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).

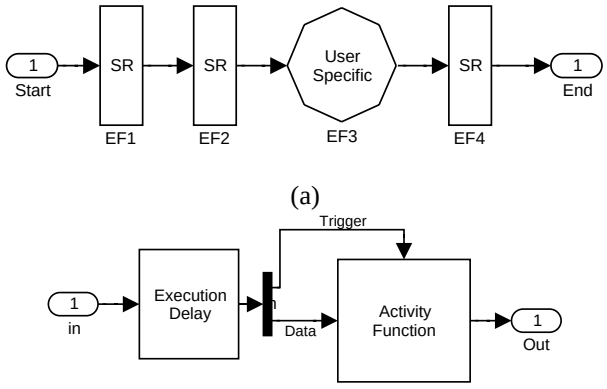


Figure 1. (a) The task model consisting of a sequence of elementary functions (EF). (b) The internal model of an EF.

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.

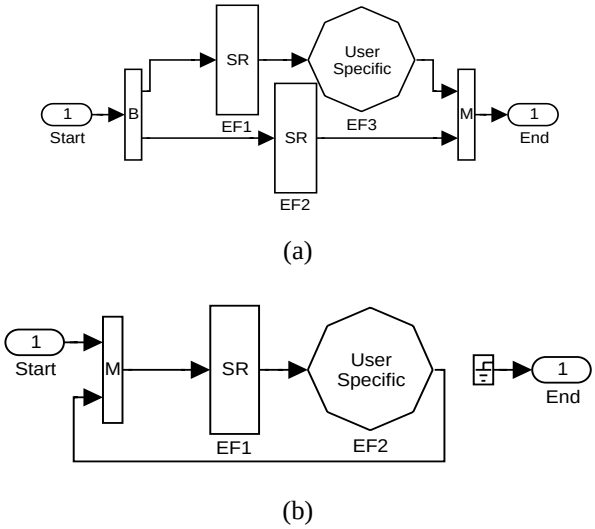


Figure 2. Examples of more complex task structures such as (a) branching or (b) looping.

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.

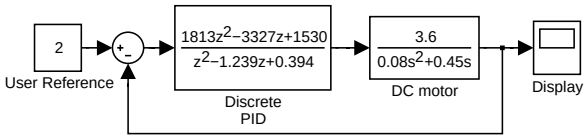
Parameters that may be of interest for timing analysis include a task's status, the times when particular events or activities within a task occur, or the time when a service request is serviced.

6. Illustrative Examples

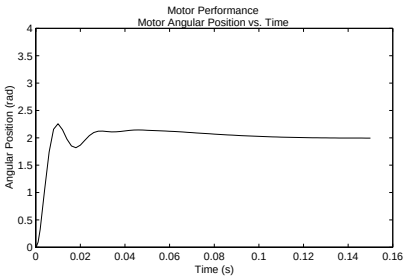
The aim of this simple example is to demonstrate how the tool can be used in architectural design for the evaluation of the computer implementation effects on the control performance. We will proceed from a pure functional design, to a single node and a distributed system implementation. In addition,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



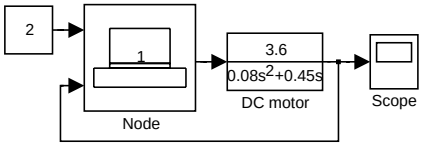
(b)

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.

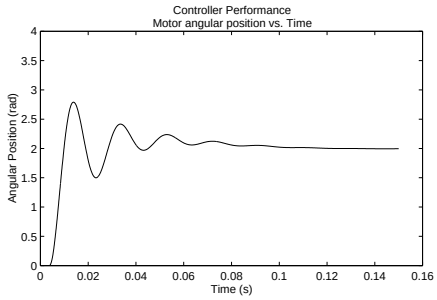
In this example, a continuous time PID controller was developed and then translated into a corresponding discrete-time controller. A sampling period of 2ms was chosen according to standard rules of thumb [24]. The angular position of the motor and the user reference value are measured by the controller. 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)

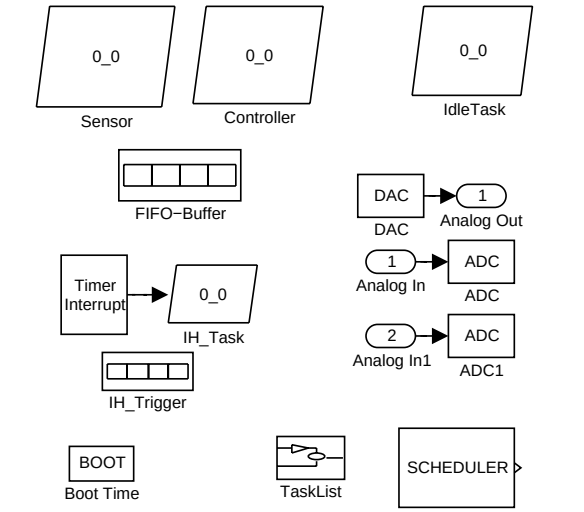


(b)

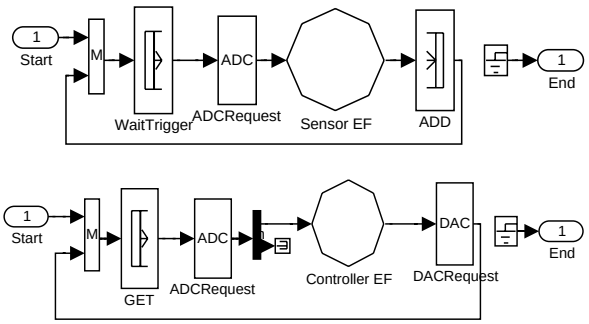
Figure 4. (a) The hardware model, showing the computer hardware and system environment. (b) Closed loop system performance where the computer-induced delays cause oscillatory behavior.

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.

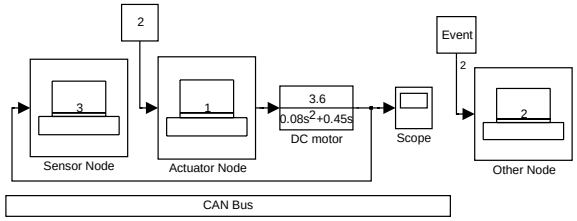
To build this model, the developer simply drags and drops the blocks from the Simulink library. The parameters of each block are then customised through a GUI. Partitioning is performed by copying the blocks from the original control model (e.g. "discrete PID" block in Figure 3a.) into the appropriate 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.

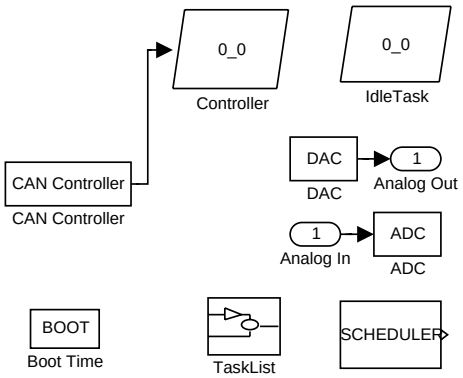
By instrumenting the model, various jitter and delay parameters were measured. For example, the motor actuation occurred 1.4ms after the sensor reading. This delay caused the change in the controller performance detected in Figure 4.

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)

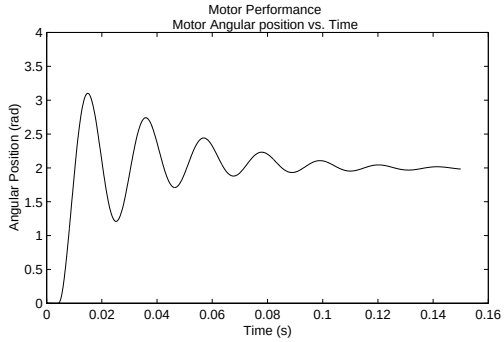


(b)

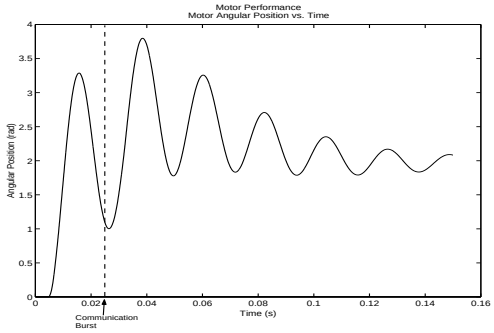
Figure 6. (a) Distributed hardware model. (b) Internal node structure for the "Actuator Node".

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.

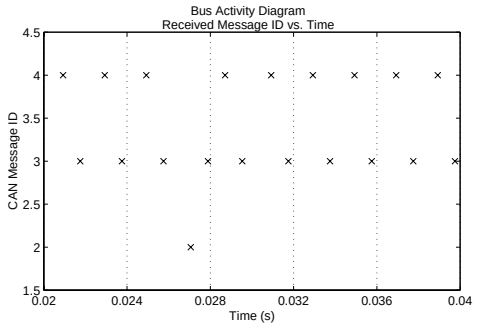


Figure 8. The arrival time of messages received by the CAN Controllers, (high priority interference message: ID=2, sensor readings: ID=3, other periodic messages: ID=4).

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.