Distributed Real-Time Kernel. In *ACM Operating systems review, 23(3)*, 1989.

[115] A. Wall. Software architecture for real-time systems. Technical Report 00/20, Mälarden Real-Time Research Center, Mälardalen University, Västerås, Sweden, May 2000.

[116] David L. Wells, José A. Blakeley, and Craig W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, 1992.

[117] M. Xiong, R. Sivasankran, J. Stankovic, K. Ramamritham, and D. Towsley. Scheduling transaction with temporal constraints: Exploiting data semantics. In *In Proceedings of the 17th IEEE Real-time Systems Symposium*, 1996.

[118] Y.K. Kim et al. A database server for distributed real-time systems: Issues and experiences. Technical report, Dept. of computer science, University of Virginia, 1994.

[119] P.S Yu, K.Wu, K. Lin, and S.H. Son. On real-time databases: Concurrency control and scheduling. In *Proceedings of the IEEE*, 1994.

[120] G. Zelesnik. *The UniCon Language User Manual*. Carnegie Mellon University, Pittsburgh, USA, May 1996. Available at http://www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual/.

[121] J. Zimmermann and A. P. Buchmann. REACH. In *N. Paton (ed): Active Rules in Database Systems, Springer-Verlag*, 1998.

Using Deterministic Replay for Debugging of Distributed Real-Time Systems

Henrik Thane and Hans Hansson
Mälardalen Real-Time Research Centre, Department of Computer Engineering
Mälardalen University, Västerås, Sweden, hte@mdh.se

Abstract

Cyclic debugging is one of the most important and most commonly used activities in program development. During cyclic debugging, the program is repeatedly re-executed to track down errors when a failure has been observed. This process necessitates reproducible program executions. Applying classical debugging techniques such using breakpoints or single stepping in real-time systems change the temporal behavior and make reproduction of the observed failure during debugging less likely, if not impossible. Consequently, these techniques are not directly applicable for cyclic debugging of real-time systems.

In this paper we present a novel software-based approach for cyclic debugging of distributed real-time systems. By on-line recording significant system events, and off-line deterministically replaying them, we can inspect the real-time system in great detail while still preserving its real-time behavior.

Keywords: Determinism, debugging, monitoring, probe-effect, testing, distributed real-time systems, replay.

1 Introduction

Testing is the process of revealing failures by exploring the runtime behavior of the system for violations of the specifications. Debugging on the other hand is concerned with revealing the errors that cause the failures. The execution of an error infects the state of the system, e.g., by infecting variables, memory, etc, and finally the infected state propagates to outputs. The process of debugging is thus to follow the trace of the failure back to the error. In order to reveal the error it is imperative that we can reproduce the failure repeatedly. This requires knowledge of the start conditions and a deterministic execution. For sequential software with no real-time requirements it is sufficient to apply the same input and the same internal state in order to reproduce a failure. For distributed real-time software the situation gets more complicated due to timing and ordering issues.

There are several problems to be solved in moving from debugging of sequential programs (as handled by standard commercial debuggers) to debugging of distributed real-time programs. We will briefly discuss the main issues by making the transition in three steps:

Debugging sequential real-time programs

In moving from debugging sequential non real-time programs to debugging sequential real-time programs temporal constraints on interactions with the external process have to be met. This means that classical debugging with breakpoints and single-stepping cannot be directly applied since it would make timely reproduction of outputs to the external process impossible. Likewise, using a debugger we cannot directly reproduce inputs to the system that depend on the time when the program is executed, e.g., readings of sensors and the local real-time clock. A mechanism, which during debugging faithfully and deterministically reproduces these interactions, is required.

Debugging multi-tasking real-time programs

In moving from debugging sequential real-time programs to debugging multitasking real-time programs executing on a single processor we must in addition have mechanisms for reproducing task interleavings. We need for example, to keep track of preemptions, interrupts, and accesses to critical regions. That is, we must have mechanisms for reproducing the interactions and synchronizations between the executing tasks.

Reproducing rendezvous between tasks have been covered by Tai et al. [14], as have reproduction of interrupts and task-switches using special hardware, Tsai et al. [17]. Reproducing interrupts and task switches using both special hardware and software has been covered by Dodd et al. [1]. However, since both the two latter approaches are relying on special hardware and profiling tools they

are not very useful in practice. They also lack support for debugging of distributed real-time systems, even though Dodd et al. claim they do.

Debugging of distributed real-time systems

The transition from debugging single node real-time systems to debugging distributed real-time programs introduces the additional problems of correlating observations on different nodes and break-pointing tasks on different nodes at exactly the same time.

To implement distributed breakpointing we either need to send *stop* or *continue* messages from one node to a set of other nodes with the problem of nonzero communication latencies, or we need *á priori* agreed upon times when the executions should be halted or resumed. The latter is complicated by the lack of perfectly synchronized clocks, meaning that we cannot ensure that tasks halt or resume their execution at exactly the same time. Consequently, a different approach is needed.

Debugging by deterministic replay

We will in this paper present a software based debugging technique based on deterministic replay [8][9], which is a technique that records significant events at run-time and then uses the recording off-line to reproduce and examine the system behavior. The examinations can be of finer detail than the events recorded. For example, by recording the actual inputs to tasks we can off-line re-execute the tasks using a debugger and examine the internal behavior to a finer degree of detail than recorded.

Deterministic replay is useful for tracking down errors that have caused a detected failure, but is not appropriate for speculative explorations of program behaviors, since only recorded executions can be replayed.

We have adopted deterministic replay to single tasking, multi-tasking, and distributed real-time systems. By recording all synchronization, scheduling and communication events, including interactions with the external process, we can off-line examine the actual real-time behavior without having to run the system in real-time, and without using intrusive observations, potentially leading to probe-effects [3]. Probe-effects occur when the relative timing in the system is perturbed by observations, e.g., by breakpoints put there solely for facilitating observations. We can thus deterministically replay the task executions, the task switches, interrupt interference and the system behavior repeatedly. This also scales to distributed real-time systems with globally synchronized time bases. If we record all interactions between the nodes we can locally on each node deterministically reproduce them and globally correlate them with corresponding events recorded on other nodes.

Contribution

The contribution of this paper is a method for debugging real-time systems, which to our knowledge is

- *The first entirely software based method for deterministic debugging of single tasking and multi-tasking real-time systems.*
- *The first method for deterministic debugging of distributed real-time systems.*

Paper outline: *Section 2* presents our system model and *Section 3* our method for real-time systems debugging. *Section 4* provides a small example to illustrate the method. *Section 5* discusses some general issues related to deterministic replay, and *Section 6* gives an overview of related work. Finally, in *Section 7*, we conclude and give some hints on future work.

2 The System Model

We assume a distributed system consisting of a set of nodes. Each node is a self sufficient computing element with CPU, memory, network access, a local clock and I/O units for sampling and actuation of an external process. We further assume the existence of a global synchronized time base [2][4] with a known precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

The software that runs on the distributed system consists of a set of concurrent tasks and interrupt routines, communicating by message passing or via shared memory. Tasks and interrupts may have functional and temporal side effects due to preemption, message passing and shared memory.

We assume a run-time real-time kernel that supports preemptive scheduling, and require that the kernel has a recording mechanism such that significant system events like task starts, preemptions, resumptions, terminations and access to the real-time clock can be recorded, as illustrated in Figure 1. The detail of the monitoring should penetrate to such a level that the exact occurrence of

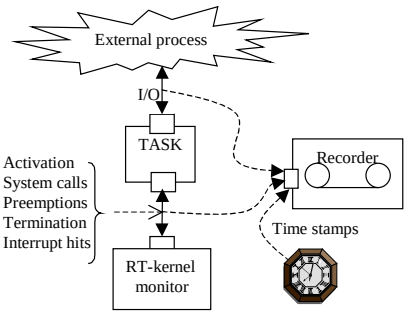


Figure 1 Kernel with monitoring and recording.

preemptions and interrupt interference can be determined, i.e., it should record the program counter values where the events occurred. All events should also be time-stamped according to the local real-time clock.

We further require that the recording mechanism governed by the run-time kernel supports programmer defined recordings. That is, there should be system calls for recording I/O operations, local state, access to the real-time clock, received messages, and access to shared data.

All these monitoring mechanisms, whether they reside in the real-time kernel or inside the tasks, will give rise to probe-effects [3][9] if they are removed from the target system. That is, removing the monitoring probes will affect the execution, thereby potentially leading to new and untested behaviors. The probes should therefore remain in the target system. It is consequently essential to consider monitoring early in the design process and allocate resources for it.

We further assume that we have an off-line version of the real-time kernel (shown in Figure 2), where the real-time clock and the scheduling have been disabled. The off-line kernel with support by a regular debugger is capable of replaying all significant system events recorded. This includes starting tasks in the order recorded, and making task-switches and repeating interrupt interference at the recorded program counter values. The replay scheme also reproduces accesses to the local clock, writing and reading of I/O, communications and accesses to shared data by providing recorded values.

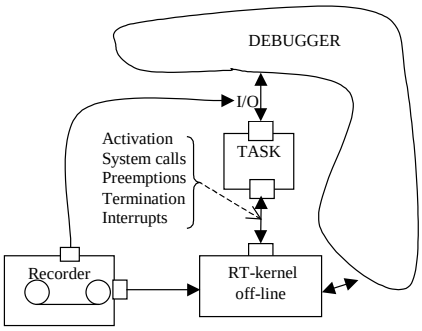


Figure 2 Offline kernel with debugger

3 Real-time systems debugging

We will now in further detail discuss and describe our method for achieving deterministic replay. We follow the structure in the introduction and start by giving our solution to handling sequential software with real-time constraints, and then continue with multitasking real-time systems, and distributed multitasking real-time systems.

3.1 Debugging single task real-time systems

Debugging of sequential software with real-time constraints, requires that the debugging is performed such that the temporal requirements imposed by the environment are still fulfilled. This means, as pointed out in the introduction, that classical debugging with breakpoints and single-stepping cannot be directly applied, since it would invalidate timely reproduction of inputs and outputs.

However, if we identify and record significant events in the execution of the sequential program, such as reading values from an external process, accesses to the local clock, and outputs to external processes, we can order them. By ordering all events according to the local clock, and recording the content of the events (e.g., the values read) together with the time when they occurred we can off-line reproduce them in a timely fashion. That is, during debugging we "short-circuit" all events corresponding to the ones recorded by substituting readings of actual values with recorded values.

An alternative to our approach is to use a simulator of the external process, and synchronize the time of the simulator with the debugged system. However, simulation is not required if we already have identified the outputs that caused the failure.

3.2 Debugging multitasking real-time systems

To debug multitasking real-time systems we need, in addition to what is recorded for single task real-time systems, to record task interleavings. That is, we should record the transfers of control. To identify the time and location of the transfers we must for each transferring event assign a time stamp, and record the program counter (PC).

To reproduce the run-time behavior during debugging we replace all inputs and outputs with recorded values, and instrument all tasks by inserting trap calls at the PC values where control transfers have been recorded. These trap calls then execute the off-line kernel, which has all the functionality of a real-time kernel, but all transfer of control, all accesses to critical regions, all releases of higher priority tasks, and all preemptions by interrupts are dictated by the recording. Inter-process communication is however handled as in the run-time kernel, since it can be deterministically reproduced by reexecuting the program code.

Figure 3 depicts a schedule with the three tasks *A*, *B*, and *C*. We can see that task *A* is being preempted by task *B* and *C*. During debugging this scenario can be reproduced by instrumenting task *A* with calls to the kernel at *A*'s PC=*x* and PC=*y*.

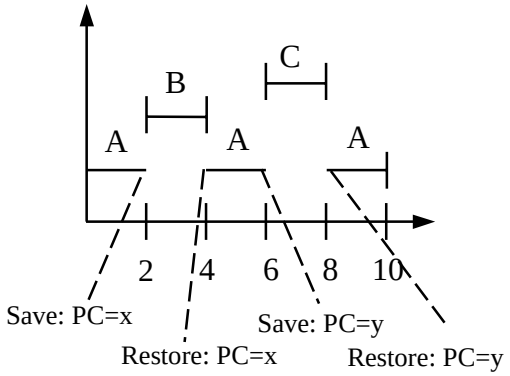


Figure 3 Task A is preempted twice by task B and C.

The above reasoning is a bit simplistic when we have program control structures like loops and recursive calls, since in such structures the same PC value will be executed several times, and hence the PC value does not define a unique program state. This can be alleviated if the target processor supports instruction or cycle counters. The PC will together with any of these counters define a unique state. However, since these hardware features are not very common in commercial embedded micro-controllers, we will use the following alternative approach:

Instead of just saving the PC, we will save all information stored by the kernel in context-switches, including CPU registers (address and data), as well as stack and program counter registers pertaining to the task that is preempted. The entire saved context can be used as a unique marker for the preempted program. The program counter and the contents of the stack register would for example be sufficient for differentiating between recursive calls.

For loops, this approach is not guaranteed to uniquely identify states, since (at least theoretically) a loop may leave the registers unchanged. However, for most realistic programs the context together with the PC will define a unique state. Anyhow, in the unlikely situation, during replay, of having the wrong iteration of a loop preempted due to indistinguishable contexts, the functional behavior of the replay will be different from the one recorded – and therefore detectable; or if the behaviors are identical, then it is of no consequence.

Any multitasking kernel must save the contexts of suspended tasks in order to resume them, and in the process of making the recording for replay we must store contexts an additional time. To eliminate this overhead we can make the kernel store and retrieve all contexts for suspended tasks from the recording instead, i.e., we need only store the contexts once.

Our approach eliminates the need for special hardware instruction counters since it requires no extra support other than a recording mechanism in the real-time kernel. If we nonetheless have a target processor with instruction counters or cycle counters, we can easily include these counters into the recorded contexts, and thus guarantee unique states.

To enable replay of the recorded event history we insert trap calls to the off-line kernel at all recorded PC values. During replay we consequently get plenty of calls to the kernel for recorded PC values that are within loops, but the kernel will not take any action for contexts that are different from the recorded one.

An alternative approach to keep track of loop executions is to make use of software instruction counters [10] that count backward branches and subroutine calls in the assembly code. However, this technique requires special target specific tools that scan through the assembly code and instrument all backward branches. The approach also affects the performance, since it usually dedicates one or more CPU registers to the instruction counter, and therefore reduces the possibility of compiler optimizations.

3.3 Debugging distributed real-time systems

We will now show how local recordings can be used in achieving deterministic distributed replay. The basic idea is to correlate the local time stamps up to the precision of the clock synchronization. This will allow us to correlate the recordings on the different nodes. As we by design can record significant events like I/O sampling and inter-process communication, we can on each node record the contents and arrival time of messages from other nodes. The recording of the messages therefore makes it possible to locally replay, one node at a time, the exchange with other nodes in the system without having to replay the entire system concurrently. Time stamps of all events make it possible to debug the entire distributed real-time system, and enables visualizations of all recorded and recalculated events in the system. Alternatively, to reduce the amount of information recorded we can off-line re-execute the communication between the nodes. However, this requires that we order-wise synchronize all communication between the nodes, meaning that a fast node waits up until the slow node(s) catch up. This can be done truly concurrently using several nodes, or emulated on a single node for a set of homogenous nodes.

Global states

In order to correlate observations in the system we need to know their orderings, i.e., determine which observations are concurrent, and which precede and succeed a particular event. In single node systems or tightly coupled

multiprocessor systems with a common clock this is not a problem, but for distributed systems where there is no common clock this is a significant problem. An ordering on each node can be established using the local clocks, but how can observations between nodes be correlated?

One approach is to establish a causal ordering between observed events, using for example logical clocks [7] derived from the messages passed between the nodes. However, this is not a viable solution if tasks on different nodes work on a common external process, without exchanging messages, or when the duration between observed events is of significance. In such cases we need to establish a total ordering of the observed events in the system. This can be achieved by forming a synchronized global time base [2][4]. That is, we keep all local clocks synchronized to a specified precision δ, meaning that no two nodes in the system have local clocks differing by more than δ.

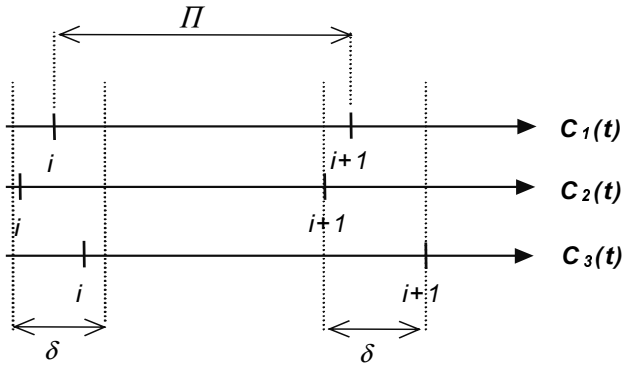


Figure 4 The occurrence of local ticks on three nodes

Figure 4 illustrates the local ticks in a distributed system with three nodes, all with tick rate Π, and synchronized to the precision δ. There is no point in having $\Pi \leq \delta$, because the precision δ dictates the margin of error of clock readings, and thus a $\Pi \leq \delta$ would result in overlaps of the δ intervals during which the synchronized local ticks may occur [6].

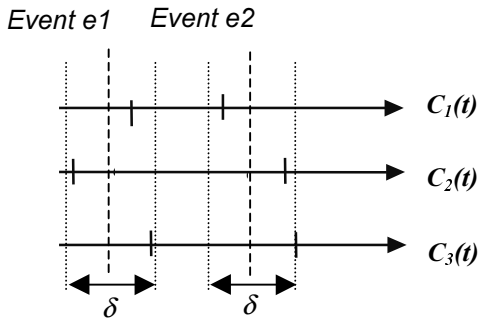


Figure 5. The effects of a sparse time base.

Consider Figure 5, illustrating two external events that all three nodes can observe, and which they all timestamp. Due to the sparse time base [5] and the precision δ, we end up with timestamps of the same event that differ by 1 time unit (i.e., Π) while still complying with the precision of the global time base. This means that some nodes will consider events to be concurrent (i.e., having identical time stamps), while other nodes will assign distinct time stamps to the same events. This is illustrated in Figure 5, where node 2 will give the events e1 and e2 identical time stamps, while they will have difference 2 and 1 on nodes 1 and 3, respectively. That is, only events separated by more than 2Π can be globally ordered.

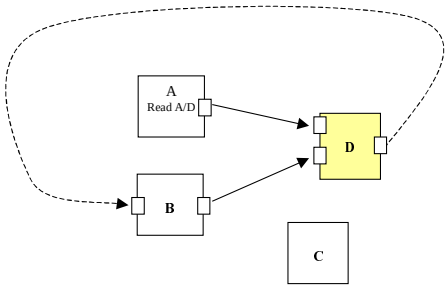


Figure 6 The data-flow between the tasks

4 A small example

We are now going to give an example of how the entire recording and replay procedure can be performed. The considered system has four tasks *A*, *B*, *C*, and *D* (Figure 6). The tasks *A*, *B*, and *D* are functionally related and exchange information. Task *A* samples an external process via an analog to digital converter (A/D), task *B* performs some calculation based on previous messages from task *D*, and task *D* receives both the processed A/D value and a message from *B*; subsequently *D* sends a new message to *B*.

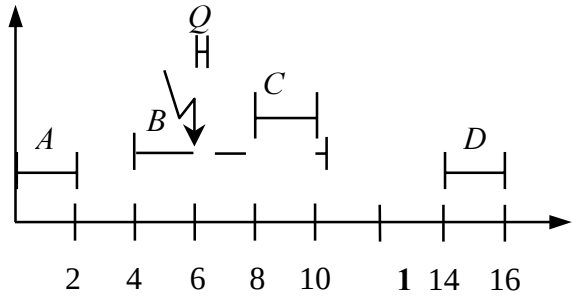


Figure 7 The recorded execution order scenario

Task *C* has no functional relation to the other tasks, but preempts *B* at certain rare occasions, e.g., when *B* is subject to interrupt interference, as depicted in Figure 7.

However, task *C* and *B* both uses a function that by a programming mistake is made non re-entrant. This function causes a failure in *B*, which subsequently sends an erroneous message to *D*, which in turn actuates an erroneous command to an external process, which fails. The interrupt *Q* hits *B*, and postpones *B*'s completion time. *Q* causes in this case *B* to be preempted by *C* and therefore becomes infected by the erroneous non-reentrant function. This rare scenario causes the failure. Now, assume that we have detected this failure and want to track down the error.

We have the following control transfer recording for time 0 -16:

1. Task *A* starts at time 0
2. Task *A* stops at time 2
3. Task *B* starts at time 4
4. Interrupt *Q* starts at time 6, and preempts task *B* at PC=*x*
5. Interrupt *Q* stops at time 6,5
6. Task *B* resumes at time 6,5, at PC=*x*
7. Task *C* starts at time 8, and preempts task *B* at PC=*y*
8. Task *C* stops at time 10
9. Task *B* resumes at time 10, at PC=*y*
10. Task *B* stops at time 10,3
11. Task *D* starts at time 14
12. Task *D* stops at time 16

Together with the following data recording:

1. Task *A* at time 1, read_ad() = 234
2. Task *B* at time 4, message from *D* = 78

During debugging all tasks are instrumented with calls to the off-line kernel at their termination, and preempted tasks *B* and *C* are instrumented with calls to the off-line kernel at their recorded PC values. Task *A*'s access to the read_ad() function is short circuited and feed with the recorded value instead. Task *B* gets at its start a message from *D*, which is recorded before time 0.

The message transfers from *A* and *B* to *C* is performed by the off-line kernel in the same way as the on-line kernel.

The programmer/analyst can breakpoint, single step and inspect the control and data flow of the tasks as he or she see fit in pursuit of finding the error. Since the replay mechanism reproduces all significant events pertaining to the real-time behavior of the system the debugging will not cause any probe-effects.

As can be gathered from the example it is fairly straightforward to replay a recorded execution. The error can be tracked down because we can reproduce the exact interleavings of the tasks and interrupts repeatedly. Experience has shown that reproducing failures of the exemplified kind is very difficult in practice. A deterministic replay mechanism is thus an invaluable tool.

5 Discussion

Schütz [12] has made three claims about deterministic replay in general, which we briefly comment below:

Claim 1: *One can only replay what has previously been observed, and no guarantees that every significant system behavior will be observed accurately can be provided. Since replay takes place at the machine code level the amount of information required is usually large. All inputs and intermediate events, e.g. messages, must be kept.*

The amount and the necessary information required is of course a design issue, and it is not true that all inputs and intermediate messages must be recorded. The replay can as we have shown actually re-execute the tasks in the recorded event history. Only those inputs and messages which are not re-calculated, or re-sent, during the replay must be kept. This is specifically the case for RTS with periodic tasks, where we can make use of the knowledge of the schedule (precedence relations) and the duration before the schedule repeats it self (the LCM – the Least Common Multiple of the task period times.) In systems where deterministic replay has previously been employed, e.g., distributed systems [11] and concurrent programming (ADA) [14] this has not been the case. The restrictions, and predictability, inherent to scheduled RTS do therefore give us the great advantage of only recording the data that is not recalculated during replay.

Claim 2: *If a program has been modified (e.g., corrected) there are no guarantees that the old event history is still valid.*

If a program has been modified, the relative timing between racing tasks can change and thus the recorded history will not be valid. The timing differences can stem from a changed data flow, or that the actual execution time of the modified task has changed. In such cases it is likely that a new recording must be made. However, the probability of actually recording the sequence of events that pertain to the modification may be very low. As explained earlier, debugging in general and deterministic replay especially is not suited for speculative

investigations of the system behavior. This is an issue for regression testing, as explained in [15][16].

Claim 3: *The recording can only be replayed on the same hardware as the recording was made on.*

The event history can only be replayed on the target hardware. This is true to some extent, but should not be a problem if remote debugging is used. The replay could also be performed on the host computer if we have a hardware simulator, which could run the native instruction set of the target CPU. Another possibility would be to identify the actual high-level language statements where task switches or interrupts occurred, rather than try to replay the exact machine code instructions, which of course is machine dependent. In the latter case however, we run into the problem of defining a unique state when differentiating between iterations in loops.

6 Related work

There are a few descriptions of deterministic replay mechanisms (related to real-time systems) in the literature:

- A deterministic replay method for concurrent Ada programs is presented by Tai et al [14]. They log the synchronization sequence (rendezvous) for a concurrent program *P* with input *X*. The source code is then modified to facilitate replay; forcing certain rendezvous so that *P* follows the same synchronization sequence for *X*. This approach can reproduce the synchronization orderings for concurrent Ada programs, but not the duration between significant events, because the enforcement (changing the code) of specific synchronization sequences introduces gross temporal probe-effects. The replay scheme is thus not suited for real-time systems, neither are issues like unwanted side-effects caused by preempting tasks considered. The granularity of the enforced rendezvous does not allow preemptions, or interrupts for that matter, to be replayed. It is unclear how the method can be extended to handle interrupts, and how it can be used in a distributed environment.

- Tsai et al present a hardware monitoring and replay mechanism for real-time uniprocessors [17]. Their approach can replay significant events with respect to order, access to time, and asynchronous interrupts. The motivation for the hardware monitoring mechanism is to minimize the probe-effect, and thus make it suitable for real-time systems. Although it does minimizes the probe-effect, its overhead is not predictable, because their dual monitoring processing unit causes unpredictable interference on the target system by generating an interrupt for every event monitored [1]. They also record excessive details of the target processors execution, e.g., a 6 byte immediate AND instruction on a Motorola 68000 processor generates 265 bytes of recorded data. Their approach can reproduce asynchronous interrupts only if the target CPU has a dedicated hardware instruction counter. The used hardware approach is inherently target specific, and hard to adapt to other systems. The system is designed for single processor systems and has no support for distributed real-time systems.

- The software-based approach *HMON* [1] is designed for the HARTS distributed (real-time) system multiprocessor architecture [13]. A general-purpose processor is dedicated to monitoring on each multiprocessor. The monitor can observe the target processors via shared memory. The target systems software is instrumented with monitoring routines, by means of modifying system service calls, interrupt service routines, and making use of a feature in the pSOS real-time kernel for monitoring task-switches. Shared variable references can also be monitored, as can programmer defined application specific events. The recorded events can then be replayed off-line in a debugger. In contrast to the hardware supported instruction counter as used by Tsai et al., they make use of a software based instructions counter, as introduced by [10]. In conjunction with the program counter, the software instruction counter can be used to reproduce interrupt interferences on the tasks. The paper does not elaborate on this issue. Using the recorded event history, off-line debugging can be performed while still having interrupts and task switches occurring at the same machine code instruction as during run-time. Interrupt occurrences are guaranteed off-line by inserting trap instructions at the recorded program counter value. The paper lacks information on how they achieve a consistent global state, i.e., how the recorded events on different nodes can consistently be related to each other. As they claim that their approach is suitable for distributed real-time systems, the lack of a discussion concerning global time, clock synchronization, and the ordering of events, diminish an otherwise interesting approach. Their basic assumption about having a distributed system consisting of multiprocessor nodes makes their *software* approach less general. In fact, it makes it a hardware approach, because their target architecture is a shared memory multiprocessor, and their basic assumptions of non-interference are based on this shared memory and thus not applicable to distributed uniprocessors.

7 Conclusions

We have presented a method for deterministic debugging of distributed real-time systems. The method relies on an instrumented kernel to on-line record the occurrences and timings of major system events. The recording can then, using a special debug kernel, be replayed off-line to faithfully reproduce the functional and temporal behavior of the recorded execution, while allowing standard debugging using break points etc. to be applied.

The cost for this dramatically increased debugging capability is the overhead induced by the kernel instrumentation and by instrumentation of the application code. To eliminate probe-effects, these instrumentations should remain in the deployed system. We are however convinced that this is a justifiable penalty for many applications.

We are currently implementing an experimental real-time kernel for evaluation of the presented debugging method, but also investigate modifications of existing real-time kernels to support deterministic replay.

8 References

[1] Dodd P. S., Ravishankar C. V. *Monitoring and debugging distributed real-time programs*. Software-practice and experience. Vol. 22(10): 863-877, October 1992.

[2] Eriksson C., Thane H. and Gustafsson M. *A Communication Protocol for Hard and Soft Real-Time Systems.* In the proceedings of the 8th Euromicro Real-Time Workshop, L'Aquila Italy, June, 1996.

[3] Gait J. *A Probe Effect in Concurrent Programs*. Software – Practice and Experience, 16(3):225-233, March, 1986.

[4] Kopetz H and Ochsenreiter W. *Clock Synchronisation in Distributed Real-Time Systems*. IEEE Transactions on Computers, August 1987.

[5] Kopetz H. *Sparse time versus dense time in distributed real-time systems*. In the proceedings of the 12th International Conference on Distributed Computing Systems, pp. 460-467, 1992.

[6] Kopetz, H. and Kim, K. *Real-time temporal uncertainties in interactions among real-time objects.* Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems, Huntsville, AL, 1990.

[7] Lamport L. *Time, clock, and the ordering of events in a distributed systems*. Communications of ACM, (21):558-565: July 1978.

[8] LeBlanc T. J. and Mellor-Crummey J. M. *Debugging parallel programs with instant replay*. IEEE Transactions on Computers,36(4):471-482, April 1987.

[9] McDowell C.E. and Hembold D.P. *Debugging concurrent programs.* ACM Computing Surveys, 21(4):593-622, December 1989.

[10] Mellor-Crummey J. M. and LeBlanc T. J. *A software instruction counter*. In Proc. of 3d International Conference on Architectural Support for Programming Languages and Operating Systerns, Boston, pp. 78-86, April 1989.

[11] Netzer R.H.B. and Xu Y. *Replaying Distributed Programs Without Message Logging.* In proc. 6th IEEE Int. Symposium on High Performance Distributed Computing. Pp. 137-147. August 1997.

[12] Schütz W. *Fundamental Issues in Testing Distributed Real-Time Systems*. Real-Time Systems journal, Kluwer A.P., vol. 7(2):129-157, 1994

[13] Shin K. G. *HARTS: A distributed real-time architecture.* IEEE Computer, 24(5):25-35, May, 1991.

[14] Tai K.C, Carver R.H., and Obaid E.E. *Debugging concurrent Ada programs by deterministic execution.* IEEE Transactions on Software Engineering. Vol. 17(1):45-63, January 1991.

[15] Thane H. and Hansson H. *Handling Interrupts in Testing of Distributed Real-Time Systems*. In proc. Real-Time Computing Systems and Applications conference (RTCSA'99), Hong Kong, December, 1999.

[16] Thane H. and Hansson H. *Towards Systematic Testing of Distributed Real-Time Systems*. Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99), Phoenix, Arizona, December 1999.

[17] Tsai J.P., Fang K.-Y., Chen H.-Y., and Bi Y.-D. *A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging.* IEEE Transactions on Software Engineering, 16: 897 - 916, 1990.

A Modelling Framework to Support the Design and Analysis of Distributed Real-Time Control Systems

Martin Törngren and Ola Redell

Mechatronics Lab, Department of Machine Design, Royal Institute of Technology
SE-100 44 Stockholm, Sweden, email: {martin, ola}@damek.kth.se

ABSTRACT: Within the AIDA project a modelling framework has been developed to support design issues related to the implementation of control applications in embedded distributed computer systems. At a relatively high level of abstraction the models describe the structure and timing behaviour of a control application (in terms of functions and operational modes) and its implementation (hardware, operating system threads and resources). The resource description allows the timing behaviour of the implementation to be analysed and fed back into the application models. The models form the basis for a decentralization tool-set, where a first prototype is under development. Examples of the models are given and the framework is compared to related modelling approaches.

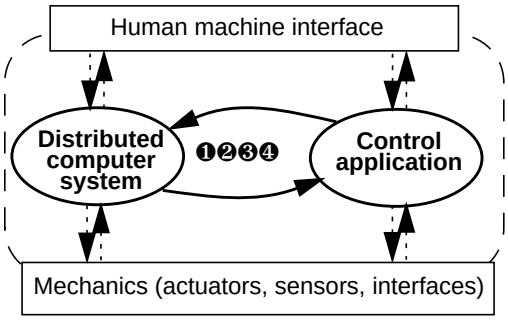
1. INTRODUCTION

Due to the dramatic evolution of electronics, machine industry faces a shift in machine design where more and more functionality is included in the form of software on embedded distributed computer systems. In the first phase of this transition, e.g. in the automotive industry, the new software based solutions were in principle stand-alone systems, [23], in terms of both the computer system and the used set of sensors and actuators. To really benefit from the potential of software based systems, the next phase in this development is that different, previously stand-alone functions, will be interacting and cooperating so as to optimize the overall performance of a system such as in the next generation of cars. Essential, in both an engineering and research context, is the resulting need for interactions between control, mechanical, and software/computer/electronics engineers, [33]. This, in addition, necessitates that new approaches and facilities are provided for system modelling, analysis and tool support.

It often pays off to introduce an embedded distributed computer control system in new machinery primarily because of the reduction in cabling, hydraulic hoses, mechanical linkage etc., but also due to the flexibility inherent in digital solutions allowing e.g. diagnostics and improved control algorithms, [28]. In a machine embedded distributed control system, electronic hardware (microprocessors, ASICs, etc.) and control software is distributed in the machine and connected to sensors and actuators. The topology of the distributed computer system is usually to a large extent given by the structure of the machine with its sensors and actuators.

In the development of such systems, however, designers are faced with several ‘new’ problems. For example, the design of the mechanics can no longer be isolated from the design of electronic control nodes because the nodes need to be embedded into the machine together with their information and energy cabling.

The design issues depicted in Figure 1 form the primary background for this work.



❶ *Structuring, partitioning* and *allocation* deals with the definition of components, their connections and to the mappings between different structures (e.g. functions/threads/processors).

❷ *Execution and communication policies*. The definition of policies and mechanisms for triggering, scheduling, synchronisation and communication.

❸ *Performance requirements.* The definition of timing requirements and related trade-offs.

❹ *Error detection and handling policies.* In particular additional failure modes of distributed real-time systems must be identified and handled.

Figure 1. Essential design issues for distributed real-time control systems

The horizontal arrows in Figure 1 indicate different design cases and disciplinary interactions. For example the performance requirements of the control application could be used to dictate the choice of the execution policy in the computer system. On the other hand, these policies may be given, thus constraining the solution space and potentially requiring appropriate counter measures to be taken in control design (e.g. including compensation for time-varying delays) or even the requirements to be reformulated.

The design issues of Figure 1 have the following overall characteristics

- The determination of a suitable degree of decentralization of the control system on the distributed computer system depends on a number of sometimes conflicting factors including control performance, reliability, flexibility and cost, forming a multi-objective design problem.
- The issues should be addressed in early design stages since they affect essential architectural properties of the integrated computer control system. For example, the degree of decentralization of the control system is highly related to the dimensioning of the computational and communicational resources of the distributed computer system. Neglecting early performance analysis may lead to costly clashes in system integration. The adoption of early performance analysis, on the other hand, enables the design of cost-effective solutions.

1.1. The AIDA project and the structure of the paper

Considering newer applications with increasing complexity, and with high reliability and safety requirements, there is an increasing need to provide models and tools to support the designers of machine embedded control systems. Although several useful modelling approaches and computer aided tools exist for the design of control, software and mechanical systems, such models and tools provide very little or no support when the applications are implemented in embedded distributed real-time computer systems.

The project *Automatic control in distributed applications* (AIDA) was therefore started with the aim of developing a supporting modelling framework and a tool-set targeting analysis and synthesis related to the design issues of Figure 1, see [2] for more background information on the project. The current status of the tool-set development is that a modelling framework has been developed on which the tool-set will be based. The modelling framework has been used in a few case studies, [28][22]. Based on the existing framework the implementation of a tool prototype has just began.

The AIDA approach is far from a complete development method with supporting models and tools. Instead, the whole concept is devoted towards solving a few concrete design problems that occur when a control application is to be implemented in a distributed computer system. Thus, the AIDA models and related design guidelines, [27][28], can be used to complement existing methodologies when appropriate. Issues such as timing analysis, the provision of graphical views of system structure and timing behaviour should be useful not only in early conceptual design stages but also in work related to the augmentation of already existing systems. The envisioned tool-set could be seen as a real-time design complement to existing control engineering tools.

The interactions between the different design choices of Figure 1 are considerable, and have a great impact on the final system performance. Therefore the AIDA tool-set is intended to support iterative engineering, answering a number of “what if” questions put by the designer. Typical analysis results that the AIDA tool-set should be able to provide are estimates of the timing behaviour of the control system given a particular implementation (allocation, processors, scheduling policies etc.). Example results include estimates of feedback delays, delay variations, jitter in sampling periods, and the utilization of processor and communication links.

Such results can be used to answer a number of different design questions. For example; With fixed hardware and application structures, what performance can be achieved by (i) changing the hardware components, (ii) changing policies for scheduling, clock synchronisation etc. Apart from analysis, synthesis functionality such as (semi) automatic allocation and control system restructuring are also planned to be included.

The purpose of this paper is to give an overview of the results so far, primarily dealing with the modelling framework. Section 1.2. gives a brief overview of related research projects with respect to tool support. Section 2 describes the developed modelling framework including examples of the usage of the models. Section 3 discusses other existing modelling approaches and their relation to the AIDA models. Finally section 4 provides conclusions and discusses further work in the project.

1.2. Related research efforts on tools

State of the art computer aided control and software engineering (CACE and CASE respectively) tools do sup-

port rapid prototyping for specific distributed computer systems. The tool-sets provided by the companies Mathworks and dSPACE enable control system design, analysis and simulation, code generation from the graphical block diagrams of Simulink, “transparent” implementation of generated code on DSPs, and real-time operator interaction and logging, [14]. Transparency here refers to the fact that the user manually specifies allocation of Simulink blocks to processors, the physical configuration of which is also described in Simulink. The required communication code is then included transparently. The resulting code is managed by a real-time executive.

While such tools are extremely useful for control designers, there is still a large amount of functionality missing with respect to implementation analysis. This holds for both related control and software engineering tools which do not allow exploration of implementation alternatives, for example with respect to resource structures and execution strategies. Nor do they provide any guarantee that an implementation will work with respect to real-time behaviour.

Out of a number of related research tools we have found the three following to be highly interesting:

- The Development Framework, [4], incorporates some of the functions and a tool structure similar to that we envision. The control design is specified in Simulink and then transferred to a commercial CASE tool (Software through Pictures) which models can be analysed and transformed by other tools (clustering, replication). The combination of CACE and CASE technology facilitates multi disciplinary project development and makes use of the strong points of the respective modelling and analysis features. Hardware modelling, execution and communication strategy considerations appear to be weak points in the Development Framework.
- The GRAPE tool-set, [18], is developed for digital signal processing systems and has an interesting tool structure, including tools for allocation, scheduling and restructuring of the computer hardware. Only point to point architectures are considered.
- An interesting effort is the Parallel Scalable Design Tool-set (PSDT) from Honeywell research. PSDT is a CASE tool specifically addressing implementation in distributed and parallel real-time systems, [5].

In the real-time research community, a number of prototypical tools have been developed for schedule simulation, timing analysis and schedule generation, [10][31][15][3]. The amount of theoretical research work is vast including work on general purpose parallel processing, parallel and distributed real-time systems, and related work in control engineering. One important shortcoming of the supporting models is that they, with a few rare exceptions, do not directly address the timing constraints of sampled data systems, [22][28].

2. THE MODELLING FRAMEWORK

2.1. Requirements on the modelling framework

The overall purpose of the models is to support the design of distributed real-time control systems by providing means for describing system structure and timing behaviour, and by including information to support analysis and synthesis. The ambition has been, and is still, to fill in the gaps in terms of missing models and tools. Commercial tools and research provide good support for analysis and simulation of continuous-time and discrete-event dynamic systems. Lacking support for behaviour modelling with respect to timing behaviour (e.g. precedence, parallelism, triggers, durations etc.) and high-level modelling of resource management is therefore the primary focus.

The basis for developing the modelling framework was to determine the information needed in the context of the early stages of design of distributed control systems. A major challenge in developing the framework concerns the provision of adequate and different abstraction levels taking into account the analysis needs of industrial designers and the modelling precision needed. Consider for example the implementation of a data-flow over two processors and a serial network. A high level of abstraction is to simply model this network as a time-delay with potential data-loss, thus providing an abstraction suitable for control engineers. Increasing detail would take into account essential resource management (e.g. processor and communication scheduling) such that the timing behaviour of system functions can be analysed sufficiently well, providing a suitable high level of abstraction for software and computer engineers.

The following requirements on the models were initially formulated:

- The models must support the multidisciplinary characteristics of the design problems where at least both computer (software and hardware) and control engineering is involved. To do this the modelling framework must provide suitable abstractions, views and links between these.

- The model abstractions should be on a relatively high level to support ‘architectural decisions’ that directly affect the system structure and timing behaviour, and thereby a number of essential system properties such as control system performance (as a function of the system timing), extensibility, testability etc.
- The aim is to give support for the description of realistic control systems. This implies the need to be able to model time- and event-triggered, multirate, control systems with different modes of operation. These control systems are to be implemented on distributed heterogeneous hardware with both serial and parallel communication links interconnecting the processing elements. Furthermore the processing elements can be placed on different locations in the mechanical structure, hence the sites of sensors and actuators should be modelled as well as the assignment of processing elements to such sites. There is also a need for modelling the various system specific overheads, scheduling policies, error handling etc.
- Since design could be based also on existing systems or components, the models must allow description of earlier implementations with many already taken and hence fixed design decisions.

2.2. The structure of the modelling framework

The AIDA modelling framework is outlined in Figure 2. The models are divided into three separate parts, *application*, *computer system* and *mechanics*. The application models describe the functional structure, data and control flows as well as timing requirements of the motion control system. Furthermore, the functions and the inter function communications that constitute the smallest building blocks of the application models are described in detail in terms of resource requirements. In the computer system models, the threads (no distinction is made here between threads and processes) and their interconnections resulting from partitioning, are described together with the computer hardware. Operating system behaviour, scheduling policies etc. are also defined. The mechanical model describes the sensors, actuators, drive units and other elements that form the interface between the computer system and the controlled process. One important attribute of the mechanical models is the physical locations of such interfaces. Currently there is no timing behaviour description implemented for the mechanical parts, instead time-constants and delays are given as attributes at the component level. No continuous and discrete-time behaviour is described for the mechanical system since this would include modelling of mechanical dynamics; for this there already exists good models and tools. The AIDA modelling framework is more thoroughly described in [22].

	Application models	**Computer system models: Software and Hardware**	**Mechanical models**
Overall behaviour	*Modes of Operation*		
Structure	Functional block diagrams	Computer hardware structure Process structure diagram	Location model Interface components placement
Timing Behaviour Specification & Implementation	Timing and Triggering Diagram (TTD) Implementation TTD	Inter Process Communications IPC model, Communication resources, Communication resource chains Processes: Process TTD and Process internal TTD Computer system: Operating system, Scheduling policy, Synchronisation	*See comment in section 2.2.*
Component Descriptions	Detailed function description Detailed data flow description	HW characteristics: Processing elements, Communication links, Clocks SW characteristics: Implemented Processes	Interface component model Environment interfaces model

Figure 2. Overview of the structure of the AIDA modelling framework

2.3. A case study used to describe a representative subset of the AIDA models

A sub-set of the models will be briefly described in the coming sections with simple examples to give a flavour of, and an introduction to the modelling framework. The emphasis in the description is on parts of the application and computer system models. The description is based on a case study in which models of the control system of a four legged vehicle were derived. In the case study, fixed vehicle mechanics is assumed as is the design of the control application. The vehicle is currently being built at the Mechatronics lab.

In the complete case study, described in [22], two choices of hardware architectures are investigated. Based on the developed application and computer models a coarse analysis of the timing behaviour of the implementation is performed.

2.4. Application models

The application is the control design as it is originally modelled in for example Simulink. In the AIDA application models, the control design is further specified with three basic types of diagrams: The macro mode transition diagram, the functional block diagrams and timing and triggering diagrams.

The Macro Mode Transition Diagram (MMTD) describes the different high-level control modes of the system. An example is the take-off, cruising and landing of an aircraft. Each mode can be expected to use substantially different controllers that are best described separately.

The Functional Block Diagrams describe the data-flows between the different control functions within each macro mode. Note that the emphasis is on describing the functional and implementation independent structure of the control application with its functional blocks and data-flows. No execution semantics (e.g. execute a block when data is available) is associated with a block diagram. Instead, precise timing and triggering specifications are given in separate diagrams.

Figure 3 shows a part of a functional block diagram for the control system of the vehicle mentioned above. The torque control loop of one joint of the vehicle's four legs is shown. The diagram shows how functions *S*, *TC* and *A* are connected to perform a sample (*S*), compute (*TC*) and actuate (*A*) loop. The position samples include measures of two joint angles (on both sides of the flexible joint), used by *TC* to compute the joint torque.

Data-flows to functions not shown in Figure 3 are indicated by dashed lines. Those data-flows correspond to the position sample, *pos*, sent to an observer function higher in the hierarchy and the reception of a torque reference signal, *torq*, from a higher level controller.

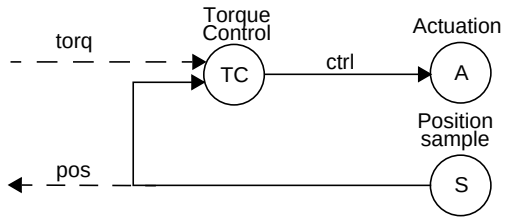


Figure 3. Functional block diagram for the torque control loop.

The behaviour of the application in terms of timing, triggering and precedence is modelled in TTDs (Timing and Triggering Diagrams). The same functions are used as building blocks as in the functional block diagrams, but the TTDs separate functions that are executed with different periods into activities. A TTD describes the control flow of the activities within each macro mode. Precedence relations, time and event triggers are shown and tolerances can be given for the control delays, jitter and synchronisation. This is useful to specify the requirements that the control design poses on the implementation.

With the example functions in Figure 3, a TTD can be specified. It is assumed that all functions in the torque control loop are specified to be executed with the same frequency (period) but that the actuate function should be triggered 0.5 *ms* after the sample function in order to achieve a constant control delay. Hence, the functions are collected into one activity, described by one single TTD. It is further assumed that there are requirements on the admissible jitter of the sample function, as well as on the variation in control delay. These requirements are shown in the TTD as specifications on the time trigger. A time trigger is defined as follows: `<Type: Source, Period:[Delay], Tolerance>`. The *Type* describes if it is a time, event or loop trigger (TT, ET or LOOP). The *source* indicates what the source of the trigger is (a clock for time triggers and an event for event triggers). *Period* gives the period of a time trigger and minimum period of an event trigger. *Delay* is an optional specification of the delay (phase) of the trigger compared to other triggers having the same source. Finally, the *tolerance* defines a tolerance of what variation is admissible from the specified triggering time.

Figure 4 shows the TTD for the torque control loop in Figure 3. The precedence relations between the functions are shown as well as the explicit triggers that both the sampler (*S*) and actuate (*A*) functions are given. It is specified that the actuation should be done 0.5 *ms* after the sampling. Both the sample and actuate functions are allowed to

deviate with at most 100 μs from their specified start times

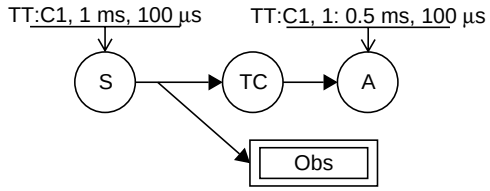


Figure 4. Timing and triggering diagram for the torque control activity.

The double rectangle defines a *rate interfacing function* which is used to describe asynchronous (non blocking) communication between different activities. The meaning is that the sample function sends data (*pos* in Figure 3) to an observer function, *Obs*, that belongs to the parts of the control system that are not modelled here. The observer function should however be part of some other TTD in a complete system modelling. In a similar manner, a rate interfacing function (tagged *TC*) is used in some other TTD (which is not given here), to describe the communication associated with the data-flow, *torq*, in Figure 3. In that case, *TC* is the receiving function.

A timing and triggering diagram can describe parallel paths as well as alternative paths, selected at run-time. A path that is not necessarily executed every time the activity is triggered is called a micro mode. Two complementary micro modes may e.g. be the use of a fast linear controller versus using a better but slower non-linear one. It should be noted that the arbitration between different micro modes is not described at a low level with e.g. if-then-else constructs. Micro modes are included only to indicate possible execution paths in an activity, not to show how the micro modes are selected.

One additional timing and triggering diagram is used to describe the timing that results from an implementation. The Implementation TTD (I-TTD) is expected to be the result of some analysis tool within the AIDA tool-set. An I-TTD is similar to its corresponding original TTD but with the difference that extra delay functions have been included to show the effects of implementation. Furthermore, the triggers are specified with their achieved values on jitter, delays and period. The added delay functions include delays originating from e.g. communications and scheduling effects such as the sequential execution of concurrent activities.

Figure 5 shows an I-TTD for the torque control activity in Figure 4. The delay functions D_1 and D_2 have been included and the values specifying the triggers have been exchanged for some estimated values obtained by analysis.

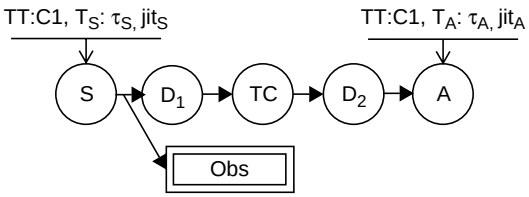


Figure 5. Implementation timing and triggering diagram for the torque control activity.

At the functional level of the application models, the functions and the data-flows are described with worst case execution times, communication requirements etc. Hence, the functions are not described in detail, only their resource needs are specified.

2.5. Computer System

The computer system models include specification of the computer hardware, the processing elements and their inter connecting communication links. Also, the operating system is described in terms of scheduling policies, dispatching time, clock tick period etc.

When the application has been partitioned into processes, the connections (communications) between the processes are shown in a process structure diagram.

Inter process communications (IPCs) are modelled with communication resources (message queues and transmitters) that specify e.g. blocking times of message queues and scheduling policies of the communication links.

The communication models can later be used for analysis of communication latencies.

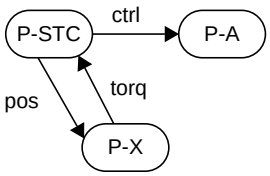


Figure 6. Process structure diagram for the three processes in the example.

Assume that the example activity shown in Figure 4 is partitioned into two processes, *P-STC* and *P-A*. The partitioning may correspond to e.g. a distributed implementation with the sample and computation functions at one processor, and the actuation at another. *P-STC* contains the sample (*S*) and computation functions (*TC*), and *P-A* contains the actuation function (*A*). Also assume that another process, *P-X*, corresponding to all other activities in the system, is defined. The corresponding process structure diagram is shown in Figure 6. Note that three IPCs are defined: *pos*, *torq* and *ctrl*. Each of the IPCs are data-flows in Figure 3 that "cross process borders". For simplicity, the IPCs are here given the same tags as their corresponding data-flows in the functional block diagram.

The timing of the processes is specified in a Process timing and triggering diagram (P-TTD) that looks like a regular TTD only that it specifies the timing and triggering of the processes and not of the functions. In Figure 7, a P-TTD is shown that defines the requirements of the process timing.

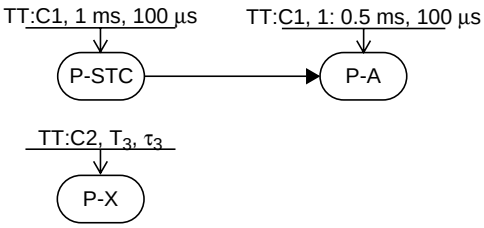


Figure 7. Process timing and triggering diagram for the three processes in the example.

The same types of triggers are used as in the regular TTD but no rate interfacing functions are included. Instead, the correspondence to rate interfacing functions, the inter process communications, are visible in the Process internal timing and triggering diagrams (Pi-TTD). A Pi-TTD defines the control flow internal to a process with the functions from the original TTD as building blocks. Also the IPCs are visible as blocks defining the reception and sending of a message. Furthermore, any functions that are allowed to execute in parallel according to the original TTDs, are shown in sequence. The blocks corresponding to the sending and the reception of a message are associated with communication resources belonging to some communication model. The communication resource blocks are shown as rectangles in the Pi-TTD.

In Figure 8 the Pi-TTDs for the processes *P-STC* and *P-A* are shown. The first of the diagrams shows the *P-STC* process with visible blocks for interfacing the sender and receiver communication resources. Sending of a position message (*pos*) is shown as the block *S_pos*, reception of torque reference is tagged *R_torq* etc.

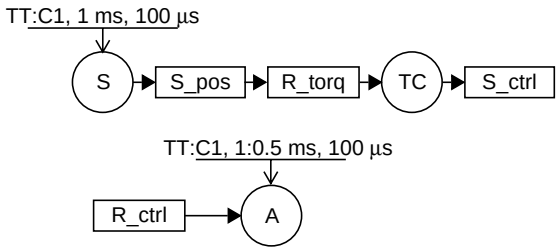


Figure 8. Process internal timing and triggering diagrams for process *P-STC* and *P-A*.

Note that all the diagrams shown in this example are specification diagrams used to specify a required behaviour rather than showing the results of an implementation. Just like in the TTD and I-TTD case however, the dia-

grams can be used with achieved values of the time triggers to show a result rather than a specification. In that case, triggers are assigned to the starting block of each process. Compare Figure 8 where the requirement specification is given for the timing of the actuate function.

To describe the behaviour of IPCs the AIDA models use communication resources (CRs) and chains of such, called communication resource chains (CR-chains). These provide the ability to describe various implementations ranging from an RTOS message queue to a complex communication system including software and networks. A communication resource is an entity used for communication. It can be either a message queue with an associated data storage and arbitration policy, or it may be a transmitter which does not store data. Typically, a register in a communication device is modelled as a one place buffer (in AIDA terms, a message queue of length one) and the communication media is described by a transmitter resource. A communication resource may be used in several CR-chains associated to different IPCs. This makes it possible to model priority arbitration between different messages using the same resource(s). The CR-chains also make it possible to describe priority arbitration in several levels (one in each resource). A CR-chain starts and ends with message queues to which the sending process writes and from which the receiving process reads the data (called sender and receiver resource respectively).

Figure 9 shows the CR-chain for the *ctrl* IPC from Figure 6 when the IPC is implemented over a Controller Area Network (CAN). *Mailbox 1* is a message queue resource describing the buffer in the CAN communication

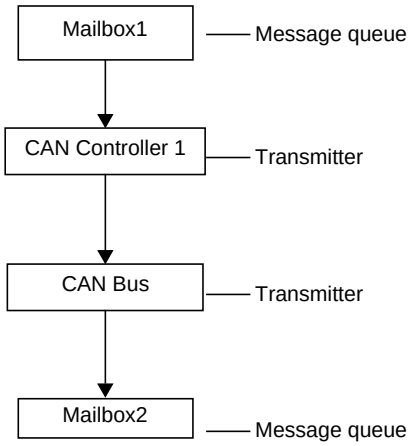


Figure 9. Visual description of a communication resource chain for an IPC using a CAN network.

chip of the sending node. The *P-STC* process 'writes' data to *Mailbox 1* when it wants to send data via the corresponding *ctrl* IPC. The *CAN controller 1* resource is a transmitter that performs arbitration on the available mailboxes that have data to send. The *CAN bus* is likewise a transmitter resource that, in priority order, selects one of the messages served to it by its connected CAN controllers. The last communication resource, *Mailbox 2* in the described chain, is the buffer in the CAN chip at the receiving node. *Mailbox 2* is modelled as a message queue from which the receiving process (*P-A*) reads the data.

3. QUALITATIVE COMPARISON WITH RELATED MODELLING APPROACHES

The purpose of this section is to provide another perspective to, and evaluation of, the AIDA models. Three interesting modelling approaches have been chosen here. These are representative in that they represent efforts from different 'communities' and methods including 'object-oriented analysis and design', 'structured analysis and design', and 'real-time scheduling research'. Some further comments on these choices and their relations to other modelling approaches are given below. The selection criteria are discussed in section 3.2.

3.1. Modelling approaches selected for comparison

- **RT-UML** - Real-time extensions to the Unified Modelling Language. UML is the result of a recent effort in the object oriented community to develop a unified (object-oriented) modelling language with a graphical notation, [29][30]. Consequently, UML incorporates most of the models and diagrams found in previously developed OO methods. The large UML repertoire includes diagrams for describing use cases, class/object

structures and relations, statecharts (state transition diagrams) and activity diagrams, sequences, collaboration, components and hardware deployment. UML also incorporates three extensibility mechanisms to extend or tailor the modelling capabilities. These mechanisms include constraints (rules applied to elements of a model), tagged values and stereotypes (with which new model elements can be created and associated with user-defined icons). UML is claimed to have very broad applicability also encompassing at least real-time event-triggered systems, [30][26]. One major question for research still appears to be to what extent object orientation and UML are sufficient for modelling hybrid real-time control systems. It is relatively well accepted that object orientation in general, see e.g. [31][6][25], and UML in its current form, see e.g. [20], are not sufficient for real-time systems. Limitations include the specification of timing constraints, the handling of periodic time-triggered systems, and the definition of specific computer system configurations. This is also being manifested by the ongoing work within the Object Management Group to define real-time extensions for UML. In this comparison the basis is the use of UML for real-time systems as described in [9] and the real-time extensions proposed in [20] and other related articles, [1].

- **RTT** - Real Time Talk**.** RTT is a design framework for developing distributed real-time applications, [10]. RTT is based on object orientation with the premise of reuse and modularity advantages. To support real-time systems a number of notions are incorporated into an extended object oriented framework. These notions include explicit descriptions of synchronization (ordering the execution of objects), timing requirements of individual and interacting objects, and communication between objects. RTT encompasses programming in the large and in the small, and includes considerations on how to combine soft and hard real-time software. In terms of timing models and analysis, RTT is closely linked to the real-time scheduling theory community. DEDOS, [31], and HRT-HOOD, [6] are two approaches that are related since they also extend traditional object oriented concepts with data- and control-flow specifications. As in RTT, timing analysis based on real-time scheduling theory is incorporated in the design guidelines.
- **DARTS/DA -** the design approach for real-time systems extended for distributed system, [12][13]. DARTS/DA builds upon the structured analysis and design (SA/SD) methods but extends these for concurrent distributed systems. In traditional SA/SD, the analysis and design steps are accompanied by a set of diagrams including hierarchical data-flow diagrams (DFD) which on the highest level describe the system context. The structure of a distributed computer system and of operating system processes are also described by DFD's. The DFD's may include control-flow (divided into triggers, enable and disable events) and control transformations. Additional models include state transition diagrams, data dictionaries, mini-specifications of primitive transformations, and structure charts showing the system decomposition into modules. DARTS/DA, and refined versions of it [13], elaborate 'task architecture diagrams' (i.e. the process structure and the inter-process communication principles) and also modularization into objects (information hiding modules), thus attempting to bring the structured and object oriented approaches together.

3.2. Selection of comparison criteria

To support the design issues targeted in this paper a modelling framework must enable multidisciplinary and concurrent development. It must enable high-level design trade-offs to be made and evaluated from both control and computer system perspectives. There is then a need to incorporate suitable viewpoints, relations and links between these views, see e.g. [32]. In addition, it is essential that the specification languages and models (in textual or graphical form) capture the designer's conceptual view of the system with minimum effort, [11][21]. I.e., the models should be natural to use and preferably directly and explicitly support the features that the designer wishes to model.

Several definitions and classifications of views exists. Some classifications focus on the users and/or subsystem viewpoints. This is for example the approach taken in the COntrolled REquirements expression requirements analysis method. This is related to the AIDA modelling parts, where domain views, corresponding to different disciplines are provided. A closely related classification of views is given by the "4+1 view model of architecture", [16]. The views include the *logical* view (describing system functionality and structure in terms of objects and classes), the *process* view (describing concurrency and communication), the *development* view (describing the organization of the actual software modules), the *physical* view (describing the mapping of the software on the hardware), and the *scenario* view, which is used to support architectural design and to indicate essential requirements.

Another classification is given in [21] where a view is said to describe a system with respect to some set of

attributes or concerns. The following views are defined: *system purpose* (i.e. requirements), *form* (i.e. structure), *behavioural/functional*, *performance*, *data*, and *managerial* (process related).

Which are the most essential views? This probably to a large extent depends on the type of system, compare with a database vs. a real-time control system. According to [21] the *system purpose* and *form* views are in general the most important ones. Given certain critical system requirements, models to analyse their satisfaction are also essential. For distributed real-time systems and with the focus of this paper it follows that the requirements and structure criteria (and views) need to be complemented by performance models that enable analysis and prediction of the timing behaviour of a particular system.

Based on the preceding discussion the following criteria for qualitative comparison have been chosen:

- *Timing behaviour - can the specified timing as well as the implemented timing behaviour be described with relation to system functions?* Timing behaviour is one of the key attributes and requirements addressed in this evaluation. There is a need to express relevant timing requirements for the targeted time- and event-triggered control systems, such as response and feedback delays, sampling periods, allowable jitter, synchronization and precedence. There is also a need to describe the timing behaviour of an implemented system at an application abstraction level, i.e. in terms of the actual or predicted delays and timing variations that are introduced due to computations, communications, blocking, non synchronous activities, and interference. Due to the different focuses of the evaluated models the distinction between time- and event-triggered systems is included. Note that time-triggered here primarily refers to the need to describe multirate systems where functions (objects or transformation) executing at different periods communicate. Behavioural descriptions of processes (threads), primarily in terms of the communication primitives (asynchronous vs. synchronous), are not included in the comparison. Such descriptions are available in mixed structural and behavioural descriptions in UML and DARTS/DA in terms of concurrency and data-flow diagrams.
- *Structure:* Given the targeted design issues it is clear that the structures corresponding to the application functions, threading in the software implementation, the hardware and environment interfaces need to be described. Links (mappings or allocations) between these structural representations must be supported. Hierarchies are an important part of structural descriptions and offer well known advantages in dealing with system complexity thereby allowing a designer to focus on different abstractions and subsystems at a time, [19], [11]. Different types of hierarchies exist; the most common is that of decomposition or refinement of objects (components, functions etc.) to describe the details of the objects in terms of lower level objects within the same domain. Compare for example with a computer hardware structural description starting at the network level, where each network is composed of nodes and communication links, and where each node is composed processors and peripherals, [8]. Nancy Leveson, [19], in the context of system specifications, discusses another type of hierarchy termed "*means-ends hierarchies*". According to [19] these have been shown to play an important role in the understanding of complex systems. "*Ends*" refers to "upper levels" and describes goals, whereas "*Means*" refers to lower level implementation descriptions. Such a hierarchy is applicable both to the structure (compare logical vs. process views) and to the timing behaviour.
- *Resource management - evaluating the performance models with respect to timing analysis:* Under the heading 'resource management' we group policies and mechanisms, in software and hardware, used to implement scheduling, process communication, interprocessor communication and synchronization. Related information includes descriptions of the provided processing and communication performance (e.g. overhead and effective bits/s in communication). Such descriptions are essential to be able to predict the timing behaviour of an application. Other important pieces of information for timing analysis, such as requirements on processing (e.g. the number of and type of operations required) and communication (e.g. messages/s) should be contained within the application description.
- *Graphical notation: evaluating the model diagrams for the engineering of distributed real-time systems.* Models in an appropriate graphical format are essential for describing complex systems (cmp. with [7]). This point should be seen as a general assessment of each of the models regarding the 'maturity' of the available graphical notations. This criteria is emphasized by the importance of models for communication, for example to provide an understanding of a system under construction, [21].
- *Guidelines: do guidelines exist that indicate how the models can be used in the design of distributed real-time systems?* Given a modelling approach there is a need for guidelines on how to use the models in system

design. A modelling approach may be closely associated with a particular design philosophy or more stand-alone. The emphasis here is to judge whether available guidelines do provide support for the design issues of Figure 1. A guideline should describe how the models can be used to represent the structure of the system, the requirements on the system, the timing behaviour of the implemented system and how the model contents can be used for analysis and other design activities. As for the timing behaviour criteria a distinction between time- and event-triggered systems is included here.

3.3. Qualitative comparison and discussion

For the related modelling approaches, the comparison should be seen as an attempt to answer the following question; How well does the modelling approach support the design issues of Figure 1? Although the comparison is qualitative it is of interest for the AIDA project also for other reasons. So far the AIDA project has focused on the multidisciplinary aspects of the models and their content rather than on the graphical notation and the language for describing the models. I.e., a further question is if the AIDA models can benefit from the other approaches in this respect.

Despite the fact that the modelling approaches chosen for comparison in some sense target distributed real-time systems, they are still rather different. The AIDA modelling framework is targeted to support certain specific design issues for real-time control systems. UML on the other hand is a “general purpose” modelling language with software origin that has been extended to better handle distributed real-time systems. Also DARTS/DA is more general in nature compared to AIDA. There is a closer similarity between AIDA and RTT since they are both targeted towards distributed real-time control systems, although AIDA is more focused on the modelling framework and RTT on software design.

The comparison is summarized in Figure 10 where, for each approach, full, partial or little/no support for the comparison criteria is indicated through black, gray and white bullets respectively.

	Timing Behavior: Time T. / Event T.	Structure	Resource Management	Graphical notation	Guidelines: Time T. / Event T.
AIDA	● ○	◉	●	○	● ○
RT-UML	○ ●	◉	○	●	○ ◉
DARTS/DA	◉ ◉	●	○	●	◉ ◉
RTT	◉ ○	○	◉	○	● ○

Figure 10. Qualitative comparison of modelling approaches for distributed real-time systems

Timing Behaviour: Time-triggered (TT) vs. Event-triggered (ET) systems

- The **AIDA** timing and triggering diagrams describe the required and implemented timing behaviour, cmp. with section 2. The models are developed primarily with TT multirate systems in mind. Limited support is provided for event-triggered parts where overall timing requirements on event-sequences can be specified. No details such as conditions on state transitions can be specified, thus the behaviour of micro-modes and modes can not be described in detail.
- **UML** can describe timing behaviour through sequence, activity and state transitions diagrams. In the real-time extensions of UML, the concept of modes and mode transitions, and timing information in sequence diagrams have been included, [1][20]. With the extended sequence diagrams it is principally possible to describe both the required and implemented timing behaviour. The basic modelling concepts, however, are developed for event-triggered object oriented systems. It remains to be shown how useful these diagrams are for time-triggered multirate systems. Compare with the pinpointing of this problem by ObjectTime, [25].

- **DARTS/DA** expresses requirements on timing textually. The behavioural model is based on data-flow diagrams extended with a discrete-event formalism. The data-flow diagrams thus combine pure data-flow with control flow, where the control transformations are linked to finite state machine descriptions. The data-flow diagrams have associated triggering semantics, transformations are triggered to execute upon the arrival of data. These descriptions are used for requirements and may encompass time- and event-triggered systems although they do not seem to have been applied to multirate systems. There are no explicit means or methods to describe the implemented timing behaviour.
- The **RTT** programming model is hierarchical and in descending order composed of modes, use-cases, tasks and objects. Tasks are used to encapsulate objects, and provide the thread of control and communication for the object. Timing behaviour is described through extended scheduling theory models; precedence graphs define the order in which tasks execute, the timing attributes of individual tasks (execution time, release time relative period and deadline relative period), constraints among tasks such as mutual exclusion, and the timing attributes of the precedence graphs (period time). Although multirate systems can be specified this is done in terms of task oriented timing constraints rather than application level timing requirements, [24]. On the other hand, the model is formal and an off-line scheduler tool has been developed and successfully commercialized. The description is focused on time-triggered systems for which off-line scheduling is used in the implementation. There is currently no way to fed back results from timing analysis to describe the actual behaviour of implemented tasks.

  Structure:
- **AIDA** includes provisions for the required structures (application functions, threading, the hardware and environment interfaces). The different hierarchical relations however need to be further developed.
- **UML** with its real-time extensions, has the models needed to describe an object-oriented structure, threading in the software implementation, the hardware and environment interfaces. Again, however, the basic modelling concepts are developed for event-triggered object oriented systems where functionality and transformational views traditionally have been de-emphasised. One slight problem with the object and class views are that they really constitute integrated views combining a number of hierarchies; aggregation, inheritance and associations. Aggregation is a decomposition of functionality (or composition of objects) whereas inheritance defines an additional class hierarchy. Associations, on the other hand, can be used to model for example inter object communication, i.e. data-flow. Although all these views are valuable it is difficult to see what a control engineer would gain from having them all integrated. Nevertheless, a tool could of course in principle solve this problem by only showing associations corresponding to data-flow, thus in principle converting the class diagram to a functional block diagram. Alternatively, a collaboration diagram could be used to show functional structure, [1]. Further evaluation is required to show that UML is useful for time-triggered control systems and that the models can satisfy the “conceptual views” of designers.
- **DARTS/DA** provides models for describing a so called physical model, i.e. hardware and interfaces, and a threading structure. The functional structure is given through the combined behavioural data-flow and control-flow diagrams. Links between these structures are supported. This provides a relatively complete view of the required structures.
- **RTT** describes the object structure and object communications and the task structure (for time-triggered threads). No hardware structure is given.

  Resource management:
- **AIDA** focuses strongly on this issue, compare with Figure 2. None of **RT-UML, DARTS/DA** and **RTT** come close to these types of models. **RTT** provides partial information here; for the hard-real time model there is ample of information regarding timing requirements and the off-line scheduling. However, other aspects are very limited.

  Graphical notation:
- In **AIDA** specific work on a graphical notation has mainly focused on the timing and triggering diagrams. These were developed to complement functional block diagrams and to separate timing behaviour from structural specifications. Both **UML** and **DARTS/DA** lean on diagramming concepts that have been used for quite some time although in different notational versions. For **UML** there is still work to do on the real-time extensions. As for AIDA, **RTT** provides a graphical notation only for parts of the models.

Guidelines: Time-triggered (TT) vs. Event-triggered (ET) systems

- For **AIDA** this is a strong point with respect to *TT* systems although there is as yet no single condensed guideline. The research in [22][24][27][28] provides guidelines on the derivation of timing requirements and constraints, control system structuring, allocation, choice of execution and communication strategies, and exploits scheduling theory for timing analysis. This is also a strong point for **RTT**, partly carried out in cooperation with the AIDA-team. For both, there is much less emphasis on *ET* systems.
- For **UML** the numerous existing OO-methodologies provide good support in the design of event-triggered non real-time systems. Guidelines which attempt to cover distributed real-time systems include [16][26][9]. These however provide relatively little support for these issues. For example, in [9], partitioning, the use of deployment diagrams, and links to scheduling are briefly discussed. The guidelines are similar in nature for **RT-UML**, [1] which however place more emphasis on modelling concurrency, timing information in sequence diagrams, the hardware representation and the system context. Regarding the event-triggered parts, research developments in timed automata, formal semantics, state machine analysis and verification techniques (see e.g. [17]) still remain to be exploited. All in all there is room for more work on guidelines having both time- and event-triggered systems in mind for object-oriented systems.
- **DARTS/DA** provides guidelines in terms of heuristics for allocation and partitioning. Some timing analysis considerations for ET and TT systems are also incorporated based on fixed priority scheduling [13].

3.4. Concluding discussion on the AIDA models: views and model relations

The “4+1 view model of architecture”, [16], corresponds relatively well to AIDA. The *logical* view matches the AIDA application models, and the *process* and *physical* views match the structural parts of the AIDA computer system model. There is no direct correspondance in AIDA to the *development* view describing the software module organization since this topic is not within the scope of AIDA. The computer system resource models come closest to the development view in that it provides a layered view of the implementation software and hardware. The scenario view can be seen as a correspondance to the timing requirements part of the AIDA application models. A difference in AIDA is that it in addition incorporates these requirements and implementation models for both the application and computer system timing behaviour models. The AIDA resource models, cmp. Figure 9, do not seem to have a correspondance in the “4+1 view model” which are more focused on describing system structure.

Compared to the classification given in [21], the AIDA models provide a set of requirements focused on timing, structural, timing behaviour and performance models. There is no data view or model in AIDA. There is no explicit *managerial* (process related) view but the models are related to an associated design method.

The multidisciplinary views provided in AIDA could be interpreted as *means-ends* hierarchies (the term introduced in [19], recall section 3.2.) for both structure and timing behaviour. The latter hierarchy is established by the functions in the application model that describe the required timing behaviour and the computer system model which describes the implementation in terms of schedulable processes and the resources and policies used in the implementation.

Reindeer is a recent research project where an environment to support design of distributed systems is being developed, [8]. Integrated within Reindeer is a diagramming methodology where sets of diagrams are provided for each hierarchical level of design (requirements, network, node, processor, task). The utilized diagrams can be seen as an extension to DARTS/DA incorporating also for example sequence diagrams and use cases. Relations between these diagrams through the hierarchies are also provided. The proposed diagrams to a large extent correspond to the AIDA models wherefore this project is of interest to study further.

4. CONCLUSIONS AND FUTURE WORK

The paper has described the modelling framework that has been developed within the AIDA project. The AIDA modelling framework is by no means complete, there still remains work and research on some of the submodels. Future work on the AIDA modelling framework will include refinements and more exact semantic definitions of the models and their hierarchies. Since the emphasis in the design of the modelling framework has been the modelling of time-triggered control systems, the modelling of event-triggered systems is only partially supported. The direction the models should take here is open at this stage. From Figure 10. and with the extensibility mechanisms of UML in mind it is straightforward to say that the combination, or integration, of the AIDA modelling framework

with RT-UML could be very fruitful. Further evaluation of RT-UML in tool implementations will therefore be carried out.

Desired analysis and synthesis capabilities of the prototype tool-set will be investigated. The primary idea is to exploit advances in real-time scheduling theory that can be incorporated into the toolset. The execution time analysis that is supported by the models is coarse. Functions for more accurate execution time analysis are not planned to be a part of the AIDA tool-set even though it may be of interest to include this in the future. The toolset will provide for further evaluation possibilities of the models and their graphical notations.

An interesting issue for further model developments is the consideration of including other architectural aspects into to the framework. A prime candidate here is dependability requirements and their implementation. Today, the AIDA models can be used to describe certain requirements and behavioural characteristics of redundant/fault-tolerant systems; for example, a particular redundancy approach usually comes along with fixed and specified strategies for synchronization, scheduling, etc. Explicit specifications of fault-tolerance requirements and the mechanisms for error detection and handling are currently not part of the models.

It appears that there are still several research challenges in modelling, in particular regarding “hybrid” systems composed of both time- and event-triggered subsystems, the relations between views, and how all this can be incorporated into a design tool provided with appropriate graphical representations to be beneficial for users. It is also clear that tool and model integration will be increasingly called for in the future to combine the strength of different modelling formalisms and analysis tools.

These interesting problems and probably many more are ahead of us as we embark on the prototype tool implementation.

5. ACKNOWLEDGEMENTS

This work was supported in part by the Mechatronics trade-association, the Swedish National Board for Industrial and Technical Development (NUTEK) and the Volvo Research Foundation. The authors want to acknowledge the AIDA reference group for providing us with useful input in the development of the modelling framework. In addition De Jiu Chen and Marcelo Masera are acknowledged for valuable comments.

REFERENCES

[1] ARTISAN Software. http://www.artisansw.com/articles/index.html

[2] AIDA (1998). http://www.damek.kth.se/~martin/aida.html

[3] Ancilotti P. et al. Design and Programming Tools for Time Critical Applications. *J. of Real-Time Systems*, Vol. 14, No. 3, 1998.

[4] Bass J. M. et al. (1994), *Automating the Development of Distributed Control Software*. IEEE Parallel & Distributed Tech., Winter 1994.

[5] Bhatt D. (1996), *A methodology and toolset for the design of parallel embedded systems.* In Proc. of the School on Embedded Systems, Organised by the European Educational Forum, Veldhoven, NL, Nov. 1996

[6] Burns A., Wellings A.J., (1994). HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. *J. of Real-Time Systems*, Vol. 6, No. 1, 1994.

[7] Cooling J.E. Real-time software systems: an introduction to structured and object-oriented design. Int. Thomson Computer Press, 1997.

[8] Cooling J.E. et al. The design of software for real-time systems. *European Embedded Systems Conf.* Sept. 1998.

[9] Douglass B.P. (1998). *Real-time UML*: Developing efficient object for embedded systems. Addison Wesley Longman, Inc.

[10] Eriksson C., et al. (1996). An Overview of RealTimeTalk, a Design Framework for Real-time Systems. J. of Parallel and Distributed Computing, No. 36, p 66-80, Academic Press Inc.

[11] Gajski. D.D. et al. Specification and design of embedded systems. Prentice Hall 1994.

[12] Gomaa H. A software design method for distributed real-time applications. J. of Systems and Software, 9, 81-94. Elsevier Science Publishing.

[13] Gomaa H. *Software design methods for concurrent and real-time systems*. Addison-Wesley publishing company, 1993.

[14] Hanselmann H. (1995), *DSP in Control: The Total Development Environment.* Int. Conference on signal processing applications and technology. Oct. 24-26, 1995, Boston, MA, USA.

[15] Hansson H. et al. (1998). BASEMENT: A distributed real-time architecture for vehicle applications. *J. of real-time systems*, 11, p. 223-244. Kluwer Academic Publishers.

[16] Kruchten P. The 4+1 view model of architecture. *IEEE Software* Nov. 1995, 12 (6), pp. 42-50.

[17] Larsen et al. (1997). UPPAAL in a Nutshell. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1/2).

[18] Lauwereins R. et al. (1995), *Grape-II: A system-level prototyping environment for DSP applications.* IEEE Computer, Feb. 1995, pp. 35-43.

[19] Leveson N.G. Intent Specifications: An approach to building human-centered specifications. 1998. *IEEE*

[20] McLaughlin M.J. and More A. (1998). Real-time extensions to UML: Timing, concurrency and hardware interfaces. *Dr. Dobb's Journal* December 1998.

[21] Rechtin E. and Maier M.W. (1997). *The Art of System Architecting*, ISBN 0-8439-7836-2, CRC Press, 1997.

[22] Redell O. *Modelling of Distributed Real-Time Control Systems, An Approach for Design and Early Analysis*, Licentiate Thesis, Dep. of Machine Design, KTH, 1998, TRITA-MMK 1998:9, ISSN 1400-1179, ISRN KTH/MMK--98/9--SE

[23] Roppenecher G. and Wallentowitz H. (1993). Integration of chassis and traction control systems: What is possible- What makes sense - What is under development. J. of Vehicle System Dynamics, 22 (1993), pp. 283-298.

[24] Sandström K. Eriksson C. Törngren M. (1999). Modeling and scheduling of control systems. Research Report Dep. of Machine Design, KTH, TRITA-MMK 1999:4, ISSN 1400-1179, ISRN KTH/MMK--99/4--SE

[25] Selic B. Periodic Tasks in Room. Position paper submitted to the *Workshop on OO Real-Time Systems, ACM OOPSLA '95 Conference*, Austin Texas, Oct. 15-19, 1995.

[26] Selic B. and J. Rumbaugh. Using UML for Modelling Complex Real-Time Systems. White Paper. www.rational.com/uml/resources/whitepapers

[27] Törngren, M. and Wikander, J., (1996). A Decentralization Methodology for Real-Time Control Applications. *Journal of Control Engineering Practice*, Vol. 4, No 2, pp. 219-228, 1996. Elsevier Science.

[28] Törngren M. (1995), *Modelling and design of distributed real-time control applications*. Doctoral thesis, Department of Machine Design, KTH, TRITA-MMK 1995:7, ISSN1400-1179, ISRN KTH/MMK--95/7--SE.

[29] UML notation guide. Version 1.1. Sept. 1997. Object Management Group, doc. no. ad/97-08-05. http://www.rational.com/uml

[30] UML Summary. Version 1.1. Sept. 1997. Object Management Group, doc. no. ad/97-08-03.

[31] Verhoosel J. (1995), *A Model for Scheduling of Object-Based, Distributed Real-Time Systems*, J. of Real-Time Systems, Vol. 8(1), January 1995

[32] Vestal S. Integrating Control and Software Views in a CACE/CASE toolset. *IEEE/IFAC Joint Symposium on Computer Aided Control System Design for Control Systems*, pp. 353-358, Tucson, AZ, 1994.

[33] Wikander J. and Törngren M. Mechatronics as an engineering science. Proc. of the *6th UK Mechatronics Forum Int. Conf.* Skövde Sweden, 9-11 September 1998.