Tools for Real-Time Control Systems Co-Design

A Survey

Henriksson, Dan; Redell, Ola; El-Khoury, Jad; Törngren, Martin; Årzén, Karl-Erik

2005

Document Version:
Publisher's PDF, also known as Version of record

ISSN 0280–5316
ISRN LUTFD2/TFRT--7612--SE

Tools for Real-Time Control Systems Co-Design — A Survey

Dan Henriksson, Ola Redell, Jad El-Khoury,
Martin Törngren, and Karl-Erik Årzén

Department of Automatic Control
Lund Institute of Technology
April 2005

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
April 2005

Document Number
ISRN LUTFD2/TFRT--7612--SE

Author(s)
Dan Henriksson, Ola Redell, Jad El-Khoury, Martin Törngren, and Karl-Erik Årzén

Supervisor

Sponsoring organisation
FLEXCON

Title and subtitle
Tools for Real-Time Control Systems Co-Design — A Survey

Abstract

This report presents a survey of current simulation tools in the area of integrated control and real-time systems design. Each tool is presented with a quick overview followed by a more detailed section describing comparative aspects of the tool. These aspects describe the context and purpose of the tool (scenarios, development stages, activities, and qualities/constraints being addressed) and the actual tool technology (tool architecture, inputs, outputs, modeling content, extensibility and availability).

The tools presented in the survey are the following; **Jitterbug** and **TrueTime** from the Department of Automatic Control at Lund University, Sweden, **AIDA** and **XILO** from the Department of Machine Design at the Royal Institute of Technology, Sweden, **Ptolemy II** from the Department of Electrical Engineering and Computer Sciences at Berkeley, California, **RTSIM** from the RETIS Laboratory, Pisa, Italy, and **Syndex** and **Orccad** from INRIA, France.

The survey also briefly describes some existing commercial tools related to the area of real-time control systems.

Key words
Simulation Tools, Real-time Control, Co-design

Classification system and/ or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
54

Recipient's notes

Security classification

Contents

1. Introduction 5

1.1 Motivation 5

1.2 A Historical Perspective 5

1.3 Tools Included 6

1.4 Organization of the Comparison and the Report 6

2. AIDA 8

2.1 Tool Overview 8

2.2 Comparative Aspects 9

3. Jitterbug 13

3.1 Tool Overview 13

3.2 Comparative Aspects 13

4. Orccad 17

4.1 Tool Overview 17

4.2 Comparative Aspects 18

5. Ptolemy II 22

5.1 Tool Overview 22

5.2 Comparative Aspects 24

6. RTSIM 26

6.1 Tool Overview 26

6.2 Comparative Aspects 29

7. Syndex 31

7.1 Tool Overview 31

7.2 Comparative Aspects 33

8. TrueTime 35

8.1 Tool Overview 35

8.2 Comparative Aspects 37

9. XILO 43

9.1 Tool Overview 43

9.2 Comparative Aspects 43

10. Other and Commercial Tools 49

11. Summary 51

12. References 51

1. Introduction

1.1 Motivation

Control systems are becoming increasingly complex from the perspectives of both control and computer science. Today, even seemingly simple embedded control systems often contain a multitasking real-time kernel and support networking. At the same time, the market demands that the cost of the system be kept at a minimum. Hence, many embedded control systems are subject to resource constraints, manifesting itself by limited CPU speed, memory, and network bandwidth of the target platform. In addition, a strong trend within industry today is to use commercially available information technology and commercial-off-the-shelf (COTS) components deeper and deeper in the real-time control systems.

Many computer-controlled systems are distributed systems consisting of computer nodes and a communication network connecting the various systems. It is not uncommon for the sensor, the actuator, and the control calculations to reside on different nodes in the system. One prominent example of this is modern automotive systems, which contain several embedded ECUs (electronic control units) used for various feedback control tasks, such as engine performance control, anti-lock braking, active stability control, exhaust emission reduction, and cruise control.

Within the individual nodes in the networked control loops, the controllers are often implemented as one or several tasks on a microprocessor with a real-time operating system. Often the microprocessor also contains tasks for other functions (e.g., communication and user interfaces). The operating system typically uses multiprogramming to multiplex the execution of the various tasks. The CPU time and the communication bandwidth can hence be viewed as shared resources for which the tasks compete.

Limited resources combined with non-optimized hardware and software components introduce nondeterminism in the real-time system. Digital control theory normally assumes equidistant sampling intervals and a negligible or constant control delay from sampling to actuation. However, this can seldom be achieved in practice in a resource-constrained system. For control systems this is of particular concern. Timing variations in sampling periods and latencies degrade the control performance and may in extreme cases lead to instability.

For optimal use of computing resources, the control algorithm and the control software designs need to be considered at the same time. For this reason, new, computer-based tools for real-time and control co-design are needed. The purpose of this survey is to summarize and compare the most prominent tools currently available for co-design of embedded real-time control systems.

1.2 A Historical Perspective

The need to support efficient digital implementation of control systems and co-design of control systems with the main implementation technology, real-time computer systems, became apparent during the 1970s and 1980s as microprocessor technology appeared and became more mature. Early efforts on real-time implementation environments and code generation can be found in the 1980s in the conferences Computer Aided Control Engineering (often called Computer Aided Control Systems Development, [Control Systems Society, 2004]).

As computer-aided engineering tools improved, it also became possible to develop tool support environments. Examples of relatively early efforts which in some way address real-time implementation of control systems include

- The Development Framework [Bass *et al.*, 1994], which combined and to some extent integrated control design (in Simulink) with software engineering capabilities using a CASE tool (Software through Pictures)
- The GRAPE tool-set [Lauwereins *et al.*, 1995], which although developed for digital signal processing systems, provides similar capabilities and supports distributed systems (allocation, scheduling, partitioning)
- Efforts by Honeywell labs including MetaH and the Parallel Scalable Design Tool-set (PSDT) [Vestal, 1994; Bhatt *et al.*, 1996]

In addition, in the real-time research community, a number of prototypical tools have been developed for schedule simulation, timing analysis and schedule generation, [Audsley *et al.*, 1994; Storch and Liu, 1996].

1.3 Tools Included

In order to limit the scope of this survey, we have chosen to focus on recent tools treating various aspects of real-time computer control systems. The following tools are included in the report

- **Jitterbug** and **TrueTime** from the Department of Automatic Control at Lund University, Sweden
- **AIDA** and **XILO** from the Department of Machine Design at the Royal Institute of Technology, Sweden
- **Ptolemy II** from the Department of Electrical Engineering and Computer Sciences at Berkeley, California
- **RTSIM** from the RETIS Laboratory, Pisa, Italy
- **Syndex** and **Orccad** from INRIA, France

All tools included in the report are suitable for co-design of real-time control systems. They, however, use different approaches and levels of abstractions. Some of the tools, such as **TrueTime** and **XILO**, are specifically tailored towards control and real-time co-design, whereas for others, such as **Ptolemy II**, the real-time control systems simulation is just one part of a larger framework. The abstraction level ranges from a very high level of abstraction of the distributed computer system in terms of time-varying delays, jitter in periods and transient faults, to detailed architectural models, as in **TrueTime** and **RTSIM**, that actually mimic the operation of for example an RTOS.

1.4 Organization of the Comparison and the Report

The remainder of the report is organized as follows. Sections 2-9 describe the different tools mentioned above. Each tool is presented by an introductory overview that describes the main use and intentions of the tool. Each tool is visualized by a simple example.

The overview is followed by a more detailed description of various aspects of the tool used for comparison. The comparative aspects are divided in two main areas; the context and purpose of the tool and the actual tool technology.

The context and purpose area treats the following aspects

- Which are the intended scenarios and development stages that the tool is supporting?
- Which specific activities are supported?
- Which qualities and constraints are addressed?
- Are there any special methodological considerations connected with the tool?

The tool technology area treats

- Description of the tool architecture
- Which inputs does the tool require
- Which outputs are generated
- Modeling content
- Tool automation
- Extensibility
- Availability

Section 10 describes a couple of commercial tools developed in industry. The report concludes with a summary of the survey in Section 11.

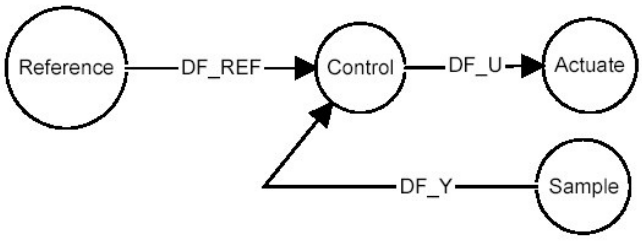

Figure 1 An example of an AIDA data flow diagram.

2. AIDA

2.1 Tool Overview

The Aida toolset [Redell *et al.*, 2004] is an environment for model-based design and analysis of real-time control systems. The most important feature of Aida is that it allows a user to take implementation effects into consideration when analyzing the performance of an automatic control system. Considered implementation effects include delays and time variations in the execution and scheduling of control functions and communication of data. The toolset also supports timing analysis of the real-time design such that an implemented solution can be shown to be schedulable and meet its timing constraints.

The toolset consists of a modelling environment, *Aidasign*, which interfaces with MATLAB/Simulink [The Mathworks, 2005], and a response time analysis tool, *Aidalyze*. In the toolset, a controller is designed using MATLAB/Simulink, which is an environment familiar to control engineers that supports simulation based analysis of control performance. The real-time system design starts with the translation of the Simulink model to a *data-flow diagram* (DFD) in Aidasign. The timing aspects of the controller, such as sampling periods and delays then constitute requirements on the real-time system design. The functions and communication flows specified in the data-flow diagram form the basis for all further modelling in Aida. Apart from being generated from Simulink models, data-flow diagrams can be specified completely or in parts within Aida. Figure 1 shows an example of a data-flow diagram with four functions and the related data flows connecting them.

Another fundamental model in Aida is the *hardware structure diagram* (HSD), where the hardware architecture, in terms of processors and their interconnections via communication links, is designed. In the HSD the functions and data flows in the associated data-flow diagram(s) are mapped to processors and communication links, respectively. Figure 2 shows an HSD with two processors (*P-1* and *P-2*) that are interconnected via a CAN-bus (*CAN-1*). The mapping of functions and data flows in Figure 1 is visualised. The utilisation (U) of each component is computed based on underlying models and the repository is used to temporarily store functions and data-flows that have not yet been mapped to any component.

Based on these two fundamental models and the mapping between them, a real-time implementation is designed. The design includes specification of operating system processes; their inter-communication in terms of messages; mapping of

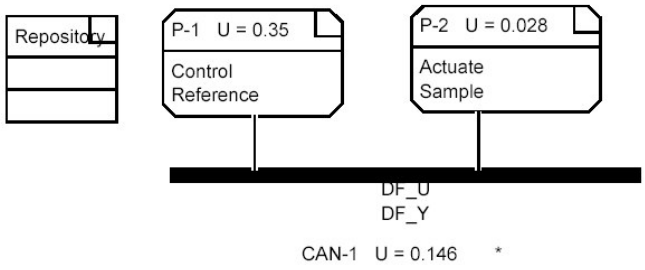

Figure 2 An example of an AIDA hardware structure diagram.

messages to CAN-frames; and triggering of process executions.

When the real-time system design has been completed, upper and lower bounds on the response times and release/response jitter (variations in release and response times) of the functions, processes and inter-process communications can be derived using the Aidalyze tool. These results can then be exported back to the control domain in the form of a Simulink model augmented with timing and execution order information. Hence, timing effects due to implementation can be incorporated in the control performance analysis through simulation of the generated Simulink diagram.

2.2 Comparative Aspects

Scenarios Supported AIDA is intended for one particular development scenario, but sub-scenarios can be followed as well. The major scenario starts in the control system design tool MATLAB/Simulink in which the data-flows in the control system are specified. The Simulink model is then imported to the Aida toolset in which a data-flow representation of the system is automatically generated. The data-flow model is augmented by the user with estimates of best- and worst-case execution times for functions and communication needs for the data-flows. The resulting model is the base for all other models in the tool-set.

Next, a real-time implementation of the control system is described using the models available in Aida. Given the model description of the implementation, a response time analysis is performed producing bounds on the response times of functions and processes. Finally, a transformed Simulink model can be generated, including delays according to the response time analysis results. The Simulink model can be used in simulation to test the control performance given the implementation induced delays.

Development Stages and Activities Supported The toolset can be used on different early stages in the development, but to make use of the complete scenario outlined above it should be used when the control system design is close to finalized and when the implementation of it is to begin. The hardware architecture could be fixed beforehand, or its design could also be guided by the results of Aida simulations. Hence, the toolset could be used to for example:

- compare and evaluate hardware architectures
- compare and evaluate software architectures
- compare and evaluate control system designs

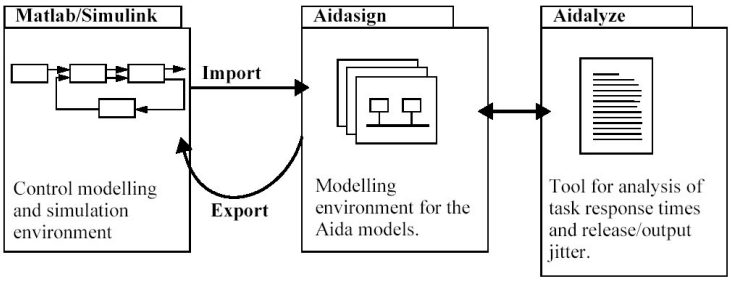

Figure 3 Architectural overview of the Aida toolset, highlighting the three parts: The interface with MATLAB/Simulink; the real-time system modelling environment (Aidasign); and the response time analysis tool (Aidalyze).

Qualities/Constraints Addressed As of today, the timing behaviour of an implementation is addressed through analysis on a real-time scheduling level. The qualities that are addressed include response time bounds and schedulability. Furthermore, using the generated control models augmented with timing information, the control system performance can be evaluated through simulation.

Methodological Considerations See Scenarios.

Tool Architecture The Aida toolset consists of two major parts, Aidasign for modelling of real-time implementations of control systems, and Aidalyze which is a stand-alone tool for response time analysis of distributed fixed-priority scheduled tasks that may be precedence related forming transactions.

Aida interfaces to MATLAB/Simulink, which enables import of control system models and export of the same models augmented with timing information. The interfacing activities are completely controlled from Aidasign. Figure 3 gives an overview of the tool set architecture.

Tool Inputs The user needs to provide estimates of worst (and possibly best) case execution times of the modelled functions, when executing on the modelled processors. Furthermore, the communication needs in each data flow (number of bytes) must to be specified.

In order to use the tool according to the intended scenario, a control system model made in Simulink is also needed. If such a model does not exist, the Aida toolset can be used to bound the response times of a system completely modelled within Aidasign. However, in that case no export to Simulink can be performed.

Tool Outputs Aidalyze produces bounds on the worst- and best-case response times of each function, process and CAN-frame in the system. The Aida tool computes the utilization of each processor and CAN-bus. If a Simulink model is imported to Aida, as a base for the implementation model, a Simulink model augmented with timing information can be generated as an output.

Modeling Content Apart from the modelling capabilities of Simulink, the Aida toolset includes the following models:

- *Data flow diagram* (DFD). Functions are specified and connected by data flow relations in the DFD. A function is parameterized by its minimum and

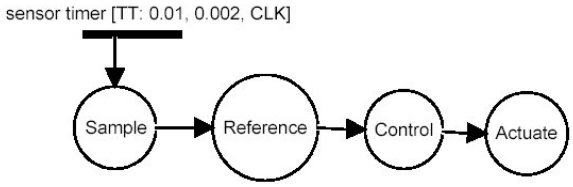

Figure 4 An example of an AIDA function timing and triggering diagram.

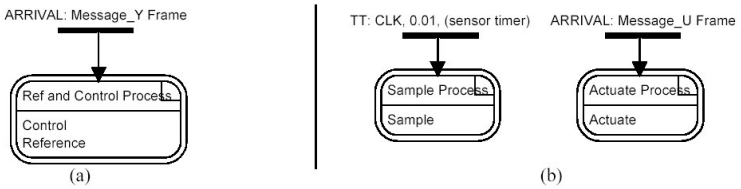

Figure 5 Process timing and triggering diagrams for processors P-1 (a) and P-2 (b) of Figure 2.

maximum execution times while a data flow is simply described with the number of bytes that it communicates. See Figure 1.

- *Function timing and triggering diagram* (FTTD). The FTTD is used to describe the sequences of precedence-related functions (the control flow) and the triggering of such sequences using periodic (time) or aperiodic (event) triggers. Figure 4 shows an FTTD where the execution sequence of the functions in the example application of Figure 1 have been specified. The diagram also shows a time trigger (TT) named *sensortimer* used to trigger the execution of the sample function. The parameters of the trigger are: the period (0.01); the admissible jitter (0.002); and the name of the source clock (CLK). The FTTD can be interpreted together with the DFD as a way to set the requirements on the implementation and does not directly specify any part of the implementation. FTTDs are therefore not necessary for a complete system description.

- *Hardware structure diagram* (HSD). As described above, the HSD is used to specify the hardware architecture as a network of processors interconnected by CAN buses. Processors are parameterized by a speed factor, used to scale the execution time of allocated functions. CAN buses are also associated with speed parameters, defining the communication speed on the bus. Furthermore, functions and data-flows are mapped to processors and buses in the HSD, as shown in Figure 2.

- *Process timing and triggering diagram* (PTTD). For each processor in an HSD there is a PTTD that describes the triggering of the contained processes' execution. Process execution may be triggered by a precedence relationship (completion of another process); by the arrival of a CAN frame; or by time or event triggers. The PTTD is also used to specify the processes by mapping functions in a processor to different processes. A process is assigned a fixed priority for scheduling. Figure 5 shows the PTTDs for the two processors in Figure 2. The execution of the sample process is triggered by a time trigger with period 0.01 while the other two processes are triggered by arriving CAN-frames.

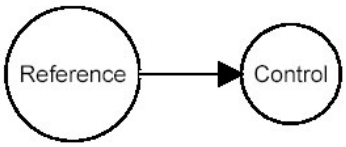

Figure 6 The process internal timing and triggering diagram of the Ref and Control process.

Figure 7 The process structure diagram for the example system.

- *Process internal timing and triggering diagram* (PiTTD). The PiTTD is used to define the execution sequence of functions within a process. It simply relates the functions included in a process in precedence order. Figure 6 shows the very simple PiTTD for the Ref and Control process executing in processor P-1.
- *Process structure diagram* (PSD). The PSD is basically an implementation version of the DFD. It defines how processes communicate via messages. The messages are composed of data flows that are communicated between functions in different processes. Many data flows may be included in a single message, if these data flows have the same sending and receiving processes. The PSD for the example application is shown in Figure 7. It defines two inter-process messages: `Message_U` and `Message_Y`.
- *Communication link diagram* (CLD). In the CLD the messages that were defined in the PSD are distributed on different CAN frames. One frame may include more than one message, but no more than 8 bytes in total. Figure 8 shows how the messages defined in Figure 7 are allocated to two different CAN frames. The arrivals of these frames are used to trigger the execution of the processes in the PTTDs in Figure 5.

Tool Automation The Simulink models are automatically transformed to Aida data-flow models when imported, and timing-augmented Simulink models are automatically generated from the Aida models.

The included tool Aidalyze may be used to perform response time analysis when an implementation model has been completely specified.

A consistency check, verifying the consistency of the information that is represented in multiple different Aida models, is performed when the user invokes an ”update”-function for either model in Aidasign.

Extensibility Aidasign is completely developed in the Domain Modelling Environment (DoME) from Honeywell. DoME is a tool for development of new modelling languages in which new models are easily added. Hence, Aidasign is easily extended with more models when needed. Furthermore, tools performing

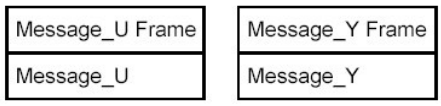

Figure 8 A communication link diagram defining the CAN-frames in the system.

automated tasks on the models, such as for example mapping of functions to processors, can easily be written in the Alter language which is an integral part of DoME.

Aidalyze is written in C++ for performance reasons. The source code is available and more algorithms for analysis can be added. Furthermore, other stand-alone tools written in other languages than Alter, can easily be added and their execution controlled from Aidasign.

Availability Developed in-house KTH. Available for anyone who asks for it.

3. Jitterbug

3.1 Tool Overview

Jitterbug [Cervin *et al.*, 2003; Lincoln and Cervin, 2002; Cervin and Lincoln, 2003] is a MATLAB-based *analysis* tool that makes it possible to compute a quadratic performance criterion for a linear control system under various timing conditions. The tool can also compute the spectral density of the signals in the system. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. The main contribution of the toolbox, which is built on well-known theory (LQG theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problems.

Jitterbug offers a collection of MATLAB routines that allow the user to build and analyze simple timing models of computer-controlled systems. A control system is built by connecting a number of continuous-time and discrete-time systems. For each subsystem, optional noise and cost specifications may be given. In the simplest case, the discrete-time systems are assumed to be updated in order during the control period. For each discrete system, a random delay (described by a discrete probability density function) can be specified that must elapse before the next system is updated. The total cost of the system (summed over all subsystems) is computed algebraically if the timing model system is periodic or iteratively if the timing model is aperiodic.

3.2 Comparative Aspects

Scenarios and Development Stages Supported Jitterbug is intended mainly as a research tool to evaluate different implementation strategies in terms of control performance. In that scenario a linear controller has been designed for a linear system and the tool will be used to evaluate how sensitive the closed-loop system is to various timing conditions imposed by the implementation.

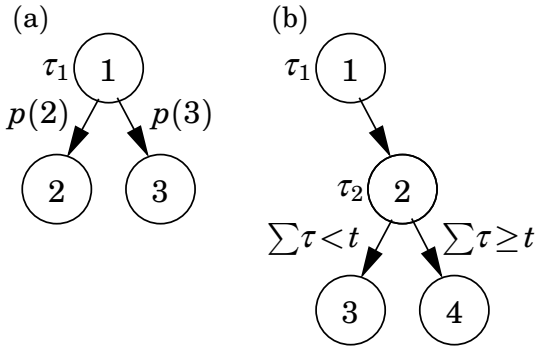

Figure 9 Alternative execution paths in a Jitterbug execution model: (a) random choice of path and (b) choice of path depending on the total delay from the first node.

Activities Supported Examples of timing conditions that may be evaluated include, e.g., how sensitive a control loop is to slow sampling and constant or random delays with jitter compensation. It is also possible to evaluate multi-rate controllers, overrun handling strategies, sensitivity to lost samples, and more.

Qualities/Constraints Addressed The main quality being addressed is control system performance (quantified by evaluating a quadratic cost function) under various timing conditions.

Methodological Considerations See above.

Tool Architecture Jitterbug consists of a collection of MATLAB functions that interface to the Control Systems Toolbox. These functions provide functionality to initialize Jitterbug, set up the timing and signal models that define a Jitterbug system, and to calculate the performance index.

Tool Inputs In Jitterbug, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period. Transitions between states in the timing model are performed depending on a chosen delay distribution.

The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. It is also possible to make the update equations depend on the time since the first node became active. This can be used to model jitter-compensating controllers for example.

For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In Jitterbug, two such cases can be modeled (see Fig. 9):

(a) A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with probability $p(i)$. This can be used to model a sample being lost with some probability.

(b) A vector n of next nodes can be specified with a time vector t. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model time-outs and various compensation schemes.

Tool Outputs A performance index that can be used for relative comparison between different scenarios. The performance criterion to be evaluated is specified as a quadratic, stationary cost function.

Modeling Content As mentioned above, Jitterbug can model most timing-related aspects of real-time control systems, such as constant and random delays, jitter in delays and sampling periods, and network issues such as lost samples.

However, to make the performance analysis feasible, Jitterbug can only handle a certain class of systems. The control system is built from linear systems driven by white noise, and the performance criterion to be evaluated is specified as a quadratic, stationary cost function. The timing delays in one period are assumed to be independent from the delays in the previous period. Also, the delay probability density functions are discretized using a time-grain that is common to the whole model.

Even though a quadratic cost function can hardly capture all aspects of a control loop, it can still be useful when one wants to quickly judge several possible controller implementations against each other. A higher value of the cost function typically indicates that the closed-loop system is less stable (i.e., more oscillatory), and an infinite cost means that the control loop is unstable. The cost function can easily be evaluated for a large set of design parameters and can be used as a basis in the control and real-time design.

As an illustration, an example of a Jitterbug model is shown in Figure 10, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

The Jitterbug commands used to define the control system of Figure 10 are given in Figure 11.

The process is modeled by the continuous-time system

$$G(s) = \frac{1000}{s(s+1)}.$$

and the controller is a discrete-time PD-controller implemented as

$$H_2(z) = -K\left(1 + \frac{T_d}{h}\frac{z-1}{z}\right),$$

The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1,$$

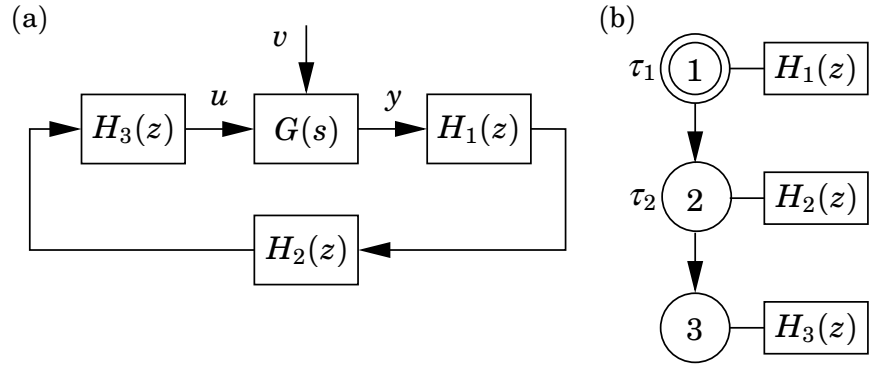

Figure 10 A simple Jitterbug model of a computer-controlled system: (a) signal model and (b) timing model. The process is described by the continuous-time system $G(s)$ and the controller is described by the three discrete-time systems $H_1(z)$, $H_2(z)$, and $H_3(z)$, representing the sampler, the control algorithm, and the actuator. The discrete systems are executed according to the periodic timing model.

```
G = 1000/(s*(s+1));                          Define the process
H1 = 1;                                      Define the sampler
H2 = -K*(1+Td/h*(z-1)/z);                    Define the controller
H3 = 1;                                      Define the actuator

Ptau1 = [ ... ];                 Define delay probability distribution 1
Ptau2 = [ ... ];                 Define delay probability distribution 2

N = initjitterbug(delta,h);                  Set time-grain and period
N = addtimingnode(N,1,Ptau1,2);              Define timing node 1
N = addtimingnode(N,2,Ptau2,3);              Define timing node 2
N = addtimingnode(N,3);                      Define timing node 3

N = addcontsys(N,1,G,4,Q,R1,R2);  Add plant, specify cost and noise
N = adddiscsys(N,2,H1,1,1);                  Add sampler to node 1
N = adddiscsys(N,3,H2,2,2);                  Add controller to node 2
N = adddiscsys(N,4,H3,3,3);                  Add actuator to node 3

N = calcdynamics(N);                         Calculate internal dynamics
J = calccost(N);                             Calculate the total cost
```

Figure 11 This MATLAB script shows the commands needed to compute the performance index of the control system defined by the timing and signal models in Figure 10.

The delays in the computer system are modeled by the two (possibly random) variables τ_1 and τ_2. The total delay from sampling to actuation is thus given by $\tau_{tot} = \tau_1 + \tau_2$.

Using the defined Jitterbug model it is straight-forward to investigate, e.g., how sensitive the control loop is to slow sampling and constant delays (by sweeping over suitable ranges for these parameters), and random delays with jitter compensation. For more details and other illustrative examples (including multi-rate control, overrun handling, and notch filter implementations), see [Cervin and Lincoln, 2003].

Tool Automation None.

Extensibility The use of Jitterbug assumes knowledge of sampling period and latency distributions. This information can be difficult to obtain without access to measurements from the true target system under implementation. Also, the analysis cannot capture all the details and nonlinearities (especially in the real-time scheduling) of the computer system. Therefore, the obvious extension of the analysis provided by Jitterbug is to resort to *simulation*. The rest of this report will describe current simulation tools for integrated control and real-time design.

Availability Jitterbug is available for download at

`http://www.control.lth.se/~lincoln/jitterbug/`

4. Orccad

4.1 Tool Overview

Orccad [Simon *et al.*, 1993; Simon *et al.*, 1999; Simon and Girault, 2001; Simon *et al.*, 1997] is a CAD system and approach aimed at the development of robotic systems from high-level specifications down to the implementation details. It deals with hybrid systems where continuous-time aspects relating to control laws, must be merged with discrete-time aspects related to control switches and exception handling. The approach taken by Orccad is based on the following considerations:

- A robotic application may be defined as a set of robot actions, the design of which needs expertise in several domains: knowledge in mechanics, control theory and computer science.
- Most actions performed by robots can be solved efficiently through control theory and the use of feedback control loops.
- The system needs to be accessible by users with different competence, from the end-user, who is mainly concerned with application specification and verification, to the control engineer, who is concerned with designing actions, to the computer scientist, who is concerned with implementation details.
- Real-time mechanisms for the execution of the final system need to be specified and verified since they influence the overall system performance.
- The object-oriented paradigm and code generation need to be used to improve software reliability and reusability.

The first step in designing a control application is to identify all the necessary elementary tasks involved. Then, for each of the tasks, issues from automatic control (such as defining the regulation problem, control law design, design of reactions to relevant events) and implementation (such as the decomposition of the control law into real-time tasks, and selection of timing parameters) aspects need to be considered. Finally, all the real-time tasks should be mapped on a target architecture. During this design, the control engineer has a lot of degrees of freedom to meet the end-user requirements and Orccad aims at allowing the

designer to exploit these degrees of freedom. Orccad promotes a controller architecture which is naturally ”open” since it allows access to every level by different users: the ”application” layer is accessed by the end-user, the ”control” layer is programmed by the control expert, and the ”system” layer is accessed by the system engineer.

Orccad provides formalised control structures, which are coordinated using the synchronous paradigm, specifically using the Esterel language (while the control laws are periodic and can be programmed using tasks and an RTOS, the discrete-event controller manages these control laws and handles exceptions and mode switching). The main entities used in the Orccad framework are:

- A *robot task* (RT), the elementary task representing basic robotic actions where the control aspects are predominant.
- A *module task* (MT), a real-time task.
- A *robot procedure* (RP), a hierarchical composition of RTs and other existing RPs, forming more complex structures.

The RT characterizes continuous-time closed-loop control laws, along with their temporal features and the management of associated events. From the application perspective, the RT's set of signals and associated behaviours represent the external view of the RT, hiding all specification and implementation details of the control laws. More complex actions, the RPs, can then be composed from RTs and other RPs in a hierarchical fashion leading to structures of increasing complexity. RPs can be used to fulfil a single basic goal through several potential solutions, or to fulfil a full mission specification.

The Orccad methodology is bottom-up, starting from the design of control laws by control engineers, to the design of more complex missions.

4.2 Comparative Aspects

Development Stages and Activities Supported Orccad can be used during the early architectural design stages of robotics mission functionality, followed by detailed design of the software implementing these functions. Both structural and behavioural design activities are supported.

Qualities/Constraints Addressed Orccad is targeted towards hybrid (continuous-time control with modes of operation) robotic activities implemented on a computer system. Certain constraints are assumed:

- System functionality is assumed periodic.
- Communication is limited to 8 predefined protocols.
- Control activities can be performed using control loops.

Methodological Considerations and Scenarios Supported Orccad suggests a bottom-up approach starting with specifications and followed by implementation details and more complex missions:

- The design starts from the end-user specification.

- The control engineer develops control laws in continuous-time that realises the specified action, in the form of block diagrams where elementary algorithmic modules are connected through input/output ports.
- Implementation aspects are taken into account by associating temporal properties to the modules (called module tasks) constituting the control law.
- The module tasks are distributed on a multiprocessor system architecture.
- Simulation and formal verification can be performed for validation.

Tool Architecture The Orccad toolset consists of a human-machine interface (RP Editor) for model specification. It also contains code generators and tools for the specification of Esterel programs at the application level.

A SIMPARC simulator is utilised for two types of system simulations: the first one takes into account discretization aspects, while the second validates a multiprocessor implementation level.

The *FC2TOOLS* tools for the synchronous language Esterel allow the formal verification of the system behaviour as well as its crucial properties, such as liveness and safety properties.

Tool Inputs System descriptions from specification down to implementation details are made through a specific human-machine interface.

Tool Outputs Final C code of the system is generated after the code generation stage. In addition, analysis results from the formal verification as well as simulations can be obtained.

Modeling Content System functionality is described through

- *Robot tasks* which describe elementary robotic control actions
- *Robot procedures* describing more complex robotic actions or a complete robotic application

The software is described through

- *Module tasks* for real-time tasks implementing parts of a robot task
- *Observers* checking conditions and generating events
- *Signals* used to synchronise the operations between robot tasks and robot procedures

The following example is extracted from "The ArmX Example" given at the Orccad homepage,
`http://www.inrialpes.fr/iramr/pub/Orccad/ExempleArmX/frame-eng.html`

The example shows how to design, validate, and execute a robotic application through the simulation of a two degrees of freedom arm.

The designed application is a target-following task. When the target is in the robot workspace, the end-effector follows the target and when it is out of the

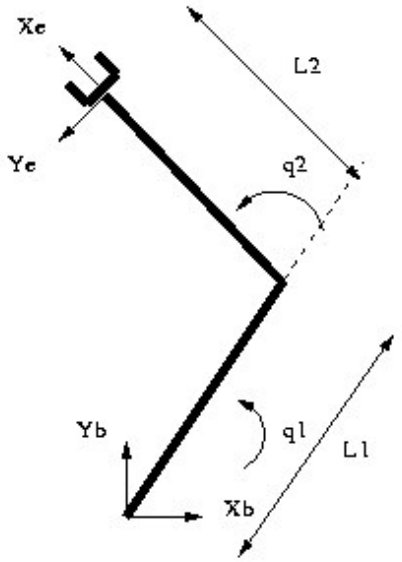

Figure 12 The robotic application: a two degrees of freedom arm.

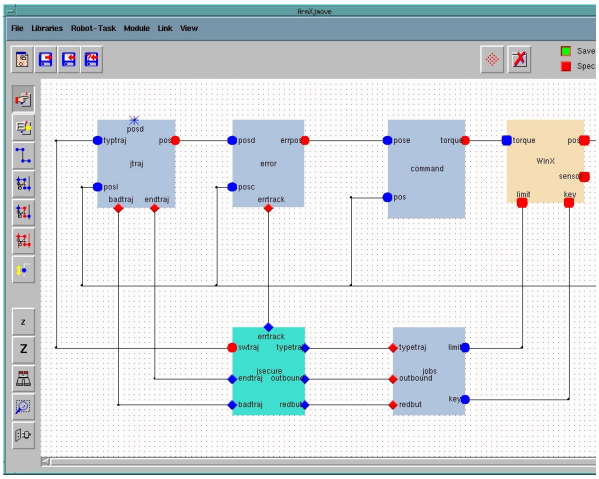

Figure 13 The *ArmXjmove* robot task.

robot workspace the manipulator points at this target. This application must be safe and therefore it is performed taking into account exceptions like too high tracking error, joint limits being reached, or required reconfiguration of the arm. The two-link manipulator with rotational joints is shown in Figure 12.

In this application, the designer identified three control laws. These three control laws will be embedded in three robot tasks:

- *ArmXjmove* : assumes movement in the joint space of the manipulator. (Further detailed below)
- *ArmXcmove* : assumes movement in the Cartesian space of the manipulator.
- *ArmXfmove* : assumes pointing at the target when it is out of the workspace of the manipulator.

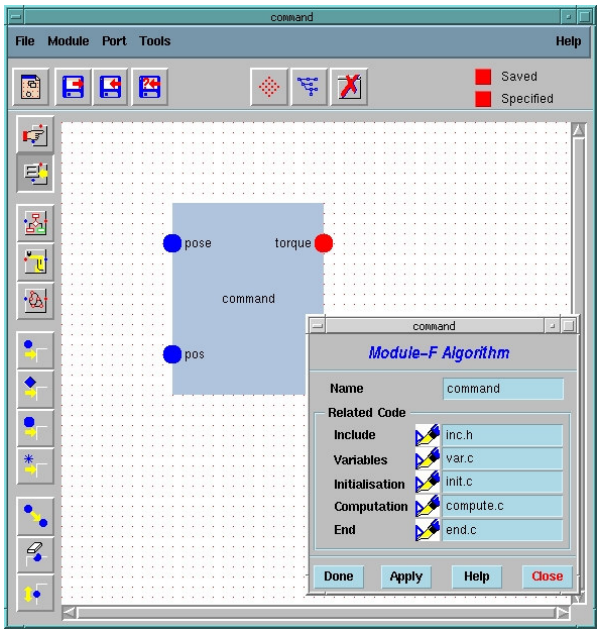

Figure 14 The *command* module task.

Considering *ArmXjmove* as an example, the events which locally control this robot task are:

- *typetraj* : Exception T1 to suspend the motion
- *outbound* : Exception T3 when joint limits are reached
- *redbut* : Exception T3 of emergency stop when the key 'q' is pressed on keyboard
- *badtraj* : Exception T3 when the parameter posd is out of bound
- *errtrack* : Exception T3 when the joint error is too high
- *endtraj* : Post-condition when the current position reached *posd*

The robot task is decomposed with algorithmical modules:

- *command*: to compute the torque with a proportional corrector with gravitational compensation,
- *error*: to compute the joint error
- *jtraj*: to compute a joint trajectory from current position to desired position *posd*
- *jobs*: observers to generate events from observation of the robot (limit) and its environment (key)

Modules are the elementary entities to construct robot tasks. The design of a robot task is achieved by connecting modules that exchange data through typed ports. For this application we must construct:

- The module *WinX* of *Physical Resource class* to specify an interface between robot tasks and the simulator.

- One module of *robot task Automaton class* to control the robot behavior locally.
- Modules of the *Algorithm class* are used to specify the algorithms necessary to compute the control law. Some modules are reused in the three robot tasks like *command* and *error*. Each piece of code of computation is encapsulated in these entities.

Each robot task must be independently tested by using a robot procedure. The user can then write the robot procedure to perform the final application *Appli-ArmX*. The application is specified in Maestro which directly generates Esterel code. The application consists of a loop sequence starting with the manipulator moving a joint (*ArmXjmove*) to a certain position. When this action is performed a sequence of two actions of pointing task (*ArmXfmove*) when the target is out of the workspace and a Cartesian movement when the target is in the workspace (*ArmXcmove*). The Cartesian move space should be preempted by a move joint position when the exception T2 *reconf* occurs.

Using the panel of Verification, you could, for example, use the criterion robot task Level to verify if the nominal specification is correct. You could see the correspondence with the textual Maestro specification and the automaton visualised.

Through the use of the last panel of Execution, the user is able to produce the code, compile and execute the application. In the panel Trace, the user can put spies. A simulation driver simulates the dynamics of the two-link manipulator. The simulation is animated through a X11 window. This window is interactive and the user can use a keyboard to give information to the robot, initialize it, put torque, get joint position, move a target (a white square) with the mouse and so on.

Tool Automation The automata of the robot tasks and robot procedures are automatically translated into Esterel code, which is then further translated into C code.

Extensibility Not supported.

Availability A single tool implementation exists by the approach developers (currently not available).

5. Ptolemy II

5.1 Tool Overview

Ptolemy II is the third generation of software produced within the Ptolemy project [Hylands *et al.*, 2003; Ptolemy Project, 2004] at the University of California at Berkeley. Ptolemy II supports *heterogeneous, hierarchical* modeling, simulation, and design of concurrent systems, especially embedded systems. The focus is on complex systems mixing various technologies and operations.

Simulation models are constructed under *models of computation* that govern the interaction of the components in the model. Different models of computation are

used for modeling different types of systems. The abstraction provided by the model of computation also simplifies code generation from the Ptolemy models.

Ptolemy is component-based and models are constructed by connecting a set of components and have them interact under the model of computation. Components in Ptolemy are called *actors*.

An important feature of Ptolemy is its focus on heterogeneous, hierarchical modeling, meaning that each system may be composed of a number of subsystems at different levels where each subsystem can have its own model of computation. This makes it easier to deal with complexity.

Ptolemy is Java-based and provides graphical user interfaces for model construction and result visualization. The visual editor framework of Ptolemy is called Vergil, and an example model is shown in Figure 15.

Actor-based Design Most models of computation in Ptolemy support actor-oriented design (one exception is finite state machines). Each actor has an interface that restricts its interaction with other actors. This interface includes ports and parameters. Ports are used for communication, whereas parameters are used to configure the actor. Actors primarily interact by sending messages through channels according to some messaging system. The concepts of models, actors, ports, parameters, and channels describe the abstract syntax of actor-based design and is often represented graphically as in Figure 15.

Models of Computation Ptolemy provides a wide variety of models of computation that deal with concurrency and time in different ways. Some of the most important include:

- **Continuous Time (CT)** – used to model physical systems with linear or nonlinear differential equation descriptions. The CT model is designed to operate with other domains, like for example the FSM domain to form hybrid models or the TM model for real-time control.
- **Discrete-Event (DE)** – used to model digital hardware (e.g. network communication) and to simulate telecommunications systems.
- **Finite-State Machines (FSM)** – here entities represent states instead of actors and connections represent transitions.
- **Giotto** – time-triggered domain with periodically triggered actors. Intended for hard real-time systems.

Timed Multi-tasking The timed multitasking (TM) model of computation [Liu and Lee, 2003] is intended to support deterministic design of concurrent real-time software. It assumes an underlying priority-driven preemptive scheduler. In TM each actor executes as a concurrent task with a fixed execution time and deadline. Actors are activated by triggering conditions (periodically for controller tasks) and outputs are delayed until the task has been active (has had access to the virtual CPU) for a time equal to its execution time.

However, the TM model provides deterministic input-output latency of actors by always delaying outputs to the deadline of the actor. This is called faster-than-real-time computing. This way the effects of scheduling on delay and jitter is suppressed, while on the same time an often unnecessary delay is introduced

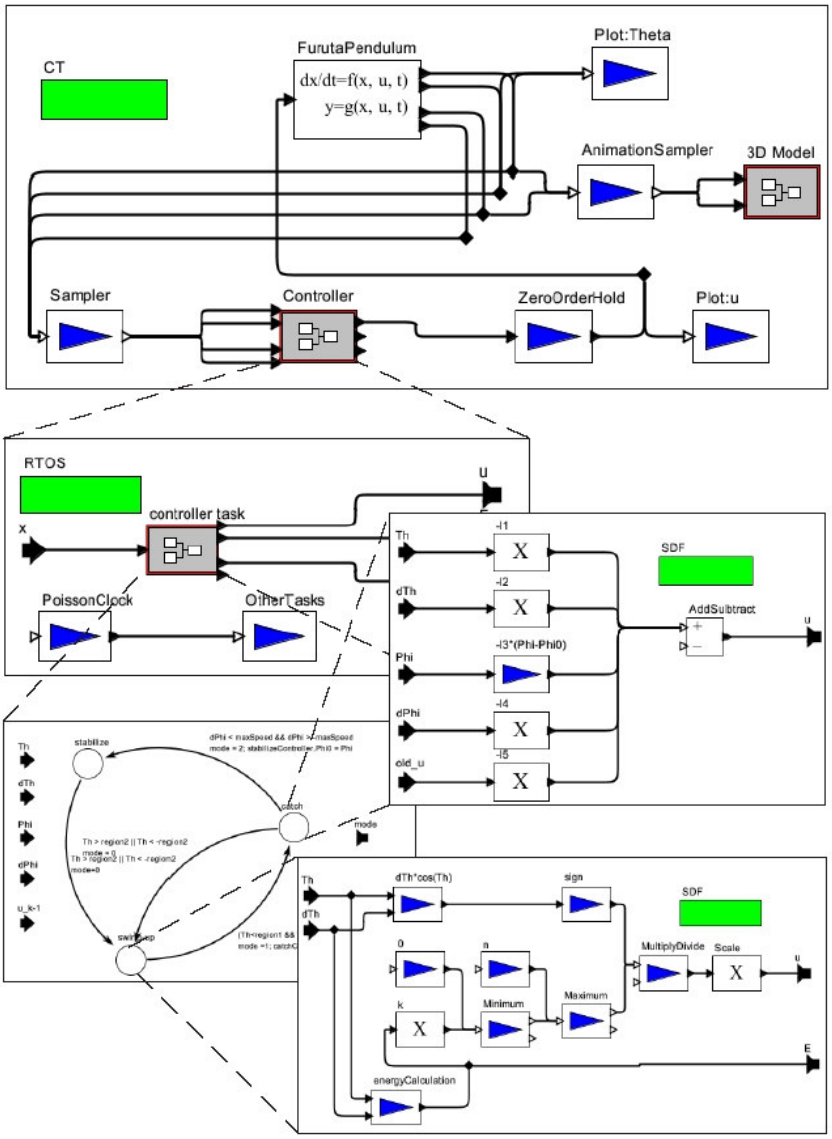

Figure 15 An integrated simulation model of an inverted pendulum process in Ptolemy II (from [Liu *et al.*, 2002]). The top level contains actors for the pendulum process and the controller and utilizes the continuous-time model of computation. The controller is implemented as a task in the TM domain (here called RTOS). In addition to that, the different states of the controller are modeled as synchronous data flows (SDF).

that reduces the performance for control tasks. The TM model supports deadline handling to deal with the fact that the execution has not finished by the task deadline. This is mainly intended to preserve the timing determinism of other actors.

5.2 Comparative Aspects

Scenarios Supported Ptolemy is directed towards modeling, simulation (executable models), and design of embedded system software. It emphasizes methodologies for defining and producing embedded software together with the systems in which the software is embedded. Ptolemy aims at covering a large area of

scenarios by use of its hierarchical, heterogeneous modeling framework. Each subsystem may have its own model of computation, different from the systems at other levels in the hierarchy.

More specifically, the timed multitasking model of computation is to be used (together with, e.g., the continuous-time and discrete-event models) for integrated design of real-time control systems. Here the performance of the real-time system (scheduling mechanisms and communication protocols) may be analyzed and evaluated against the applications performance.

Development Stages Supported As indicated by the simulation scenario described above, the main aim of Ptolemy is to provide a complete modeling and design framework which is intended to facilitate the use of Ptolemy throughout the development process, from early conceptual models to implementation and verification.

Activities Supported The supported activities depend mainly on the model of computation chosen. Within the timed multitasking model, it is possible to do scheduling analysis, change software architecture, do code generation and hardware-in-the-loop simulation. Adding discrete-event models, it is possible to simulate network protocols and distributed control systems.

Qualities/Constraints Addressed The timed multitasking model considers concurrent tasks (actors), each characterized by trigger conditions, computation times, and deadlines. Task execution is started by the trigger conditions and outputs are not produced until the actor has have access to the CPU for a time specified by its computation time. Overrun handling is available if the task exceeds its deadline. CPU access is granted based on the actor priority within the simulated real-time scheduling scheme.

However, outputs are not produced until the task deadline even if they are computed earlier. This has the benefit of guaranteeing a constant and known input-output latency, but many applications exist for which this design choice is undesirable. Since all task outputs are delayed one period, the effects of the real-time scheduling are of less importance, and jitter, delay, and compensation schemes can, consequently, not be simulated.

Methodological Considerations See above.

Tool Architecture Ptolemy is written in Java, and highly modularized. The architecture consists of two sets of packages; one that provides generic support for all models of computation, and one that provides more specialized support for particular models of computation. The latter includes *domains* which are Java packages that implement particular models of computation.

The packages structure is divided in core packages, UI packages, and library packages. The core packages support abstract syntax and semantics of Ptolemy. The UI packages contain support for the XML file format and the visual interface for graphical model construction, called Vergil. The library packages provide domain polymorphic actor libraries, i.e., actors that can operate in a variety of domains. See [Hylands *et al.*, 2003] for a more detailed architecture description.

Tool Inputs The simulation model is defined graphically by connecting actors in a fashion similar to Simulink. The inputs for the timed multitasking model include trigger conditions, deadlines, execution times, and priorities of the various tasks. Priorities can be automatically computed using schedulability analysis for the given task parameters.

Tool Outputs Relevant outputs can be found on different levels of the simulation hierarchy. Within the TM model it is possible to see the activations of the various tasks, and within, e.g., the CT model it is possible to obtain time domain plots of the physical processes being controlled.

Modeling Content Ptolemy is a large modeling and design framework for embedded system design. However, the support for integrated real-time control system design is quite limited due to the restrictions imposed by the timed multitasking model of computation. It only facilitates simulation of fixed priority scheduling of tasks with constant execution times. Also, input-output latencies are forced to be constant and well-known.

Tool Automation Ptolemy contains many library objects that simplify the building of models. This includes actors for continuous processes and real-time tasks. However, no support for automatic model generation is provided.

Extensibility Being developed in Java and because of its high modular properties, it is, in theory, straight-forward to extend the Ptolemy libraries with new actors and also new or modified models of computation.

Availability Ptolemy II 4.0 is available for download at

`http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII4.0/index.htm`

6. RTSIM

6.1 Tool Overview

RTSIM [Palopoli *et al.*, 2002; Lipari, 2003b] is a tool that is aimed at simulating real-time embedded control systems. The main goal is to facilitate co-simulation of real-time controllers and controlled plants in order to evaluate the timing properties of the architecture in terms of control performance.

The tool consists of a collection of C++ libraries and uses the mathematical library OCTAVE [Eaton, 1998] for the continuous plant simulation. The libraries allow the user to specify; a set of plants, the functional controller behavior, the implementation architecture, and a mapping of functional behavior onto the architectural components. The simulation model is constructed based on this separation between functional behavior and the HW/SW architecture, see Figure 16.

The functional design involves controller operations such as extracting sensor data and computing control signals. It also produces timing constraints based on the closed-loop dynamics. The architectural design involves specifying a model of entities such as software tasks, schedulers and network protocols. The functional design is mapped onto the architectural design and the timing constraints are translated into real-time constraints.

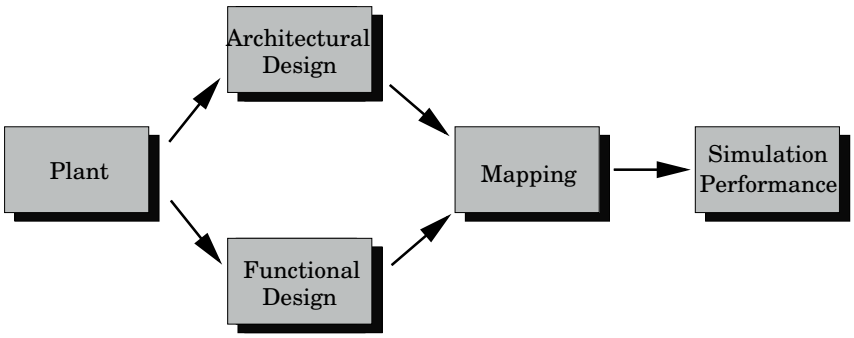

Figure 16 The design of a real-time control simulation using RTSIM.

The simulation produces results related both to the real-time performance and the control performance. This includes the generation of execution traces, real-time statistics (e.g., delays and jitter), and control performance metrics such as time responses and quadratic costs.

Functional Behavior RTSIM exploits a data flow approach for the functional modeling based on two types of functional abstractions; the *computing unit* and the *storage unit*. Figure 17 shows an example of functional model of an inverted pendulum control system.

Each computing unit has a number of input and output ports that must be connected to storage units. The requirement on the computing units is furthermore that they should be able to respond to three different external commands; *read*, *execute*, and *write*. The execute command can implement an arbitrary control algorithm and the computing units may also have internal states. Pre-defined computing unit library objects are provided for many existing controller structures.

Storage units are of three types; *input buffers*, *memory buffers*, and *output buffers*. Input and output buffers model I/O between computing units and the environment and can be thought of as sensors and actuators, respectively. Memory buffers are used for communication between different computing units. No assumptions are made in the functional model regarding hardware implementations of the I/O or how to deal with concurrent access requests.

System Architecture In the architectural model, a *task* is a finite or infinite sequence of *jobs* (requests for execution). Each job implements some functional behavior and may be periodic, sporadic or aperiodic. The jobs execute a sequence of instructions, each modeled by a constant or stochastic execution time and associated with a *read*, *execute*, or *write* operation of a computing unit.

Tasks are assigned to *nodes*, each consisting of one or more processors and a real-time kernel. The kernel is assigned a scheduling policy and a synchronization protocol. The state of the art scheduling algorithms as well as many *aperiodic server* schemes developed in Pisa are provided by the tool.

The system may also be built up as a number of nodes connected by network links, where the nodes communicate using real-time messages over a physical link using a certain access protocol.

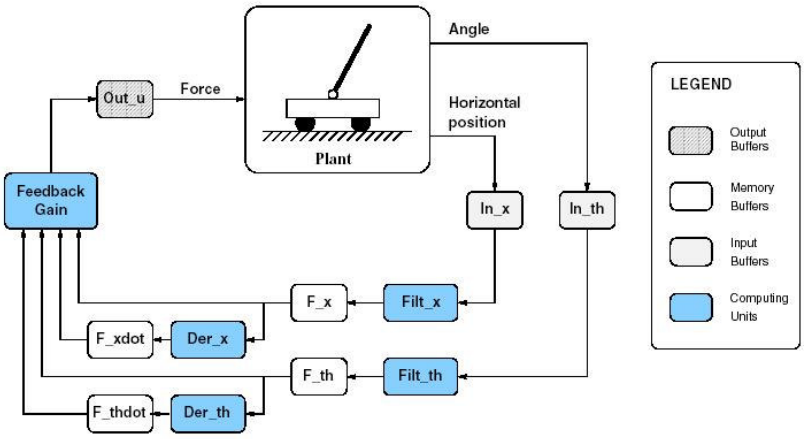

Figure 17 Example taken from [Palopoli *et al.*, 2002] of a functional design for an inverted pendulum system. Input buffers are used to model sensors and output buffers to model actuators. Computing units exist for filtering and derivative actions and to compute the control signal.

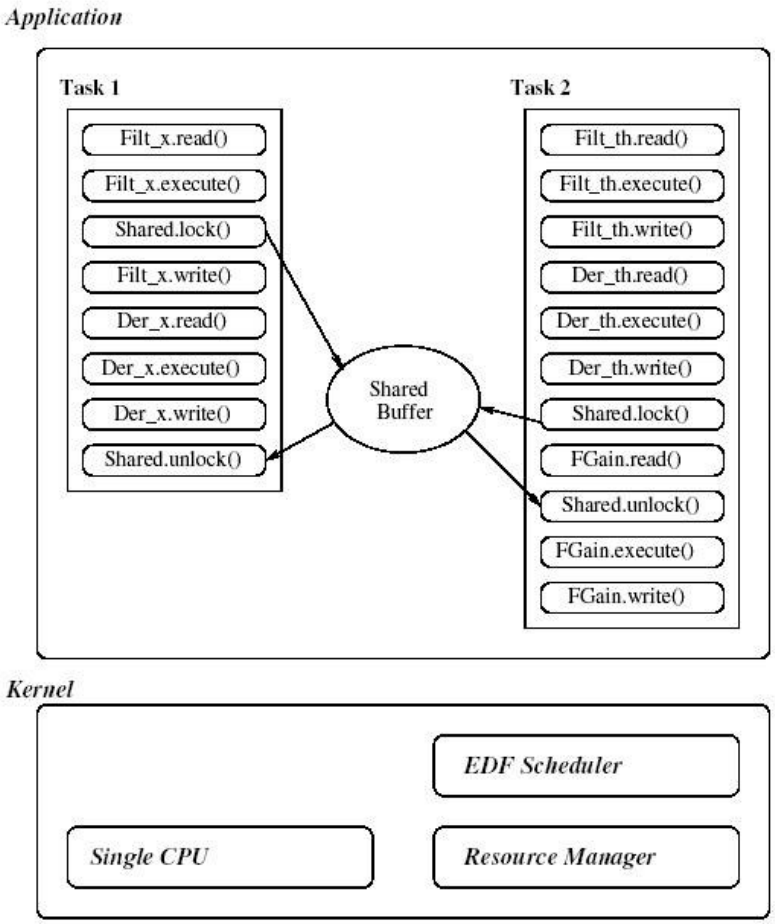

Figure 18 Example of an architectural design for the system from Figure 17. Here it is assumed that the horizontal position of the cart is obtained from camera images, whereas a potentiometer is used to acquire the angle. Therefore, the architecture uses two tasks for the controller computations.

Performance Evaluation A RTSIM simulation is based on *events* (e.g., task arrivals, task terminations and task deadlines). The events are associated with situations in the architectural model and will subsequently trigger actions in the functional model.

All events of a simulation may be recorded in a trace file and displayed using the Java-based utility RTTracer provided in the RTSIM distribution. This is used for classical real-time schedulability analysis of the simulation in terms of task activations and deadline misses. It is also possible to use statistical probes to measure, e.g., jitter and delay distributions over multiple runs.

Finally, for control performance evaluation, special buffers may be used to record time responses of certain plant variables and to compute quadratic performance indices.

6.2 Comparative Aspects

Scenarios Supported The main scenario intended to be supported is integrated real-time control system design. The functional behavior of the system is the result of classical control system design for the continuous-time plant based on the specifications of the closed-loop performance. The architectural model can be developed in complete separation and involves specifying hardware and software components of the implementation.

The functional model is then mapped onto the architectural model and the integrated system can be simulated. Based on the simulation results it is then possible to iteratively update the functional and/or architectural models to obtain the results that best fits the requirements of the project.

Development Stages Supported As indicated by the simulation scenario described above, the tool can be used at any time of the development process as long as an functional and architectural model of the control system exist. This can be anytime from early development to the verification stage. The tool is, however, mainly used as a research tool to evaluate novel scheduling algorithms in terms of both real-time and control performance.

Activities Supported RTSIM supports simulation of various hardware and software aspects when implementing real-time control systems. This involves real-time task scheduling, synchronization protocols and network communication. All these aspects may be evaluated against the control performance of the physical plant under control.

Qualities/Constraints Addressed The RTSIM tool addresses various types of evaluation qualities. The tool was initially a pure real-time scheduling tool (without support for continuous-time dynamics simulation) and advanced scheduling schemes may be simulated and evaluated in terms of pure timing behavior. It contains, already implemented, most of the scheduling algorithms developed at Retis Lab as well as many other state of the art scheduling schemes.

However, using the OCTAVE library for physical plant modeling the evaluation can be taken one step further. Consequently, the main quality being addressed is that of the control performance as a result of the complete functional/architectural model. This can be quantified either in terms of time re-

sponses such as the overshoot or rise times or using quadratic performance metrics. However, the plant modeling is still limited and lacks the graphical features of, e.g., Simulink.

Methodological Considerations See above.

Tool Architecture RTSIM consists of a collection of C++ libraries that contain three types of objects, namely *continuous-time plants*, *functional components*, and *architectural components*.

The main package of RTSIM is RTLIB that is used to describe the architectural components. This is based on the MetaSim [Lipari, 2003a] library for simulation of discrete event systems. RTLIB may be used on its own (for real-time simulation) or together with CTRLIB for complete real-time control systems simulation.

RTLIB models architectural entities such as real-time tasks, scheduling algorithms, single- and multi-processor nodes, and network links. These will be described in some more detail below.

CTRLIB provides a hierarchy of classes that implement various computing and storage units.

Tool Inputs Apart from providing the functional and architectural models the user needs to provide a number of parameters for the simulation. This includes relative and absolute deadlines of tasks, task periods, and instruction execution times for individual jobs. Depending on the scheduling algorithm a number of associated parameters can be set and changed between simulations. This includes, e.g., bandwidth assignments between tasks when using the Constant Bandwidth Server.

Tool Outputs The simulation generates execution traces and statistical timing measures of jitter and latencies. It also returns quantities related to the control performance, such as time responses and quadratic performance metrics.

Modeling Content In terms of scheduling the RTSIM tool is very general. It contains library object for many existing policies and provides support for easy modeling of schemes not provided in the libraries. It supports both single and multiprocessor scheduling.

RTSIM also supports many existing synchronization protocols to avoid priority inversion. Again, defining and implementing your own protocol is straightforward.

The network support, however, is quite limited and in the current version only Ethernet and CAN bus networks are provided. The main drawback of the tool lies in its plant modeling environment that lacks the graphical drag-and-drop features of Simulink. This also limits the possibilities to analyze the simulation results on a more detailed level.

Tool Automation RTSIM contains library objects for standard control algorithms (computing units), scheduling algorithms, and synchronization protocols. This facilitates the construction of the functional and architectural models of the system. However, no support for automatic generation of these models is provided.

Extensibility Being developed in C++, the RTSIM libraries should be easily extensible and modular. For example, it should be straightforward to use other numerical packages for the plant modeling as well as adding more scheduling schemes, synchronization protocols, or network protocols.

Availability RTSIM is available for download at

`http://rtsim.sssup.it/`

7. Syndex

7.1 Tool Overview

The Syndex tool supports rapid prototyping of reactive data-driven algorithms implemented on distributed heterogeneous hardware architectures [Pernet and Sorel, 2003; Grandpierre *et al.*, 1999; Lavarenne *et al.*, 1991; Forget *et al.*, 2004]. Syndex lets the user specify both the algorithm and the distributed hardware in a graphical environment, and then automates the mapping and scheduling of functions (called operations) and communications (called transfers) on the processors (operators) and communication buses (transformators). During the mapping and scheduling process, the hardware architecture can be refined to better match the algorithm needs. When a sufficiently good solution has been found, Syndex generates executable code that can be downloaded to the target hardware.

Algorithms are specified in Syndex as conditioned data-flow graphs that are indefinitely repeated. The graphs are conditioned because there may be branches that are only executed given that some condition is satisfied. The graph describes data-dependency relations between operations and form a directed acyclic graph. An operation can be hierarchically decomposed into sub-operations. The algorithm model has formalized semantics equivalent to the synchronous language SIGNAL and can therefore be verified with tools for this purpose.

Figure 19 shows a description of a very simple algorithm, *algobasic*, that contains two constant blocks (*cste1* and *cste2*) that represent constant integers. The constants are fed into two operation blocks (*add* and *mul*) that perform operations on their inputs and produces output that is forwarded to either of the two actuator blocks (*visuadd* and *visumul*).

The hardware architecture specification consists of components interconnected via edges representing communication media. A component may be either an operator, which executes operations, or a transformator, which sequences data transfers between communication media. Figure 20 shows an example architecture, *biProc*, that consists of two processors (*root* and *pc1*) interconnected via a TCP transformator (*link*).

The automated step supported by Syndex (referred to as adequation) is performed by an heuristic algorithm that both maps operations and data transfers to operators and transformators, and schedules the operations and transfers on their respective components. The scheduling is an off-line ordering of operations and transfers, assumed to be indivisible in their execution/transmission. Whether or not the adequation algorithm can handle multiple algorithms with

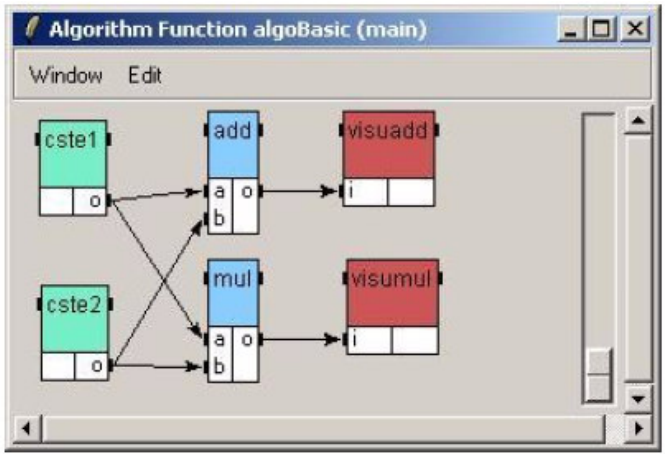

Figure 19 The algorithm graph *algoBasic*.

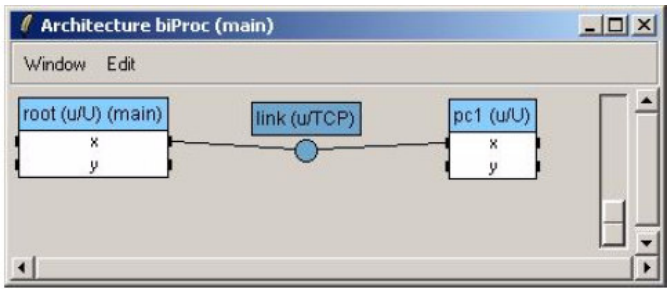

Figure 20 The architecture graph *biProc*.

different periods (a multi-rate system) and map and schedule these successfully on a common hardware architecture, is not clear. The result of the adequation algorithm, called an implementation model, is visualized in a timing diagram that shows the parallel execution and transmission on all components in the system. Figure 21 shows a timing diagram of the schedule generated by the adequation algorithm when the operation *add* has been constrained to execute on the *root* operator and the *visuadd* operation has been constrained to execute on the *pc1* operator. These constraints where included to force some communication via the TCP transformator in the simple example. Note how the constants do not show in the timing diagram since they do not need any execution.

Given the implementation model, Syndex is able to automatically generate a distributed executive for the algorithm. The executive is built from a library of architecture-dependent primitives that compose an execution kernel. One such

Figure 21 Timing diagram generated by the adequation algorithm.

kernel is needed for each supported processor type.

7.2 Comparative Aspects

Scenarios Supported Syndex is intended to be used for rapid prototyping of computational algorithms such as control and signal processing algorithms. A graphical interface is used to formally specify the algorithm and the distributed hardware. Then, when the automated mapping and scheduling has been performed, the user has the possibility to refine the hardware algorithm descriptions to make better use of resources and reduce cost. The algorithm can also be formally verified using other tools for that purpose. The automated mapping and scheduling is performed after each refinement and when the implementation has converged to a satisfactory result, executable code can be generated.

Development Stages and Activities Supported The toolset is intended to be used in early stages when the hardware architecture has not yet been finally selected. The tool gives good support for comparing different hardware architectures for the implementation of a given algorithm. Due to the rapid prototyping functionalities, the tool is also valuable for the implementation of early test systems in which different algorithms can be implemented and compared.

Qualities/Constraints Addressed The automated mapping and scheduling step is mainly focused on finding a solution that optimizes the usage of available resources, subject to the timing constraints of the algorithm that have been specified by the user. The user may also specify constraints on the mapping of operations and data transfers to components.

Methodological Considerations The methodology supported by Syndex is called A3 - Algorithm Architecture Adequation - and it follows the steps outlined above. These include: specification of the algorithm in a formal synchronous graphical language; specification of the heterogeneous target hardware architecture; and automated specification of an implementation of the algorithm on the architecture, using the adequation step which involves spatial mapping and scheduling in time. Finally, an executable prototype may be generated and executed on the target hardware.

Tool Architecture The tool uses a graphical interface in which algorithms and architectures are described. Different types of objects can be specified and instantiated directly as locally defined operations, operators etc. It is also possible to use and instantiate pre-defined types from libraries, including types for, e.g., mathematical operations and TCP communication links. The adequation (mapping and scheduling) step is performed in the same environment.

When an implementation has been fully specified, different code generators can be used on the model. The Syndex code generator generates a file with Syndex code that fully describes the model, which can be used to later reload the model into the graphical interface. A postscript code generator can be used to produce more detailed textual information than is shown in the graphical interface. Finally, an executive code generator produces code that is needed to execute the algorithm on the modelled target hardware.

Tool Inputs The toolset needs the designer to describe the algorithm in terms of a directed acyclic graph containing operations that are to be executed, and data-transfers to be transmitted between the operations. The designer also needs to describe the target hardware including the processors and the interconnecting communication links. Each operation and data-transfer is given a duration (execution/transmission time) for each operator or communication medium available in the system.

Tool Outputs The tool derives a mapping of operations and data-transfers as well as an off-line schedule of these. Furthermore, executable code can be generated given that specific macros have been developed for the included processors.

Modeling Content The algorithm is described as a directed acyclic graph that is executed repeatedly with a given period. The operations in the graph have input and output ports that are typed and can represent integers, floats or boolean variables, or arrays thereof. The ports are connected to corresponding ports of other operations in the graph. There are two types of edges between operations: a strong data communication and execution precedence; or execution precedence only. Operations in a data flow graph can be hierarchically decomposed into sub-graphs. One operation may have many parallel sub-graphs and the selection of the sub-graph to execute for any given invocation, is controlled by a conditioning dependence of the operation. Hence a data flow graph can conditionally execute different branches on different invocations. Furthermore, each operation and data-transfer is associated with one duration parameter for each possible operator or communication medium in the system.

Syndex also allows algorithm specification in SyncCharts [Pernet and Sorel, 2003], a state diagram language that is similar to Statecharts but with a stronger semantics compliant with the deterministic real-time scheduling of Syndex. A SyncChart diagram can be included in an algorithm by first translating it (automatically) to a Syndex data flow representation.

The architecture model is a non-directed graph of operators and communication media interconnected by edges describing the topology of the architecture. The communication media may be e.g. Ethernet, CAN or RS232 and the operators may be micro-controllers, DSPs or FPGAs of various types.

Tool Automation The tool automates the mapping and scheduling of the algorithm to the specified hardware. It also able to generate code for various types of processors.

Extensibility The code generation can be extended to support more operator (processors and communication media) types through the inclusion of more macro-executive source files.

Availability Syndex is available for download at

`http://www-rocq.inria.fr/syndex`

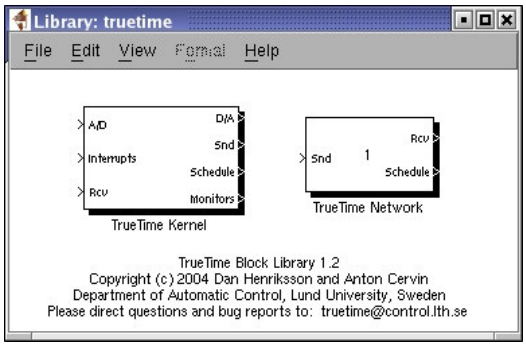

Figure 22 The TrueTime block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

8. TrueTime

8.1 Tool Overview

TrueTime [Cervin *et al.*, 2003; Henriksson *et al.*, 2003; Henriksson and Cervin, 2003; Henriksson *et al.*, 2002b] is a MATLAB/Simulink-based tool that facilitates simulation of the temporal behavior of a multitasking real-time kernel executing controller tasks. The tasks are controlling processes that are modeled as ordinary continuous-time Simulink blocks. TrueTime also makes it possible to simulate models of standard MAC layer network protocols, and their influence on networked control loops.

In TrueTime, kernel and network Simulink blocks are introduced, the interfaces of which are shown in Figure 22. The kernel blocks are event-driven and execute code that models, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policy of the individual kernel blocks is arbitrary and decided by the user. Likewise, in the network, messages are sent and received according to the chosen network model.

The level of simulation detail is also chosen by the user—it is often neither necessary nor desirable to simulate code execution on instruction level or network transmissions on bit level. TrueTime allows the execution time of tasks and the transmission times of messages to be modeled as constant, random, or data-dependent. Furthermore, TrueTime allows simulation of context switching and task synchronization using events or monitors.

In addition to the block library in Figure 22, TrueTime provides a collection of C++ functions with corresponding MATLAB MEX-interfaces. Some functions are used to configure the simulation by creating tasks, interrupt handlers, monitors, timers, etc. The remaining functions are real-time primitives that are called from the task code during execution. These include functions for A/D-D/A conversion, changing task attributes, entering and leaving monitors, sending and receiving network messages, and more.

TrueTime is configured in a C++ or MATLAB m-file, called an initialization script. Likewise, task and interrupt handler code is defined by C++ functions or MATLAB m-files according to a pre-specified format. The possibility for graphical modeling has been avoided to make the tool more general and more connected to the real implementation code.

The Kernel Block The kernel block is a MATLAB S-function that simulates a computer with a simple but flexible real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels. The kernel executes user-defined tasks and interrupt handlers. Internally, the kernel maintains several data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for tasks, interrupt handlers, monitors and timers that have been created for the simulation.

An arbitrary number of tasks can be created to run in the TrueTime kernel. Tasks may also be created dynamically as the simulation progresses. Tasks are used to simulate both periodic activities, such as controller and I/O tasks, and aperiodic activities, such as communication tasks and event-driven controllers. Aperiodic tasks are executed by the creation of task instances (jobs).

Each task is characterized by a number of static (e.g., relative deadline, period, and priority) and dynamic (e.g., absolute deadline and release time) attributes. In accordance with the Real-Time Specification for Java (RTSJ) [Bollella *et al.*, 2000], it is furthermore possible to attach two overrun handlers to each task: a deadline overrun handler (triggered if the task misses its deadline) and an execution time overrun handler (triggered if the task executes longer than its worst-case execution time).

Interrupts may be generated in two ways: externally (associated with the external interrupt channel of the kernel block) or internally (triggered by user-defined timers). When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt.

The execution of tasks and interrupt handlers is defined by user-written code functions. These functions can be written either in C++ (for speed) or as MATLAB m-files (for ease of use). Control algorithms may also be defined graphically using ordinary discrete Simulink block diagrams.

Simulated execution occurs at three distinct priority levels: the interrupt level (highest priority), the kernel level, and the task level (lowest priority). The execution may be preemptive or non-preemptive; this can be specified individually for each task and interrupt handler.

At the interrupt level, interrupt handlers are scheduled according to fixed priorities. At the task level, dynamic-priority scheduling may be used. At each scheduling point, the priority of a task is given by a user-defined priority function, which is a function of the task attributes. This makes it easy to simulate different scheduling policies. For instance, a priority function that returns a priority number implies fixed-priority scheduling, whereas a priority function that returns the absolute deadline implies earliest-deadline-first scheduling. Predefined priority functions exist for rate-monotonic, deadline-monotonic, fixed-priority, and earliest-deadline-first scheduling.

The Network Block The network block is event-driven and executes when messages enter or leave the network. When a node tries to transmit a message, a triggering signal is sent to the network block on the corresponding input channel. When the simulated transmission of the message is finished, the network block sends a new triggering signal on the outport channel corresponding to the receiving node. The transmitted message is put in a buffer at the receiving computer node.

A message contains information about the sending and the receiving computer node, arbitrary user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

The network block simulates medium access and packet transmission in a local area network. Six simple models of networks are currently supported: CSMA/CD (e.g. Ethernet), CSMA/AMP (e.g. CAN), Round Robin (e.g. Token Bus), FDMA, TDMA (e.g. TTP), and Switched Ethernet. The propagation delay is ignored, since it is typically very small in a local area network. Only packet-level simulation is supported, i.e., it is assumed that higher protocol levels in the kernel nodes have divided long messages into packets.

Configuring the network block involves specifying a number of general parameters, such as transmission rate, network model, and probability for packet loss. Protocol-specific parameters that need to be supplied include, e.g., the time slot and cyclic schedule in the case of TDMA.

8.2 Comparative Aspects

Scenarios and Development Stages Supported The main use of TrueTime is for simultaneous simulation of all aspects of distributed real-time control applications. By co-simulation of continuous process dynamics, task execution in real-time kernels, and network communication, it is possible to evaluate the performance of control loops subject to the constraints of the target system.

In a typical scenario, a controller design has been performed (without considering implementation constraints) and is about to be implemented on the target system. In this scenario, TrueTime can be used to evaluate different real-time implementations, and the effects of CPU and network scheduling, task attributes, etc, on the control performance.

For a given implementation architecture, TrueTime may also be used to obtain temporal statistics that can be used as constraints in the design of the controller.

In the optimal scenario, however, the controller and architectural designs are performed at the same time. Here, TrueTime provides a convenient framework for integrated control and real-time design.

TrueTime is also used as an experimental platform for research on flexible approaches to real-time implementation and scheduling of controller tasks. One example is feedback scheduling [Cervin *et al.*, 2002; Henriksson *et al.*, 2002a] where feedback is used in the real-time system to dynamically distribute resources according to the current situation in the system.

TrueTime may be used in all stages of the development process, from the early stages and system specifications, during the actual system construction, and finally for testing and validation.

Activities Supported TrueTime makes it possible to simulate the temporal behavior of the computer architecture (e.g., scheduling policies and network protocols) and its effect on the control performance. Standard scheduling policies may be used, e.g., priority-based preemptive scheduling and earliest-deadline-first scheduling, but it is also straight-forward to define arbitrary user-defined

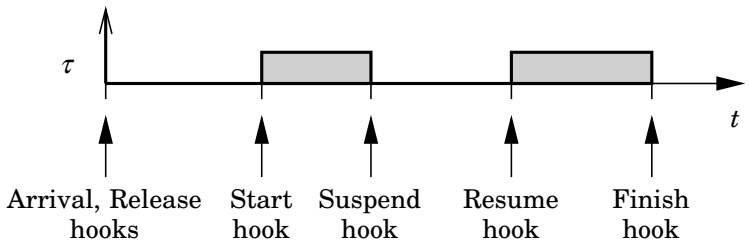

Figure 23 TrueTime scheduling hooks.

policies. Task overrun strategies may be evaluated and easily implemented using the TrueTime overrun handlers.

TrueTime can also be used as an experimental platform for research on co-design of control algorithms and computer resource scheduling mechanism. It is possible to study dynamic compensation schemes that adjust the controller on-line based on measurements of actual timing variations, i.e., treat the temporal uncertainty as a disturbance and manage it with feed-forward or gain scheduling. It is also easy to implement new more flexible approaches to dynamic scheduling, e.g., feedback scheduling [Cervin *et al.*, 2002] of CPU time and communication bandwidth and quality-of-service (QoS) based scheduling, in the TrueTime CPU kernel.

TrueTime may also be used only as a scheduling simulator, without being connected to any continuous-time processes. This can be used to get information of the timing of the real-time system, and various scheduling policies can be evaluated in terms of deadline misses and response times.

Qualities/Constraints Addressed Being developed in Simulink, TrueTime allows for traditional control system assessment in terms of performance, stability and robustness. Compared to normal control system development in Simulink, TrueTime also considers the constraints imposed by the implementation platform.

Methodological Considerations See above.

Tool Architecture TrueTime is primarily intended to be used together with MATLAB/Simulink. However, the TrueTime kernel actually implements a complete event-based kernel and Simulink is only used to interface the kernel and the tasks with the continuous-time processes.

TrueTime is written in C++ and consists of two Simulink S-functions for the kernel and network block, and a collection of C++ functions for the initialization commands and real-time primitives. All TrueTime objects, such as tasks, interrupt handlers, monitors, timers, and events, are defined by C++ classes. These classes as well as the real-time primitives may easily be extended by the user to add more functionality.

The Simulink engine is used only for timing and interfacing with the rest of the model (the continuous dynamics). Since it is written in C++, it should thus be easy to port the block code to other simulation environments, provided these environments support event detection (zero-crossing detection).

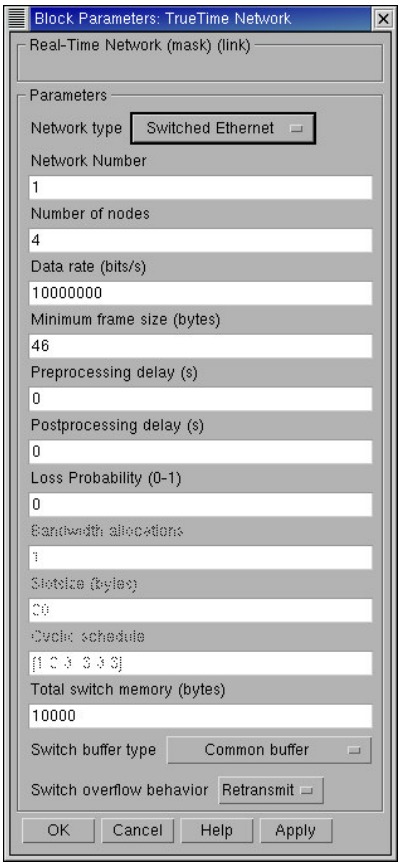

Figure 24 The dialog of the TrueTime Network block.

Tool Inputs TrueTime is initialized in a script for each kernel block (node). In this script, the user specifies the scheduling policy of the kernel, creates tasks and assigns task attributes (period, priority, deadlines, etc), and creates any other objects for the simulation (interrupt handlers, timers, monitors, mailboxes, etc). The execution of each task and handler is defined by a code function (see Modeling Content below) with constant or random execution time. It is also possible to specify a simulated time associated with context switches.

Furthermore, to facilitate arbitrary dynamic scheduling mechanisms, it is possible to attach small pieces of code (*hooks*) to each task. These hooks are executed at different stages during the simulation, as shown in Figure 23.

The network block is configured through the block mask dialog, see Figure 24. The following network parameters are common to all models; number of nodes in the network, data rate (bits/s), minimum frame size (bytes), pre- and post-processing delay, and loss probability. Protocol-specific attributes include slot sizes for TDMA, and buffer size and buffer type for switched Ethernet.

Tool Outputs Depending on the simulation a number of different output graphs are generated by the TrueTime blocks. Each kernel block will produce two graphs; a computer schedule and a monitor graph, and the network block will produce a network schedule. The computer schedule will display the execution trace of each task and interrupt handler during the course of the simulation. If context switching is simulated, the graph will also display the execution of the kernel.

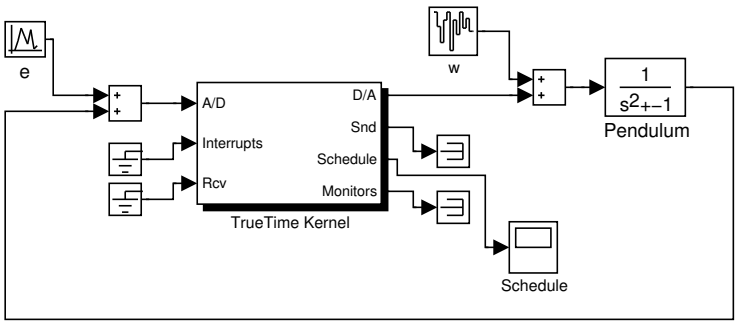

Figure 25 A TrueTime computer block connected to a continuous pendulum process.

There will be one execution trace for each task and handler. If the signal is high this means that the task is running. A medium signal indicates that the task is ready but not running (preempted), whereas a low signal means that the task is idle. In an analogous way the network schedule shows the transmission of messages over the network, with the states representing sending (high), waiting (medium), and idle (low). The monitor graph shows which tasks that have been holding the different monitors during the simulation.

It is also possible to create logs for each tasks. These will log arbitrary task attributes, such as response times and latencies, during the simulation and write them to the MATLAB workspace after the simulation.

Plant and controller outputs are conveniently displayed and evaluated using the Simulink built-in outputs. It is also possible to dynamically evaluate for example quadratic performance functions, within Simulink.

Modeling Content The TrueTime blocks are connected with ordinary Simulink blocks to form a real-time control system, see Figure 25.

Before a simulation can be run it is necessary to initialize the individual kernel blocks. Initialization of a TrueTime kernel block involves specifying the number of inputs and outputs of the block, defining the scheduling policy, and creating tasks, interrupt handlers, events, monitors, etc for the simulation. This is done in an initialization script for each kernel block.

The initialization code in Listing 1 shows the minimum of initialization needed for a TrueTime simulation (e.g., corresponding to the simple simulation model in Figure 25). The kernel is initialized by providing the number of inputs and outputs and the scheduling policy using the function `ttInitKernel`. A periodic task is then created by the function `ttCreatePeriodicTask`. The execution of the task is given by the code function `Pcontroller`, described below.

The execution of tasks and interrupt handlers is defined by code functions. A code function is further divided into code segments according to the execution model in Figure 26. The code can interact with other tasks and with the environment at the beginning of each code segment. This execution model makes it possible to model input-output latencies, blocking when accessing shared resources, etc. The number of segments can be chosen to simulate an arbitrary time granularity of the code execution. Technically it would, e.g., be possible to simulate very fine-grained details occurring at the machine instruction level, such as race

Listing 1 Example of a simple TrueTime initialization function.

```
function example_init

ttInitKernel(2, 1, 'prioFP');

name   = 'ctrl';
offset = 0;
period = 0.005;
prio   = 2;
data.u = 0;
data.K = 2;

ttCreatePeriodicTask(name, offset, period, prio, 'Pcontroller', data);
```

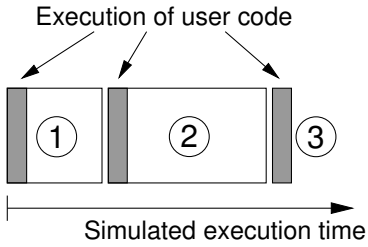

Figure 26 The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

conditions. However, that would require a large number of code segments.

The simulated execution time of each segment is returned by the code function, and can be modeled as constant, random, or even data-dependent. The kernel keeps track of the current segment and calls the code functions with the proper argument during the simulation. Execution resumes in the next segment when the task has been running for the time associated with the previous segment. This means that preemption by higher-priority activities and interrupts may cause the actual delay between execution of segments to be longer than the execution time.

Listing 2 shows an example of a code function corresponding to the time line in Figure 26. The function implements a standard P-controller. In the first segment, the plant is sampled and the control signal is computed. In the second segment, the control signal is actuated and the controller states are updated. The third segment indicates the end of execution by returning a negative execution time.

The data structure `data` represents the local memory of the task and is used to store the control signal and measured variable between calls to the different segments. A/D and D/A conversion is performed using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Note that the input-output latency of this controller will be *at least* 2 ms (i.e., the execution time of the first segment). However, if there is preemption from

Listing 2 Example of a standard code function written in MATLAB code. The local memory of the controller task is represented by the data structure `data`. This stores the controller gain and the control signal between invocations of different code segments.

```
function [exectime, data] = Pcontroller(segment, data)
switch segment,
  case 1,
    r = ttAnalogIn(1);
    y = ttAnalogIn(2);
    data.u = data.K*(r-y);
    exectime = 0.001;
  case 2,
    ttAnalogOut(1, data.u);
    exectime = 0.001;
  case 3,
    exectime = -1; % finished
end
```

other high-priority tasks, the actual input-output latency will be longer.

TrueTime interrupt handlers is used to model code that is executed in response to interrupts. Interrupt handlers are scheduled with fixed priorities on a higher priority level than tasks. Interrupt handlers may be associated with timers, the network receive channel, external interrupt channels, or attached to tasks as overrun handlers. Timers can be one-shot or periodic.

TrueTime monitors are used to provide mutual exclusion and synchronization between tasks. Tasks waiting for monitor access are sorted according to their priority under the given scheduling policy. Standard priority inheritance is implemented as resource access policy. TrueTime events may be free or associated with monitors as condition variables. The event waiting queues are also priority-sorted.

Tool Automation MATLAB scripts can be used to run sequences of simulations with different input parameters. Other than that, no automation is provided.

Extensibility Several possible extensions to the simulation environment exist. Some important issues include

- increased support for using legacy code directly in the simulator (e.g., by adhering to the POSIX standard and providing special wrapper functions that translates POSIX-code to the TrueTime environment)
- extensions of the network simulation (e.g. by adding support for simulation of wire-less and ad-hoc networks)
- connections with worst-case execution time analysis tools to come up with reasonable code execution times

Availability TrueTime is available for download at

`http://www.control.lth.se/~dan/truetime/`

9. XILO

9.1 Tool Overview

XILO - standing for X-in-the loop simulation - is a prototypical toolset, built upon Simulink, developed to support detailed architectural design of distributed real-time control systems, [El-Khoury and Törngren, 2001]. The approach enables the co-simulation of functionality, from discrete-time control to logic, together with the controlled continuous-time processes and the behaviour 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 XILO toolset is composed of a number of libraries that lets a designer configure a distributed computer control system and to allocate and partition the functionality as desired. Along with the basic toolset, an additional library that supports fault-injection in terms of bit-flips in all types of blocks, signals and constants has been developed, [Norberg and Törngren, 2003]. Some of the basic mechanisms of XILO have been reused in the Aida toolset [Redell *et al.*, 2004].

9.2 Comparative Aspects

Scenarios and Development Stages Supported The usage of this toolset is similar in nature to the AIDA toolset [Redell *et al.*, 2004] and TrueTime [Cervin *et al.*, 2003; Henriksson *et al.*, 2003]. The main emphasis is on architectural design on a rather detailed level, although usage in earlier stages is also possible. Given a control design, the resulting control performance can be evaluated for different computer system architectures and for different mappings to the architecture. Alternatively, it is possible to just use the simulation to get an idea about the timing of the computer system.

In the typical usage scenario, the control system has been designed, and it is of interest to study how it can be implemented on a single processor or in a distributed computer system. The hardware and software architecture could be fixed beforehand, or its design could also be guided by the results of XILO simulations. To use XILO, a new Simulink model is created, and by using the XILO block libraries the hardware and software architectures are defined. The functionality of the original control system model is then partitioned to the tasks (in general time or event-driven activities) of the nodes, inter-thread communication is added and the whole system is configured (e.g., by adding execution time information and priorities). Based on such a developed model in Simulink, alternative designs can be simulated and their behaviour compared. Hence, the toolset could be used to for example:

- compare and evaluate hardware architectures and function allocation
- compare and evaluate software architectures and task partitioning
- compare and evaluate different execution strategies, e.g. different triggering and scheduling approaches
- compare and evaluate control system designs

Qualities/Constraints Addressed The main qualities addressed include control system performance and robustness, and the timing behaviour of the computer system.

Methodological Considerations The above described scenarios relate to and support system design for distributed systems [Törngren and Wikander, 1996]. The modelling foundations for XILO are based on the AIDA modelling framework [Redell, 1998; Redell and Törngren, 1998] and also inspired by the CODARTS method [Gomaa, 1993].

Tool Architecture XILO is completely implemented within MATLAB/Simulink [The Mathworks, 2005]. MATLAB/Simulink was chosen because of the relative ease with which it could be extended to model real-time implementations, because of its support for the modelling of event-triggered and time-triggered systems, and because of its wide-spread use as a control design environment. 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.

On the other hand, there is a gap between the Simulink modelling level and the real-time system implementation. For example, in Simulink, the execution of a block takes zero time according to the simulation time, thus there is a need to provide a mechanism whereby durations due to execution or communication can be modelled. Moreover, the peculiarities of embedded system platforms (including, among other things, hardware, interrupts and real-time operating systems) need to be appropriately modelled and ”instrumented” into Simulink to ensure that the simulation behaves as a real-time implementation, in particular considering scheduling and preemptions. That is, the original execution ordering undertaken by Simulink during simulation is not sufficient [Törngren *et al.*, 2001].

The handling of hybrid systems requires 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 in the XILO libraries (further discussed below) is represented in the simulator as a Simulink block.

From the definition of the modelling content below it will be seen that there is extensive data exchange between the XILO library 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.

Table 1 Explicit XILO 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

Tool Inputs Each of the model components in the XILO libraries is 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. 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. In order to use the tool according to the intended scenario, a control system model made in Simulink is needed.

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.

Tool Outputs The simulator permits the user to monitor any variable in the system. Parameters that may be of interest for timing analysis include: task status, the times when particular events or activities within a task occur, or the time when a service request is serviced. Apart from these outputs, ordinary Simulink outputs can be used to for example store or visualize the behaviour of the controller and the controlled system.

Modeling Content 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 environment is described using standard Simulink modelling techniques and blocks, typically to capture a model of the dynamics of a mechanical system including sensors and actuators. It is 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 signals of the mechanics.

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

A node consists of the following types of components:

- one or more tasks from which the system software is built (the following section describes the task model)
- 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 role of the scheduler 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 pre-defined 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 customising 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.

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 27, 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).

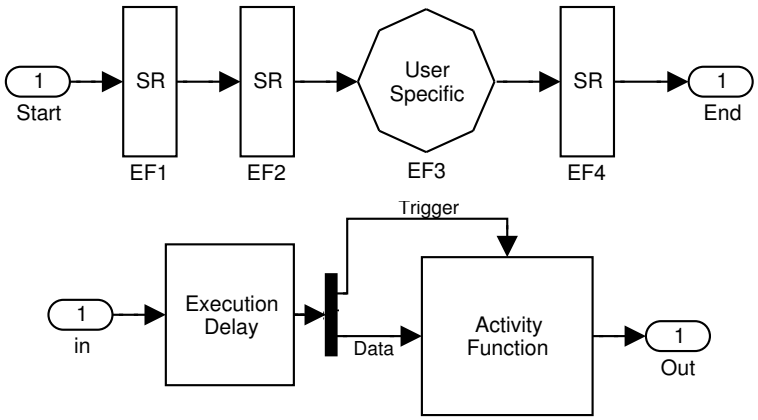

Figure 27 (top) The task model consisting of a sequence of elementary functions (EF). (bottom) The internal model of an EF.

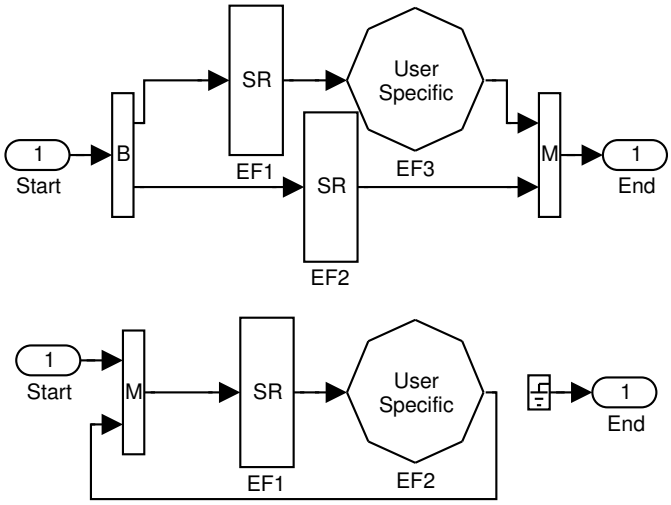

Figure 28 Examples of more complex task structures such as branching (top) or looping (bottom).

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 28. 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.

Elementary Function (EF). The internal model of an EF is shown in the bottom part of Figure 27. 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 pre-emption.

Tool Automation Currently no automation is implemented in XILO (apart from certain configuration aspects as described previously in the tool architecture section).

Extensibility XILO is completely developed in Simulink. Given the rich API of Simulink and possibilities to integrate custom-code there are many possibilities open for extensions. For example, it would be possible to develop platform models at different levels of abstraction, and to support different kinds of platforms (e.g. different networks and unsynchronized nodes). In addition, the idea with the toolset was to also support the definition and analysis of different approaches for error detection and handling. Such facilities have been studied but are currently only partially implemented in the toolset. It is straightforward to implement fault-injection in Simulink related to different fault-models, including permanent and transient faults, and component failure modes of different kinds, from crash to asymmetric failures. For experiences with such models in Simulink see [Törngren *et al.*, 2001; Norberg and Törngren, 2003]. In [Norberg and Törngren, 2003] we describe the development of a Simulink library that models transient hardware faults, in particular single bit-flips, and its use in control system robustness evaluation. 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. Integration with commercial tools such as code generators is also possible.

Availability Developed in-house at the Division of Mechatronics at KTH; available upon request.

10. Other and Commercial Tools

The survey in this report mainly focuses on recent tools as developed in academia. Apart from an increasing interest in the academic communities, there are also strong industrial needs for tools supporting a range of issues including timing, quantization, code generation, testing and so on.

To further illustrate the range of tools being developed, this section provides a glimpse of existing commercial tool efforts related to the area of real-time control. For other surveys see [ARTIST2 Network of Excellence, 2005; Jeutter and Heppner, 2004; Törngren and Larses, 2004].

Existing commercial tools provide a broad range of capabilities including support for (see for example [dSPACE, 2004; ETAS, 2004; National Instruments, 2004]):

- system modeling and design where for example effects due to constant or varying delays can be investigated in simulation
- rapid control prototyping (RCP), allowing control designs to be quickly prototyped using general purpose controller hardware
- code generation from control system models
- RTOS configuration and integration within the design models
- analysis of quantization effects, e.g., relevant for fixed-point implementation
- testing of models, generated code, and final implementations
- calibration of target systems, e.g., over CAN

The use of code generation has increased significantly only over the last few years in the vehicular industry. For example, Volvo cars are using Simulink models in the design of power train controllers including simulation and rapid prototyping. Code generated from the models is used in the final product, [Lygner, 2002]. Here it is interesting to note that the code generator design environment acts as an interface between control designers and implementation engineers

Although RCP tools and some testing and calibration tools can be used for distributed systems, the main effort has been on support for single processor systems.

Newer tools that are emerging have different origins and thus different emphasis. For example, efforts from electronics design automation are (both within research and as commercial spin-offs) addressing architectural design of embedded systems, thus for example supporting distributed system analysis and implementation of functionality, see for example [Metropolis, 2004].

There are also efforts originating from safety- and mission-critical systems, emphasizing for example fault-tolerance and formal verification. This range of tools also promise support for distributed control systems, see for example [TTTech, 2004; Sildex, 2004; TNI, 2004].

As an illustration of this functionality, the tools provided by TT-Tech support configuration and off-line scheduling of fault-tolerant distributed systems (including nodes and the communication), target downloading, testing and on-line monitoring. The tools have been integrated with Simulink and available code generators. From the perspective of control and real-time implementation co-design, the tools work with control designs and their timing requirements as inputs, and then essentially produce as outputs a distributed fault-tolerant middle-ware together with an off-line schedule. Apart from being one part of the distributed RTOS configuration, the synthesized off-line schedule can be used to instrument

a Simulink simulation (TTP-Matlink) to investigate the control system behavior subject to the expected timing of the implementation (this is similar to the approach taken in Aida, which however in the current version supports this for asynchronous and fixed-priority scheduled systems).

11. Summary

Designing a real-time control system is essentially a co-design problem. Choices made in the real-time design will affect the control design and vice versa. For instance, deciding on a particular network protocol will give rise to certain delay distributions that must be taken into account in the controller design. On the other hand, bandwidth requirements in the control loops will influence the choice of CPU and network speed. The need for a co-design approach is further accentuated in embedded control systems with limited computing and communication resources.

In order to simplify the design process for this type of systems it is important with tool support. Unfortunately the tools that allow a co-design approach are quite few. Instead most tools specialize on a single domain, e.g., control design, schedulability analysis or UML-type software modeling and code generation.

The aim of this survey has been to identify and summarize some of the most important co-design tools available. The tools presented are in general specialised on a certain aspect of the co-design problem. For example, Jitterbug support statistical control performance analysis taking computing and communication effects into account whereas TrueTime and RTSIM are tools for co-simulation of networked embedded control systems. The tools AIDA, Orccad, Ptolemy II, Syndex, and XILO all aim at providing environments for model-based developed of real-time control systems.

What so far mainly is lacking is tools that focus on the actual design part of co-design, i.e., which aid the designer with the development of the actual embedded control algorithms taking the control and communication aspects into account. The reason for the lack of this type of tool is the lack of theory and methods in the field. Co-design of embedded control system is a fairly new area and most of the methods and theory developed so far are aimed at analysis rather than design and synthesis.

12. References

ARTIST2 Network of Excellence (2005): “ARTIST roadmaps, part I.” http://www.artist-embedded.org/Roadmaps/ARTIST_Roadmaps_Y2.pdf.

Audsley, N., A. Burns, M. Richardson, and A. Wellings (1994): “STRESS—A simulator for hard real-time systems.” *Software—Practice and Experience*, **24:6**, pp. 543–564.

Bass, J. M., A. R. Browne, M. S. Hajji, D. G. Marriott, P. R. Croll, and P. J. Fleming (1994): “Automating the development of distributed control software.” *IEEE Parallel and Distributed Technology: Systems and Technology*, **2**, pp. 9–19.

Bhatt, D., V. Thomas, and J. Shackleton (1996): "A methodology and toolset for the design of parallel embedded systems." *ACM SIGPLAN OOPS Messenger*, **7**, pp. 5–12.

Bollella, G., B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull (2000): *The Real-Time Specification for Java*. Addison-Wesley.

Cervin, A., J. Eker, B. Bernhardsson, and K.-E. Årzén (2002): "Feedback-feedforward scheduling of control tasks." *Real-Time Systems*, **23:1–2**, pp. 25–53.

Cervin, A., D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén (2003): "How does control timing affect performance?" *IEEE Control Systems Magazine*, **23:3**, pp. 16–30.

Cervin, A. and B. Lincoln (2003): "Jitterbug 1.1—Reference manual." Technical Report ISRN LUTFD2/TFRT--7604--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.

Control Systems Society (2004): "CACSD history." Home page, http://www.robotic.dlr.de/control/cacsd/cacsd/history.shtml.

dSPACE (2004): "Solutions for control." Home page, http://www.dspace.de.

Eaton, J. W. (1998): "OCTAVE." Home page, http://www.octave.org/.

El-Khoury, J. and M. Törngren (2001): "Towards a toolset for architectural design of distributed real-time control systems." In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. London, England.

ETAS (2004): "Engineering products and services." Home page, http://www.etasgroup.com.

Forget, J., C. Lavarenne, and Y. Sorel (2004): "Syndex v6 – user manual." Technical Report.

Gomaa, H. (1993): *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley.

Grandpierre, T., C. Lavarenne, and Y. Sorel (1999): "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors." In *Proceedings of the 7th International Workshop on Hardware/Software Co-design*. Rome, Italy.

Henriksson, D. and A. Cervin (2003): "TrueTime 1.1—Reference manual." Technical Report ISRN LUTFD2/TFRT--7605--SE. Department of Automatic Control, Lund Institute of Technology.

Henriksson, D., A. Cervin, J. Åkesson, and K.-E. Årzén (2002a): "Feedback scheduling of model predictive controllers." In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*. San Jose, CA.

Henriksson, D., A. Cervin, and K.-E. Årzén (2002b): "TrueTime: Simulation of control loops under shared computer resources." In *Proceedings of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

Henriksson, D., A. Cervin, and K.-E. Årzén (2003): "TrueTime: Real-time control system simulation with MATLAB/Simulink." In *Proceedings of the Nordic MATLAB Conference*. Copenhagen, Denmark.

Hylands, C., E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng (2003): "Overview of the Ptolemy project." Technical Report UCB/ERL M03/25. Department of Electrical Engineering and Computer Science, University of California Berkeley, CA.

Jeutter, R. and B. Heppner (2004): "Model-based system development–is it the solution to control the expanding system complexity in the vehicle?" In *Proceedings of the SAE World Congress*. Detroit, USA.

Lauwereins, R., M. Engels, M. Adé, and J. a. Peperstraete (1995): "Grape-II: A system-level prototyping environment for dsp applications." *IEEE Computer*, **28**, pp. 35–43.

Lavarenne, C., O. Seghrouchni, Y. Sorel, and M. Sorine (1991): "The Syndex software environment for real-time distributed systems design and implementation." In *Proceedings of the European Control Conference*. Grenoble, France.

Lincoln, B. and A. Cervin (2002): "Jitterbug: A tool for analysis of real-time control performance." In *Proceedings of the 41st IEEE Conference on Decision and Control*. Las Vegas, NV.

Lipari, G. (2003a): "MetaSim." Home page, http://metasim.sssup.it/.

Lipari, G. (2003b): "RTSIM." Home page, http://rtsim.sssup.it/.

Liu, J., J. Eker, J. W. Janneck, and E. A. Lee (2002): "Realistic simulation of embedded control systems." In *Proceedings of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

Liu, J. and E. Lee (2003): "Timed multitasking for real-time embedded software." *IEEE Control Systems Magazine*, **23:1**, pp. 65–75.

Lygner, M. (2002): "Model-based development chain at Volvo Cars." dSPACE News, 1/2002, http://www.dspace.de.

Metropolis (2004): "Design environment for heterogeneous systems." Home page, http://www.gigascale.org/metropolis/.

National Instruments (2004): "Test and measurement." Home page, http://www.ni.com.

Norberg, J. and M. Törngren (2003): "Fault injection into control algorithms." Technical Report TRITA–MMK 2003:37, ISSN 1400–1179, ISRN KTH/MMK/R-03/11-SE. Department of Machine Design, KTH, Sweden.

Palopoli, L., G. Lipari, G. Lamastra, and L. Abeni (2002): "An object-oriented tool for simulating distributed real-time control systems." *Software – Practice and Experience*, **32**, pp. 907–932.

Pernet, N. and Y. Sorel (2003): "Optimized implementation of distributed real-time embedded systems mixing control and data processing." In *Proceedings of the ISCA 16th International Conference: Computer Applications in Industry and Engineering (CAINE-2003)*. Las Vegas, USA.

Ptolemy Project (2004): "Ptolemy II." Home page, http://ptolemy.eecs.berkeley.edu/.

Redell, O. (1998): "Modelling of distributed real-time control systems, an approach for design and early analysis." Licentiate thesis TRITA-MMK 1998:9, ISSN 1400–1179, ISRN KTH/MMK–98/9–SE. Department of Machine Design, KTH, Stockholm, Sweden.

Redell, O., J. El-Khoury, and M. Törngren (2004): "The AIDA tool-set for design and implementation analysis of distributed real-time control systems." *Journal of Microprocessors and Microsystems*, **28:4**, pp. 163–182.

Redell, O. and M. Törngren (1998): "A modelling framework for design and analysis of distributed real-time control implementations." In *Proceedings of the 6th UK Mechatronics Forum*. Skövde, Sweden.

Sildex (2004): "An integrated toolset for systems engineering, from specification to tested code." Home page, http://www.tni-world.com/sildex.asp.

Simon, D., B. Espiau, E. Castillo, and K. Kapellos (1993): "Computer-aided design of a generic robot controller handling reactivity and real-time control issues." *IEEE Transactions on Control Systems Technology*, **1:4**.

Simon, D., B. Espiau, K. Kapellos, and R. Pissard-Gibollet (1997): "Orccad: Software engineering for real-time robotics." *A Technical Insight, Robotica, Special Issues on Languages and Software in Robotics*, **15:1**, pp. 111–116.

Simon, D. and A. Girault (2001): "Synchronous programming of automatic control applications using Orccad and Esterel." In *Proceedings of the 40th IEEE Conference on Decision and Control, CDC'01*. Orlando, USA.

Simon, D., R. Pissard-Gibollet, K. Kapellos, and B. Espiau (1999): "Synchronous composition of discretized control actions: Design, verification, and implementation with Orccad." In *Proceedings of the 6th International Conference on Real-Time Control Systems and Applications*.

Storch, M. F. and J. W.-S. Liu (1996): "DRTSS: A simulation framework for complex real-time systems." In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 160–169.

The Mathworks (2005): "MATLAB and Simulink for technical computing." Home page, http://www.mathworks.com.

TNI (2004): "Object oriented design & analysis tools." Home page, http://www.tni-world.com.

TTTech (2004): "Time-triggered technology." Home page, http://www.tttech.com.

Törngren, M., J. El-Khoury, M. Sanfridsson, and O. Redell (2001): "Modelling and simulation of embedded computer control systems: Problem formulation." Technical Report TRITA-MMK 2001:3, ISSN 1400–1179, ISRN KTH/MMK/R–01/3–SE. Department of Machine Design, KTH, Stockholm, Sweden.

Törngren, M. and O. Larses (2004): "Characterization of model-based development of embedded control systems from a mechatronic perspective – drivers, processes, technology, and their maturity." Technical Report TRITA-MMK 2004:23, ISSN 1400–1179, ISRN KTH/MMK/R–04/23–SE. Department of Machine Design, KTH, Stockholm, Sweden.

Törngren, M. and J. Wikander (1996): "A decentralization methodology for real-time control applications." *Journal of Control Engineering Practice, Special section on the Engineering of Complex Computer Control Systems*, February.

Vestal, S. (1994): "Integrating control and software views in a cace/case toolset." In *Proceedings of the IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, pp. 353–358. Tucson, Arizona.