Formal and Probabilistic Arguments for Component Reuse in Safety-Critical Real-Time Systems

H. Thane and A. Wall

Mälardalen Real-Time research Center (MRTC)

Mälardalen University, Västerås, Sweden

{awl,hte}@mdh.se

Abstract: In this paper we are going to introduce a novel framework for formal and probabilistic arguments of component reuse in safety-critical real-time systems. Using both quantitative and qualitative descriptions of component attributes and assumptions about the environment, we can relate input-output domains, temporal characteristics, fault hypotheses, reliability levels, and task models. Using these quantitative and qualitative attributes we can deem if a component can be reused or not. We can also deem how much and which subsets, of say input-output domains, that need additional functional and safety verification. This framework will give formal and probabilistic arguments for reuse of components in safety-critical real-time systems.

1 Introduction

The introduction of computers into safety-critical systems lays a heavy burden on the software designers. The public and the legislators demand reliable and safe computer systems, equal to or better than the mechanical or electromechanical parts they replace. The designers must have a thorough understanding of the system and more accurate software design and verification techniques than have usually been deemed necessary for software development. However, since computer related problems, relating to safety and reliability, have just recently been of any concern for engineers, there exists no holistic engineering knowledge of how to construct safe and reliable computer based systems. There exist only tidbits of knowledge and no silver bullets that can handle everything. Some people do nonetheless, with an almost religious glee, decree that their method, principle or programming language handles or kills all werewolves (bugs) [8].

In order to be able to design software that is as safe and reliable as the mechanical or electromechanical parts it replaces, and to reduce the time to market (TTM) focus of current research and practice has turned to the reuse of proven software components. The components can be arbitrary small or large in terms of functionality. The basic idea is that the system designer should be able to procure (or reuse) software components in the same manner as hardware components can be acquired. Hardware component like nuts, bolts, CPUs, memory, A/D converters can be procured based on functionality, running conditions (environment) and their reliability. Hardware components are typically divided into two classes, commercial and military grade components, where military grade electronics can withstand greater temperature spans, handle shock and humidity better than commercial components. However, for software it is not so easy to make the same classification since the metrics of software are not based on physical attributes, but rather on its design, which is unique for each new design. In fact, software has no physical restrictions what so ever, like mass, size, number of components. Software has neither structure nor function related attributes like strength, density and form. The sole physical entity that can be modeled and measured by software engineers is *time*. With but a few exceptions all safety critical systems are also real-time systems.

In this paper we are going to introduce a framework for formal and probabilistic arguments of component reuse in safety-critical real-time systems, based on the restrictions that time, and the component contracts impose on the system behavior.

1.1 What are software components?

There exists yet no commonly agreed upon definition of what constitutes a software component. Most people do however agree that the concept of components is the idea of reusable software entities. In this paper, for the sake of enabling analysis of components in safety-critical real-time systems we do not only define a component in terms of software. A component must also be well documented in terms of design documentation and preferably also in terms of test documents, and in evidence of the reliability achieved.

We have chosen a hierarchical/recursive definition of components in real-time systems (RTS). The smallest components are the actual assembly instructions provided by the processor, some way up in the hierarchy a component is a task or process. A task is a unit that usually performs a single calculation, for instance calculating a new process value based on measurements collected by another task. On the next level, several tasks can work together in a transaction where the transaction itself is a component. A transaction is a set of functionally related and cooperating tasks, e.g., sample-calculate-actuate loops in control systems. The relationship between the cooperating tasks with respect to precedence (execution order), interactions (data-flow), and a period time typically define each transaction.

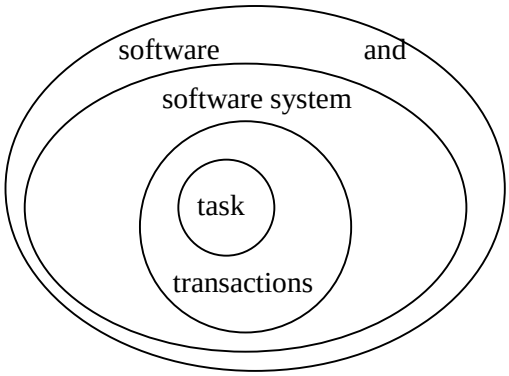

Figure 1. Hierarchical definition of components for real-time systems.

All tasks and transactions together make up a software system that also might be a component from the complete systems point of view. As an example, the software controlling the air bag in a car could be considered as a component from the air bags point of view.

As this definition eventually forces us to define the universe as a software component, we have to stop the recursion somewhere. At least on the level of abstraction in this paper, the operating system is not considered a component. We like to think of the operating system as the infrastructure for the computer system. It provides the necessary services for the software but it also restricts the possibilities of implementing the requirements put upon tasks and transactions.

2 Component contracts

A contract is a specification for a component in terms of what functionality a component provides. Furthermore, the contract states the conditions under which the specified functionality works. In other words, a contract of a software component is a statement saying that: provided that certain assumptions are fulfilled, the specified functionality will be provided and error free. Such contracts can be described with arbitrary formality. The more formal the contracts are, the richer possibilities of verifying that the component will work correctly in a system, or that the software system will work correctly with the new component as a part of it. In this paper we will settle for a functional description in some natural language although we could use some formal temporal language such as timed automata [1]. However, the main focus in this paper is on the specific demands put on component contracts in safety-critical real-time systems.

As the temporal behavior is of great importance for the correctness of real-time systems, the temporal assumptions have to be specified in the contracts. The temporal attributes make up the task model that is used when formally verifying the temporal behavior of the system, i.e., if the system is schedulable or not. However, there exist two different task models for tasks in a real-time system, one that exists on the design level and one that is actually provided by the given infrastructure. Typically, the design task model consists of end-to-end deadlines for transaction, jitter constraints, etc which is realized in the implementation using the temporal attributes at hand in the infrastructure. As a consequence, we will divide the contract into two different main parts, the design task model and the infrastructure task model.

2.1 The design task model

As discussed in the introduction, our view on components in real-time systems is that they are either single tasks or a transaction consisting of tasks and transactions. Thus, the contract must specify whether or not the component is a transaction or not. If the component is indeed a transaction it has a precedence relation between the cooperation tasks and there is a flow of data between the tasks. Note that a single task is a specialization of a transaction as it is a transaction without any precedence relations or data-flow. There might be several different ways to represent precedence relations. We will, however settle with a simple graphical notation called a precedence graph. A precedence graph is simply a directed graph visualizing all tasks in the transaction interconnected by an arrow indicating the direction of the precedence relation between those tasks. In Figure 2 a simple precedence graph is shown where task *A* precedes, task *B* and task *C*. We can also see that task *B* and *C,* precede task *D*.

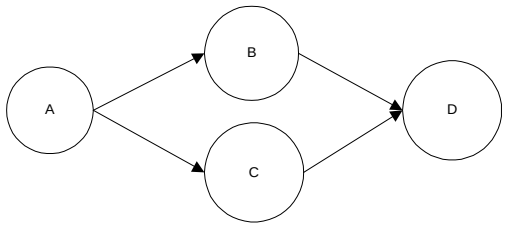

Figure 2 A simple precedence graph

Concerning the data-flow in a component we are interested in exposing three different attributes in the contract namely: where, how and what. The attribute “Where”, declares the receiver of data, i.e. another task within the component. The data can flow throughout the component using different strategies. The two different possibilities are synchronous and asynchronous. If data is transported in an asynchronous manner, the contract must specify how the data is treated on the receiver side. This is important since data can be consumed either faster or slower than it is produced in a multi-rate transaction. If data is produced in a faster pace than it is consumed, the contract must specify whether or not new data should overwrite old data or if data should be buffered to provide history. Finally, concerning communication, the size of data is important, as the size will restrict the speed at which the component can execute.

The temporal attributes for a real-time component specifies the required temporal behavior for the component. As discussed, these attributes must not necessarily have there correspondence in the infrastructure, but should rather be realized using the constructions provided by the infrastructure. As we focus on reuse in this paper, we want our components to be as general as possible in terms of the temporal constraints. The generality is obtained by specifying the temporal constraints as time intervals. Later on in Section 3, we will elaborate further on how these intervals are obtained. Transaction components (where a single task is a special case) have period times, jitter constraints on period times, end-to-end deadlines, etc. However, it is not trivial to represent the period time for a component consisting of several tasks all running with different period times (multi-rate). We propose that period times are only valid for single-task components. The period time of a transaction is inherited from the system in which the component will be reused. The tolerances on the period times of a component are given in terms of decreased/increased inter arrival times, however where the precedence, and mutual exclusion relations, are preserved between the sub-components. This is a way of parameterize the component to fit in a new environment.

2.2 Infrastructure

Derived from the recursive definition of components and component relations, we can with respect to real-time systems define the components forming the basis for running the real-time tasks as the infrastructure, i.e., the real-time operating system and its attributes. We are now going to give a qualitative classification of the infrastructure for real-time operating systems based on their execution strategy, synchronization mechanisms, and communication mechanisms.

Execution strategy

The execution strategy of a real-time system defines how the tasks of the system are run. A usual classification is event and time triggered systems. An event-triggered system is a system where the tasks activation is determined by an event that can be of external or internal origin, e.g., an interrupt, or a signal from another task. There are usually no restrictions what so ever on when events are allowed to arrive, since the time base is dense (continuos). For time-triggered systems events are only allowed to arrive into the system, and activate tasks, with a certain minimum and maximum inter-arrival time; the time-base is sparse. The only event allowed is the periodic activation of the real-time kernel, generated by the real-time clock.

Another characteristic of an execution strategy is the support of single tasking or multitasking. That is, can multiple tasks share the same computing resource (CPU), or not? If the infrastructure does support multitasking does it also support task interleavings, i.e., does it allow an executing task to be preempted during its execution by another task, and then resumed after the completion of the preempting task?

A very common characteristic of an execution strategy, especially for multi-tasking RTS, is the infrastructure task model. The task model defines execution attributes for the tasks [2][7][11][12]. For example:

- **Execution time**, C_j. Where $C_j \subseteq [BCET_j, WCET_j]$, i.e., the execution time for a task, *j*, varies depending on input in an interval delimited by the task's best case execution time ($BCET_j$) and its worst case execution time ($WCET_j$).
- **Periodicity**, T_j. Where $T_j \subseteq [T_j^{min}, T_j^{max}]$, i.e., the inter-arrival time between the activations of a task, *j*, is delimited by an interval ranging from the minimum inter-arrival time, to the maximum inter-arrival time. For strictly periodic RTS, $T_j^{min} = T_j^{max}$.
- **Offset**, O_j. Where O_j defines an offset relative the period start at which the task *j* should be activated. Offsets also go by the name release times.
- **Priority**, P_j. Where P_j defines the priority of task *j.* When several tasks are activated at the same time, or when an activated task has higher priory than the currently running task, the priority determines which task should have access to the computing resource.
- **Deadline**, D_j. Where D_j defines the deadline for the task, *j*, that is, when it has to be finished. The deadline can be defined relative task activation, its period time, absolute time, etc.

For different infrastructures the task model vary with different flavors of the above-exemplified attributes.

Synchronization

Depending on the infrastructure we can either make necessary synchronizations between tasks off-line if it is time triggered and supports offsets, or we can synchronize tasks on-line using primitives like semaphores. In the off-line case we guarantee precedence and mutual exclusion relations by separating tasks time-wise, using offsets.

Communication

Communication between tasks in RTS can be achieved in a multitude of ways. We can make use of shared memory which is guarded by semaphores, or time synchronization, or we can via the operating system infra structure send messages, or signals between tasks. Depending on the relation between the communicating tasks, in respect to periodicity, the communication can vary between totally synchronous communication to totally asynchronous communication. That is, if task *i* sends data to task *j*, and both tasks have equal periodicity, $T_j = T_i$, we can make use of just one shared memory buffer. However, if $T_j > T_i$ or $T_j < T_i$ the issue gets more complicated. Either we make use of overwriting semantics (state-based communication), using just a few buffers, or we record all data that has been sent by the higher frequency task so that it can be consumed by the lower frequency task when it is activated. There are several approaches to solving this problem [3][4][5].

2.3 Failure semantics

Components can fail in different ways. The manner in which they fail can be categorized into failure modes. Failure modes are defined through the effects, as perceived by the component user:

1. ***Sequential failure behavior***

   This failure mode includes:

   - *Control failures*, e.g., selecting the wrong branch in an if-then-else statement.
   - *Value failures*, e.g., assigning an incorrect value to a correct (intended) variable.

- *Addressing failures*, e.g., assigning a correct (intended) value to an incorrect variable.
- *Termination failures*, e.g., a loop statement failing to complete because the termination condition is never satisfied.
- *Input failures, e.g.*, receiving an (undetected) erroneous value from a sensor.

Multitasking and real-time failure behavior

2. ***Ordering failures***, e.g., violations upon precedence relations or mutual exclusion relations.
3. ***Synchronization failures***, i.e., ordering failures but also deadlocks.
4. ***Interleaving failures***, e.g., side effects caused by non-reentrant code, and shared data, in preemptively scheduled systems.
5. ***Timing failures.*** This failure mode, yields a correct result (value), although the procurement of the result is time-wise incorrect. For example, deadline violations, early start of task, incorrect period time, too much jitter, too many interrupts (too short inter-arrival time), etc.
6. ***Byzantine and arbitrary failures***. This failure mode is characterized by a non-assumption, meaning that there is no restriction what so ever with respect to which effects the component user may perceive. Therefore, has the failure mode been called malicious or fail-uncontrolled. This failure mode includes two-faced behavior: a component can output "X is true" to one component user, and "X is false" to another component user.

The above listed failure modes build up a hierarchy where byzantine failures are based on the weakest assumption (a non-assumption) on the behavior of the components, and sequential failures are based on the strongest assumptions. Hence byzantine failure is the most severe and sequential failure the least severe failure mode. The byzantine failure mode covers all failures classified as timing failures, which in turn covers interleaving failures, and so on.

More formally: Sequential failures $\subset$ Ordering failures $\subset$ Synchronization failures $\subset$ Interleaving failures $\subset$ Timing failures $\subset$ Byzantine failures. That is, $1 \subset 2 \subset 3 \subset 4 \subset 5 \subset 6$.

The component user can also characterize the failure modes according to the viewpoints domain and perception. A distinction can be made between primary failures, secondary failures and command failures [6]:

- *Primary failures*

  A primary failure is caused by an error in the software of the component so that its output does not meet the specification. This class includes byzantine failure modes, and sequential failure modes.

- *Secondary failures*

  A secondary failure occurs when the input to a component does not comply with the specification. This can happen when the component is used in an environment not designed for, or when the output of a preceding task (component) does not comply with the specifications of a succeeding task's (component) input. This class includes interleaving failures, and sequential input failure modes.

- *Command failures*

  Command failures occur when a component delivers the correct result but at the wrong time or in the wrong order. This class covers timing failures, synchronization failures, and ordering failures.

The above classification of failure modes is not restricted to individual instances of failures, but can be used to classify the failure behavior of components, which is called a component's failure semantics [9].

- *Failure semantics*

  A component exhibits a given failure semantic if the probability of failure modes, which are not covered by the failure semantic, is sufficiently low.

If a given component is defined to have ordering failure semantics, then all individual failures of the component should be ordering or sequential failures. The possibility of more severe failures, like timing failures, should be sufficiently low. The failure semantic is a probabilistic specification of the failure modes a component may exhibit, which has to be chosen in relation to the application requirements. In other words, the failure semantics defines the most severe failure mode a component user may have to consider. Fault-tolerant systems are designed with the assumption that any component that fails will do so according to a given failure semantic. If nonetheless, the failure of a component violates the failure semantic, then nothing is guaranteed and the whole system might fail. This consideration leads to the important concept of assumption coverage [10], which is closely related to the definition of failure semantics.

- *Assumption coverage*

  Assumption coverage is defined as the probability that the possible failure modes defined by the failure semantic of a component proves to be true during practical conditions on the fact that the component has failed.

For components defined as having byzantine failures semantics the assumption coverage is always 1.0, i.e., the confidence in the failure semantic is total. This is safe since byzantine failures belong to the most severe failure mode. In all other cases the assumptions of the failure semantics may be violated because the component may show byzantine behavior. Consequently, will assumption coverage (confidence) be less than total (1.0).

The assumption coverage is a critical parameter when designing and reusing reliable components. If the assumptions are relaxed too much in order to achieve good assumption coverage, the design becomes overly complicated since severe failure semantics (like byzantine) have to be considered. On the other hand, if too many assumptions are made, the system design is easier, but the assumption coverage might be unacceptably low. In practice there is thus a need to compromise between system complexity and assumption coverage.

3 Component based analysis

In this section we are going to introduce a framework for formal and probabilistic arguments of component reuse in safety-critical real-time systems. The basic idea is to provide evidence, based on the components contracts and the experience accumulated, that a component can be reused immediately, or if only parts can be reused - or not at all. That is, we want to relate the environment for which the component was originally designed and verified for, with the new environment where it is going to be reused. Depending on the match with respect to input-output domains and temporal domains we want to deem how much of the reliability from the earlier use of the component can be inherited in the new environment. Faced with a non-match we are usually required to re-verify the entire component. However, we would like to make use of the parts that match and only re-verify the non-matching parts.

Specifically, we are now going to introduce two analyses, one with respect to changes in the input-output domain, and one with respect to changes in the temporal domain.

3.1 Input-output domain analysis

We are now going to establish a framework for comparative analysis of components input-output domains, i.e. their respective expected inputs and outputs, as defined by the components contracts (interfaces). But, we are also going to relate the input-output domain to its verified and experienced reliability, according to certain failure semantics. We can represent the input-output domain for a component, c, as a tupel of inputs and outputs; $\Pi(c) \subseteq I(c) \times O(c)$. Where the input domain, $I(c)$, is defined as a tupel of all interfaces and their respective sets of input; $I(c) \subseteq I(c)_1 \cup \ldots \cup I(c)_n$. Further, the output domain, $O(c)$, is defined as a tupel of all interfaces and their respective sets of output; $O(c) \subseteq O(c)_1 \cup \ldots \cup O(c)_n$.

The basic idea is that, given that we have a reliability profile, $R(c)$, of a component's input-output domain, $\Pi(c)$ and that the attributes for the real-time components are fixed, we can deem how well the component would fare in a new environment. That is, we would be able to make comparative analysis of the experienced input-output domain and the domain of the new environment.

For example, assume that we have designed and verified a component, c, that has an input domain $I(c)$ corresponding to a range of integer inputs, $I(c)$ = [40,70], to a certain level of reliability.

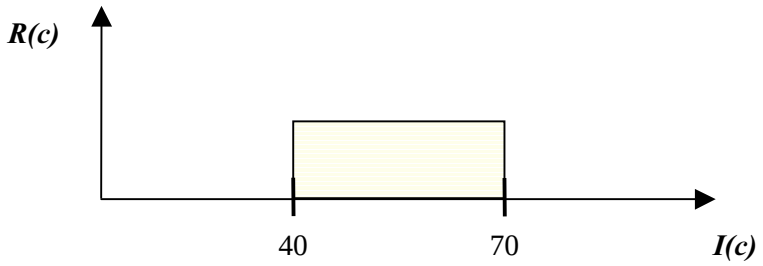

Figure 3 The reliability for input domain [40,70].

Consider now that we reuse this component in another system, where all things are the same except for the input domain, $I_2(c)$ = [50,80], and that we by use and verification have achieved another level of reliability.

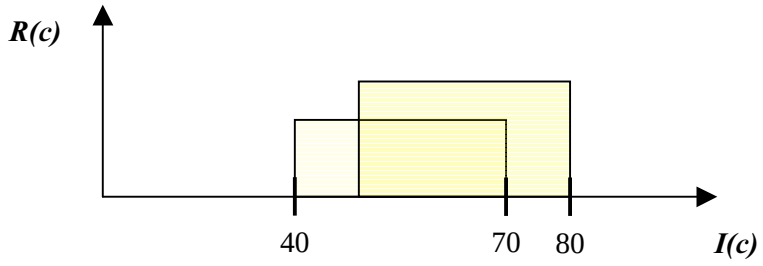

Figure 4 The reliability for joint input domain.

We can now introduce a new relation called the experienced input-output-reliability domain, $E(c) \subseteq (\Pi_1(c) \times R_1(c)) \cup \ldots \cup (\Pi_n(c) \times R_n(c))$, which represent the union of all input-output domains and the achieved reliability for each of these domains. This union is illustrated in Figure 4. Using this $E(c)$ we can deem if we can reuse the component immediately in a new environment $\Pi_{new}(c)$ and inherit the experienced reliability, or we can put another condition on the reuse, a reliability requirement.

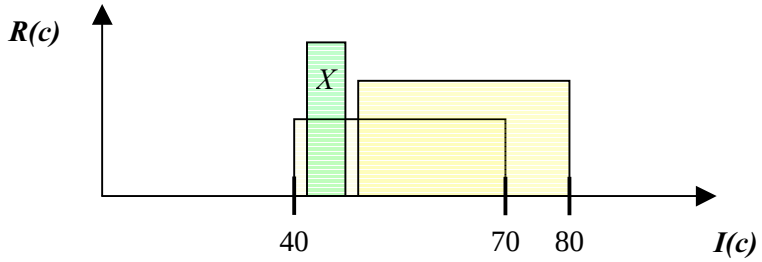

Figure 5 Reliability cannot be guaranteed in new environment.

In Figure 5, the new environment and the area named *X*, illustrates the reliability requirement. If it can be shown that $\Pi_{new}(c) \times R_{new}(c) \subseteq E(c)$ then the component can be reused without re-verification. However if the reliability requirement cannot be satisfied as illustrated in Figure 5, reuse cannot be done without additional verification.

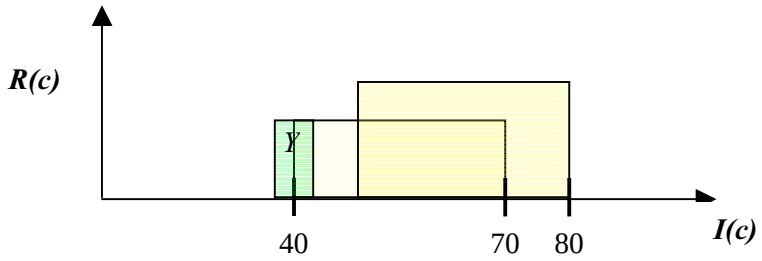

Figure 6 Verification only necessary for limited area.

If the $\Pi_{new}(c) \times R_{new}(c) \subset E(c)$, then we cannot immediately reuse the component without re-verification, however we need not re-verify the component entirely. It is sufficient to verify the part of the input domain that is non-overlapping with the experienced one, i.e., $(\Pi_{new}(c) \times R_{new}(c)) \setminus E(c)$.

By making this classification we can provide arguments for immediate reuse, arguments for re-verification of only cut sets and non-reuse. For example, assume that the $\Pi_{new}(c)$ (the area named *Y* in the Figure 6) has an $I_{new}(c) = [35,45]$ then we can calculate the cut set for $I_{new}(c)$ and $I(c)$ to be $I_{cut}(c) = [35,40]$

3.2 Temporal analysis

Whenever a new component is to be used in a real-time system, the new temporal behavior of the system must be verified, i.e. that all tasks are schedulable. For real-time systems schedulability means that no task in the system will ever violate its timing requirements, for instance deadlines, start times, jitter, etc. In order schedule a system the components have to be decomposed into their smallest possible entities, i.e., their tasks. Thus, the level of abstraction provided by the component concept is violated.

In this section we will discuss the impact of changing some of the temporal attributes for a component when reusing it. The following situations are dealt with:

- The same infrastructure and the same input-output domain
- The same input-output domain but a different infrastructure

A change in the temporal domain is always initiated by the system in which the component will run in. For instance, if the new system executes on different hardware, the execution times might vary (a faster or slower CPU). The changes can also originate from the requirements of the new system, for example the component has to run with a higher frequency than originally designed for.

3.2.1 The same infrastructure and input-output domain

In this case, the infrastructure is exactly the same in the new environment as the one previously used. Consequently, we have an exact match between the sets of services provided (denoted as S below). The services required by the design task model for the component have been satisfied earlier which implicates that they still are satisfied in the new environment. Thus, $S(c) \subseteq S_{old}(infrastructure)$ and $S_{new}(infrastructure) = S_{old}(infrastructure)$ holds, where $S(c)$ is the set of services required by component c.

Moreover, the new input-output domain for the new instance of component c, $\Pi_{new}(c)$, is completely within the verified range of input-output for the component $\Pi(c)$. Formally the following must hold: $\Pi_{new}(c) \subseteq \Pi(c)$.

The only alteration is in one or several of the temporal attributes for the component. Such a change requires the system to undertake a schedulability analysis [2][7][12] where the component is decomposed into its smallest constituents, the tasks. The component can be considered to fit into the new system if the system is schedulable and the relations between the tasks in the component, as well as the tasks in the new system, are not violated.

3.2.2 The same input-output domain but different infrastructure

There exist two different types of infra-structural changes:

- One where the infrastructures are different, but where the infrastructure parts pertaining to the component are the same. More precisely, if $S(c) \subseteq S_{new}(infrastructure) \cap S_{old}(infrastructure)$, then the necessary services are provided by the new infrastructure. If this is the case, and a correct mapping of the services required by the component to the new infrastructure is performed, we can reuse the component with same confidence in its reliability as in the original environment.
- One where the infrastructures are different and the infrastructure parts pertaining to the component are non-compliant. Formally if the $S(c) \setminus S_{new}(infrastructure) \neq \emptyset$.

In the latter case where the new infrastructure does not enable the same implementation of the design task model, a new mapping from the design must be performed. This mapping is a matter of implementing the new infrastructure model using the services provided in the new infrastructure. If this mapping is not possible then we cannot reuse the component at all. If the mapping is possible, we can still argue if this is reuse at all, since major parts of the component must be modified. Here we make a clear distinction between modify and parameterize, where modifications are changes in the source code and parameterizations leave the source code unchanged, but its behavior can be changed dynamically.

As an example, consider a component that has been proven to have certain reliability in an infrastructure that provides synchronization between tasks using time-wise separation. If now the component is reused in an infrastructure that handles synchronization using semaphores the synchronization strategy has to be changed. Consequently, we cannot assume anything about the reliability regarding synchronization failures. We have to assume weaker failure semantics since

the preconditions for which the reliability estimates regarding synchronization failures are no longer valid. That is, we cannot guarantee anything regarding synchronization failures in reusing the component.

The graph below illustrates the different reliabilities for the fault hypotheses (assumptions of failure semantics as introduced in section 2.3), 1 through 5 for component *c*.

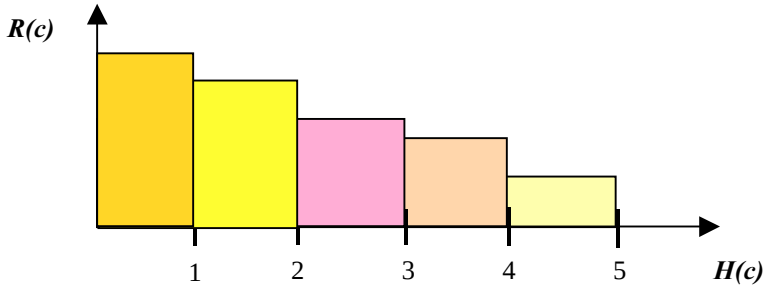

Figure 7 The reliability/fault hypothesis before reuse in a new infrastructure.

If the assumptions made for the reliability measure of fault hypothesis 3 is changed, the reliability is inherited from fault hypothesis 2 (see Figure 8), and we cannot say anything about the reliability regarding stronger failure semantics.

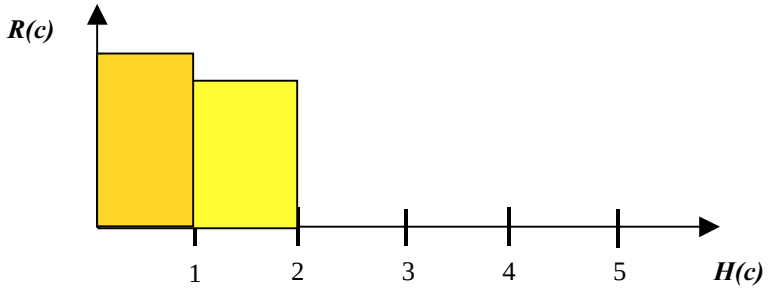

Figure 8 The new reliability/fault hypothesis after reuse in a new infrastructre.

Just as in the previous case where the infrastructure was not changed, the schedulability analysis must be performed all over again, ensuring that no temporal constraints are violated in the component as well in the rest of the system.

3.2.3 A measurement of reusability for components

For every new environment in which a component is successfully reused, the broader is the usability of that component. That is, it has been empirical proven that the component can be used in different environments, and is thus highly reusable. The graph in Figure 9 exemplifies this for changes in the period times of a component. The greater the reliability (denoted as *R),* the more confident we can be that the component will work correctly for that given period time. The greater the number of period times covered by the diagram, the more reusable the component is. However, one can argue that a component that has been reused in a lot of different environments but where every reuse resulted in a low reliability is really reusable? Consequently, the reusability is a combination of the number of environments where the component has been used, and the success of every such reuse.

Such a graph can be generated for every type of attribute in the component contract. For instance, the different types of real-time operating systems for which the component has been reused. In this case, the distribution is also a measurement of the portability for the component.

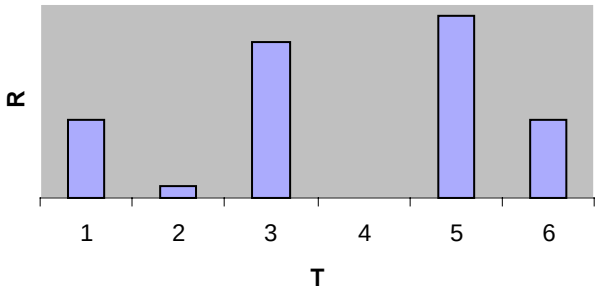

Figure 9 Distribution of period times for which the component has been reused

4 Conclusions

In this paper we have presented a novel framework for arguments of reuse of components in safety-critical real-time systems. In this framework, we formally describe component contracts in term of its temporal constraints given in the design phase (the design task model) and the temporal attributes available in the implementation (the infrastructure). Furthermore, the input-output domain for a component is specified in its contract. By relating the input-output domain, fault hypotheses, probabilistic reliability levels and the temporal behavior of the component, we can deem if a component can be reused or not. We can also deem how much and which subsets, of say input-output domains, that need additional functional and safety verification based on reliability requirements in the environment in which the reuse is intended.

5 References

[1] Alur R. and Dill D. A theory of timed automata, Theoretical Computer Science vol. 126 pp. 183-235, 1994

[2] Audsley N. C., Burns A., Davis R. I., Tindell K. W. *Fixed Priority Pre-emptive Scheduling: A Historical Perspective.* Real-Time Systems, The Int. Journal of Time-Critical Computing Systems, Vol. 8(2/3), March/May, 1995.

[3] Eriksson C., Mäki-Turja J., Post K., Gustafsson M., Gustafsson J., Sandström K., and Brorsson E. *An Overview of RTT: A design Framework for Real-Time Systems.* Journal of Parallel and Distributed Computing, vol. 36, pp. 66-80, Oct. 1996.

[4] H. Kopetz and J. Reisinger The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem. In Proceedings of he 14th Real-Time Systems Symposium, pp. 131-137, 1993.

[5] J. Chen and A. Burns Asynchronous Data Sharing in Muliprocessor Real-Time Systems Using Process Consensus. 10th Euromicro Workshop on Real-Time Systems, June 1998,

[6] Leveson N. G. *Safeware - System, Safety and Computers*. pp. 414. Addison Wesley 1995. ISBN 0-201-11972-2.

[7] Lui C. L. and Layland J. W.. *Scheduling Algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM 20(1), 1973.

[8] Parnas D.L., van Schouwen J., and Kwan S.P. *Evaluation of Safety-Critical Software*. Communication of the ACM, 6(33):636-648, June 1990.

[9] Poledna S. *Replica Determinism in Distributed Real-Time Systems: A Brief Survey*. Real-Times systems Journal, Kluwer A.P., (6):289-316, 1994.

[10] Powell D. *Failure Mode Assumptions and Assumption Coverage.* In Proc. 22nd International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, pp.386-395, July 1992.

[11] Puschner P. and Koza C. Calculating the maximum execution time of real-time programs. Journal of Real-time systems, Kluwer A.P., 1(2):159-176, September, 1989.

[12] Xu J. and Parnas D. *Scheduling processes with release times, deadlines, precedence, and exclusion, relations*. IEEE Trans. on Software Eng. vol. 16, pp. 360-369, 1990.