



ROYAL INSTITUTE
OF TECHNOLOGY

TRITA - MMK 2006:11
ISSN 1400 -1179
ISRN KTH/MMK/R-06/11-SE

Co-design of Control Systems and their real-time implementation - A Tool Survey

by

Martin Törngren, Dan Henriksson, Ola Redell,
Christoph Kirsch, Jad El-Khoury, Daniel Simon,
Yves Sorel, Hanzalek Zdenek and Karl-Erik Årzén

Stockholm
2006

Technical Report

Mechatronics Lab, Department of Machine Design
Royal Institute of Technology, KTH
100 44 STOCKHOLM, Sweden



ROYAL INSTITUTE
OF TECHNOLOGY

TRITA - MMK 2006:11
ISSN 1400 -1179
ISRN KTH/MMK/R-06/11-SE

Co-design of Control Systems and their real-time implementation - A Tool Survey

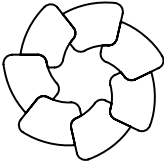
by

Martin Törngren, Dan Henriksson, Ola Redell,
Christoph Kirsch, Jad El-Khoury, Daniel Simon,
Yves Sorel, Hanzalek Zdenek and Karl-Erik Årzén

Stockholm
2006

Technical Report

Mechatronics Lab, Department of Machine Design
Royal Institute of Technology, KTH
100 44 STOCKHOLM, Sweden

 <p style="text-align: center;">MMK</p> <p style="text-align: center;">Mechatronics Lab Department of Machine Design Royal Institute of Technology, KTH 100 44 STOCKHOLM, Sweden</p>	TRITA - MMK 2006:11 ISSN 1400 -1179 ISRN KTH/MMK/R-06/11-SE	
	<i>Document type</i> Technical Report	<i>Date</i> 2006-09-26
	<i>Author(s)</i> Martin Törngren, Dan Henriksson, Ola Redell, Christoph Kirsch, Jad El-Khoury, Daniel Simon, Yves Sorel, Hanzalek Zdenek and Karl-Erik Årzén	
	Corresponding author: martin@md.kth.se	
<i>Title</i> Co-design of Control Systems and their real-time implementation - A Tool Survey		

Abstract

Increasing needs for optimized designs and for handling the dependencies among control systems and their real-time implementation, cause a resulting need for tools that have the abilities to support design across these traditional discipline boundaries. Tools supporting such co-design provide new opportunities in developing cost-efficient, dependable and robust solutions where the interactions between control and implementation engineers can be improved, and where further possibilities are obtained by increasing the information flow between the control system and the hardware/software platform during run-time. Work on co-design has the important effect to stimulate new theoretical research directions where more work is needed because of the lack of theory and methods in the field. Co-design related to embedded control system is a fairly new area and most of the methods and theory developed so far are aimed at analysis rather than synthesis. A further great challenge is that of model and tool integration, where there are needs to coordinate and integrate the multitude of aspects and specialized models/tools that are used in the development of embedded control systems. This survey sets the context of embedded control systems development describing what is achievable with current generation industrial tools. The context is further elaborated by discussing categories of tools from different related research disciplines. A set of representative co-design tools are then described in depth and discussed.

Index

1	INTRODUCTION.....	9
1.1	REPORT BACKGROUND	9
1.2	BACKGROUND AND NEEDS.....	9
1.3	OPPORTUNITIES AND CHALLENGES IN CO-DESIGN.....	11
1.4	FOCUS, ORGANIZATION AND CONTENT OF THE SURVEY	13
2	AN OVERVIEW OF EMBEDDED CONTROL SYSTEMS DEVELOPMENT AND AVAILABLE TOOL SUPPORT.....	17
2.1	MODEL BASED DESIGN AND RAPID CONTROL PROTOTYPING	19
2.2	TARGET CODE GENERATION.....	19
2.3	ANALYTICAL VERIFICATION	21
2.4	SIMULATION AND TESTING BASED VERIFICATION AND VALIDATION	22
3	TOOLS FROM CLOSELY RELATED DOMAINS	25
3.1	HARDWARE-SOFTWARE CO-DESIGN	25
3.2	MULTI-MODEL DESIGN ENVIRONMENTS AND HYBRID SYSTEMS	26
3.3	DISCRETE-EVENT SYSTEMS.....	28
3.4	NETWORKING TOOLS	28
3.5	REAL-TIME SCHEDULING AND COMPUTING	29
3.6	SAFETY AND RELIABILITY.....	31
4	OVERVIEW OF SELECTED TOOLS.....	33
4.1	AIDA	33
	<i>Tool Overview</i>	33
	<i>Comparative Aspects</i>	35
4.2	JITTERBUG	39
	<i>Tool Overview</i>	39
	<i>Comparative Aspects</i>	40
4.3	ORCCAD.....	44

<i>Tool Overview</i>	44
<i>Comparative Aspects</i>	45
4.4 PTOLEMY II	51
<i>Tool Overview</i>	51
<i>Comparative Aspects</i>	52
4.5 RTSIM.....	55
<i>Tool Overview</i>	55
<i>Comparative Aspects</i>	58
4.6 SYNDEX.....	60
<i>Tool Overview</i>	60
<i>Comparative Aspects</i>	62
4.7 TORSCHÉ.....	64
<i>Tool Overview</i>	64
<i>Comparative Aspects</i>	65
4.8 TRUETIME	71
<i>Tool Overview</i>	71
<i>Comparative Aspects</i>	74
5 DISCUSSION: TRENDS AND CHALLENGES	81
6 CONCLUSIONS.....	82
7 ACKNOWLEDGEMENTS.....	83
8 REFERENCES.....	84

1 Introduction

1.1 Report background

This report has been produced as part of the research work carried out by the Control for Embedded Systems Cluster within the ARTIST2 network of excellence [ARTIST2, 2006].

The origin of the report traces back to work carried out at KTH and LTH in work on co-design tools. A first attempt of a tool survey was carried out as part of the Swedish Flexcon project (ending 2005). During the first year of the ARTIST2 project this work was extended to encompass a contextual perspective of industrial tool usage and tools from related domains. During the second year of ARTIST2 this draft report was extended with descriptions of more tools and finalized. A short summary of the report has been published separately [Törnngren et al., 2006].

1.2 Background and needs

In the early days of computer control system design, resource constraints were legion. Memory was scarce as was computational performance and accuracy. Failure rates were high and communication rates low. During the 60s-70s the (few) designers working in the area were well aware of the need for co-design of the control systems and its electronics and software implementation; that is, the fact that control design decisions had an impact on the implementation whereas decisions taken in terms of which electronic components to use and how the software was implemented, constrained the control system design. Designers at that time were often responsible for developing both the control system and its implementation, [Motus and Rood, 1994].

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 became possible to support a wider range of tool functionality. 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], developed for digital signal processing systems and supporting distributed systems (allocation, scheduling, partitioning).
- Efforts by Honeywell labs including MetaH and the Parallel Scalable Design Tool-set, [Vestal, 1994; Bhatt et al., 1996]. The MetaH effort has been the basis for further work the Architecture and Analysis Description Language, which now has been standardized by the SAE, [AADL, 2004].

The evolution of electronics and software over the last decades has provided a technology basis making it possible to realize virtually all kinds of control related functionality in different products. Software has become the competitive advantage in embedded control systems allowing unprecedented flexibility. Many examples can be given including the evolution of automotive control systems such as braking control and engine management, the use tracking and focus control in CDs/DVDs (now available per piece at a few dollars cost) to industrial robotics (where software now is the dominating cost in development).

Networking of such embedded control systems have followed the introduction of stand-alone controllers in a rather rapid pace, being introduced in process control in the 70s, in aerospace in the 80s and in the automotive industry in the 90s. Networking initially had the basic purpose to reduce the cost of installations (reduced cabling, shared sensors, facilitated diagnostics etc.), but once in place, will be used to realize new coordinating of existing subsystems thus creating new functionality. Apart from product internal networks, embedded systems are also increasingly being connected to external systems, for example for maintenance purposes. Again entirely new functionality is possible by such connections, for example coordinating a fleet of vehicles.

In consequence, many computer-controlled systems are today 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.

While the above mainly shows the possibilities in terms of new functionality and increased performance, this evolution has also drastically increased the complexity of the resulting systems. This complexity has many facets. For example, considering again the automotive industry, a top-of-the line car today has some 70 nodes (microprocessor based core entities of the distributed system) which are delivered by some 30 vendors. 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 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.

Pursuing the use of embedded control systems further thus requires efficient complexity management. The traditional separation of engineers into different disciplines is closely related to the increasing complexity but the division into disciplines can also create problems. For control systems, the typical separation between control and implementation engineers makes it important to define and appropriately handle design issues that have an impact across these domains. Issues such as control bandwidth and computational structure vs. choice of processors and processor scheduling will affect, and be dependent, on each-other. Ineffective support for communication and analysis across the domains covered by the disciplines may not only cause lengthy and costly iterations but also later product failures with even more serious consequences.

In addition, there is a strong trend within industry today to use commercially available information technology and commercial-off-the-shelf (COTS) components deeper and deeper in the real-time control systems. These components have restricted configurability and may not be well suited for control systems. Limited resources combined with non-optimized hardware and software components introduce non-determinism 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. Tough product demands in terms of competition and legislation will moreover cause a need for optimizing system designs where the trade-offs (and design issues) at hand, e.g. control performance vs. word-length/price of processor, will affect more than one engineering domain. The needs for optimization are particularly relevant for large series production where the goal is to make the hardware cost proportion as small as possible. Typical examples are provided by the automotive industry which try to squeeze control functionality onto as small microprocessors/controllers as possible.

Co-design is also required for emerging highly safety-critical applications such as steer and brake by wire. The traditional approaches for achieving highly dependable systems are not really viable here because they are so costly; instead there is a need to combine application and computer system measures in order to develop cost-efficient dependable computer control systems – i.e. co-design between the control system and its implementation is required.

Optimization of the control system implementation will have the effect that resource constraints appear, constraints that to be properly accounted for during control design. Constraints can also occur in products with smaller series. This is for example the case in novel applications such as wireless distributed systems where power and communication constraints will affect the control system design.

All in all there is consequently an increasing need for tools that support co-design of control systems with their electronics and software implementation. For optimal use of computing resources, the control algorithm and the control software designs need to be considered at the same time, or alternatively, given the constraints of e.g. a COTS based platform, the imperfections it provides has to be taken into account in the control design.

1.3 Opportunities and challenges in co-design

There are many instances of the control / computer implementation co-design problem. These can typically be formulated as optimization problems; consider the two following examples:

Control and scheduling co-design problem: *Given a set of systems to be controlled and a computer with limited computational resources, design a set of controllers and schedule them as real-time tasks such that the overall control performance is optimized.*

Control and cost co-design problem: *Given a set of systems to be controlled and control performance specifications for these, choose an implementation in terms of a distributed computer system including deciding the allocation of control functions, their partitioning into tasks, scheduling and triggering, such that the overall production cost is minimized while guaranteeing the specified control performance.*

A major challenge, however, in solving these types of optimization problems is that the relations between the involved parameters are non-linear and sometimes even difficult to formulate, see FIG. 1 which illustrates some of the relationships. Although confined optimization problems can be formulated, and sometimes also solved analytically, real problems are even more complex and typically involve optimization of several variables. In realistic settings, the support of simulation and what-if-type analysis is therefore an important approach.

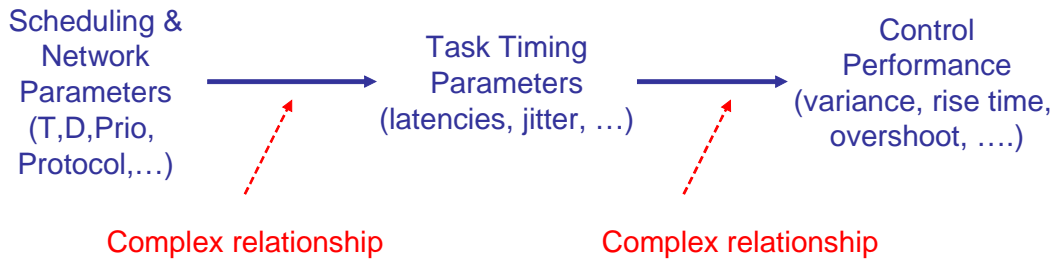


Figure 1. Complex relationships between control system qualities and computer system design parameters

As outlined above, there are many variants of the co-design problem and thus problems that tools can be targeted to solve.

Whereas the co-design takes place during system development it can target both design time optimization as well as the design of on-line interactions, for example, dealing with re-configuration in order to handle changing conditions such as changing loads or partial system failures. The definition of the division of responsibilities and the information flow, required during run-time, between the controller and the platform is an important part of the co-design work. Examples of such issues include the determination of where to detect certain conditions such as e.g. a computational failure; should this be the responsibility of the platform or the controller? Where should the resulting error handling be placed?

A key challenge for the implementation of control systems is that of trying to define the appropriate abstractions and their dependencies with discipline specific design parameters (compare with FIG.1). Experiences have shown that research into an interdisciplinary area such as control/computer co-design can help to identify the gaps

between different theories and stimulate entirely new research directions, [Törnngren et al., 2001].

1.4 Focus, organization and content of the survey

The main focus in this survey is on tools that provide a bridge between the domains of control function and computer system design in that they allow

- one or more trade-offs to be resolved (e.g. cost vs. performance),
- constraints, on behalf of the control or computer system, to shape the design of the other.

Examples of issues involved include from the control side

- choice of controller synthesis method (e.g. taking robustness and compensation with respect to computer system deficiencies into account)
- algorithm computational requirements (e.g. types of arithmetics required)
- resulting memory requirements (code and data, volatile-/non volatile storage)
- algorithm structure and information flow (causing data and control flow requirements)
- control design (causing accuracy requirements in computation as well as requirements on assumed timing; delays, periods and admissible jitter)
-

and from the computer system side

- choice of hardware components (affecting cost, quantization in computations, and basic communication and computational speed)
- the choice of networking protocol
- the software logical structure, partitioning into tasks
- the overall execution structure of the system (including the triggering of actions, synchronization and scheduling – determining the actual timing of the system).

In addition, certain decision and issues lie in-between these traditional views, for example the allocation of control functionality to computational elements. The execution time of a particular control system function will also depend both on the controller design as well as on the implementation method including the software/hardware platform. Other, non-control related functionality, also has to be considered since it can have an impact on for example the system timing.

The focus in the survey in terms of the product life-cycle is on tools that support system design related to modeling, analysis and synthesis. The emphasis is on “upper case” tools – i.e. tools that support design at the conceptual level of function and implementation architecture design. A further delimitation is that the types of analysis in main focus are related to the run-time behavior of the system (e.g. control system, timing behavior and power consumption) – as opposed to non-run time properties (typically more purely structurally related) such as maintainability and hardware reliability.

A challenge in this survey is that fact that the area is very dynamic with new tools frequently emerging and that there are already a multitude of existing tools that in one way or the other support co-design. While academic tools often are dedicated to one or a few tasks, commercial tools often support a wide range of design tasks that may span a large portion of the system development.

The tools chosen for the given focus have in common that they directly support co-design of control systems and their implementation in terms of being able capture the appropriate models, analyse dependencies related to trade-offs and/or capable of constraints based synthesis.

Tools that support a one way synthesis from control specification to implementation, without explicitly (or weakly) formulated constraints or dependencies, are not covered in the survey part, thus for example excluding pure rapid prototyping systems. Generic tools that do not explicitly support control systems have been excluded from the survey. In addition, for tools that have a basic ability for co-design but where there are add-ons that provide much more interesting capabilities, only add-on tools are covered (one example of this is Matlab/Simulink¹ which is not included by itself but rather indirectly since some of the chosen tools are based on it).

The tools chosen for the survey should finally not be seen as exhaustive, but rather as highly representative for the given focus.

The following tools are included in the survey:

- *AIDA* from the Royal Institute of Technology, Sweden.
- *Jitterbug* and *TrueTime* from Lund University, Sweden
- *Ptolemy II* from the University of Berkeley, California
- *RTSIM* from the RETIS Laboratory, Pisa, Italy
- *SynDEx* and *Orccad* from INRIA, France
- *TORSCHÉ* from the Czech Technical University in Prague

The tools included in this report approach co-design of real-time control systems in different ways, with specific design scenarios and concerns in mind. Some of the tools, such as TrueTime and AIDA, are specially 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. Some of the tools are more directed towards synthesis, e.g. in terms of code generation such as ORCCAD, but also for the assignment in space and time of control functions, the case for SynDEx and TORSCHÉ. Finally, the tools also have an origin from different disciplines, but all have been extended to cater for control and computer system co-design.

¹ From the Mathworks: <http://www.mathworks.com>

As an aid to the reader, and also to be able to relate the chosen focus area to other related tools, the survey begins in Section 2 with an overview of control systems development and the functionality that is provided by existing high-end commercial tools. As a further contextual outlook, Section 3 gives an overview of closely related domains with which there are some connections to the tools discussed here. Several examples of tools are given in Section 3, where the reader will note that these tools often are on the borderline to the area of co-design treated in this report; examples of such tools include AIRES (from the Univ. of Michigan), Sildex (from TNI), the TT-Tech tool-suite (from TT-Tech) and CAMEL-View (developed at the Univ. of Paderborn, now available as a commercial tool).

Section 4 contains the survey of the above mentioned tools. Finally, Section 5 discusses their characteristics and outlines trends and challenges in the further research and development of co-design tools.

A complementary description of the area of co-design is given in the corresponding Roadmaps by the ARTIST2 Control for Embedded Systems Cluster, [ARTIST2 Control cluster roadmaps, 2006]. These roadmaps are available on-line² together with other publications produced by the cluster.

Other useful surveys and roadmaps include the

- Handbook of Networked and Embedded Control Systems [Hristu-Varsakelis and Levine, 2005], which indirectly describes a number of related tools and many of the concerns involved in co-design.
- ARTIST roadmaps [Bouyssounouse and Sifakis, 2005], describing the needs in many application domains and giving examples of tools and their functionalities.
- The COLUMBUS project survey on modeling and tools for hybrid systems [Carloni et al., 2004], giving an overview of tools dedicated to hybrid systems.

² Publications by the ARTIST2 Control for Embedded Systems Cluster:
<http://www.md.kth.se/RTC/ARTIST2/publications.html>

2 An overview of embedded control systems development and available tool support

The strong market demands, an increasingly competitive pressure and the increasing system complexity are driving and making the use of powerful tool environments ever more important.

Control engineering has a strong tradition in model based development, using well founded mathematical concepts for describing the plant to be controlled, the control systems as well as disturbances affecting the plant and sensors. Control engineering models are used to communicate designs among developers (within and in-between companies), for system analysis as well for synthesis. However, industrial adoption and practices still vary to a great extent. Some industrial domains are characterized by control theoretic and model based approaches supporting the development. Other domains, however, such as automotive engine control, rely heavily on look-up tables and calibration of systems for control purposes – i.e. there is less of a tradition of model based control.

The maturity of model based development thus varies a lot but is in general more developed in “Control for embedded systems” compared to “software development for embedded systems”. Many domain specific traditions are also in place as mirrored by the wide variety of specialized modelling languages and tools, see e.g. [Bouyssounouse and Sifakis, (2005), and Törnngren & Larses (2005)].

The development and tools discussed in this section represent the more advanced model-based development practices that can be found in industry. This section provides an introduction to representative capabilities provided by existing commercial tools, including their support for

- system modeling and design
- rapid control prototyping (RCP), allowing control designs to be quickly prototyped using general purpose controller hardware
- analytical verification of system properties
- code generation from control system models including analysis of quantization effects, e.g., relevant for fixed-point implementation
- testing of generated code and final implementations

Other important capabilities supported by many tools, but not covered in this report, include the calibration of target systems (e.g., over CAN, where for example control parameters and look-up tables can be fine tuned and the control performance analyzed on-line) and configuration management, providing versioning and change management of for example design models, tests and components.

The above listed functionalities are not distinct and can thus be used in different development stages and the functionalities to some extent also overlap. They are briefly described in the following sections.

There exist a large number of tools that support more or less of the above sketched functionality. The tools can differ in the modeling paradigms provided (e.g. continuous-time, discrete-time, vs. discrete-event) and in the provided analysis and synthesis support. The most common type of analysis is that of simulation based

testing; examples of several forms of testing are given in the following sub-sections. Many tools support some form of code generation where two types can be distinguished; code generation for rapid prototyping vs. for production systems. In the former case, there are less stringent requirements on the code generator since prototyping systems typically constitute resource adequate systems, i.e. systems with more than plenty of execution and memory resources. For production code generation there are additional requirements which require more elaborate optimization of the use of execution and memory resources.

Examples of tools that support a larger portion of the above mentioned functionality include the tool chains provided by the companies Mathworks³/dSPACE⁴, Etas⁵ and Esterel Technologies⁶.

Systems development is often depicted using different process models. The V-model, illustrated in FIG. 2, is such a model, which is commonly used although it is highly simplified.

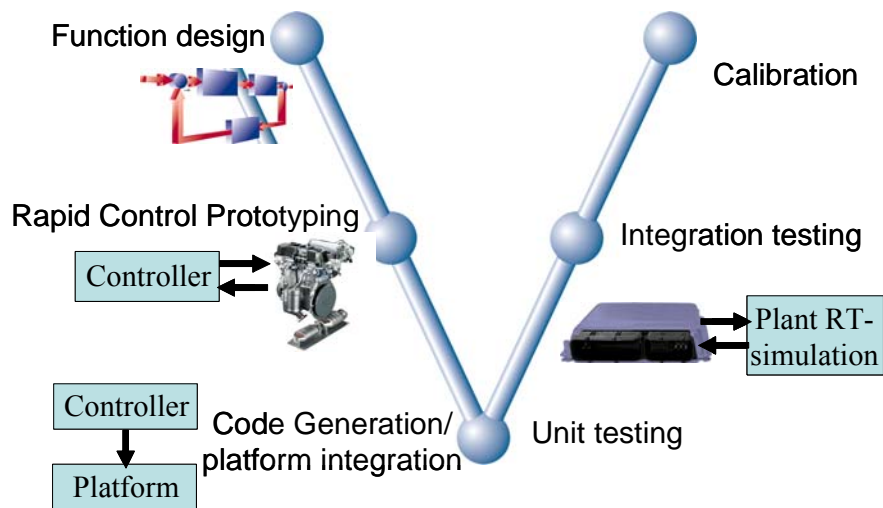


Figure 2. The V-cycle model of control system development, illustrating the use of tools that support various design activities, from early modeling and simulation to hardware in the loop testing.

The V-model does not, for example, illustrate the iterations part of the development and the dependencies that exist between different development stages. The simplifications part of the V-cycle can also be seen by considering tasks that have to be part of control system development. Examples of such tasks include

- Defining initial requirements and constraints

³ <http://www.mathworks.com>

⁴ <http://www.dspace.com>

⁵ <http://en.etasgroup.com/>

⁶ <http://www.esterel-technologies.com/>

- Defining and planning the development project including choosing supporting tools
- Deciding the control design approach and performing the control design
- Choosing sensors and actuators
- Choosing the computing platform (software and hardware)
- Mapping the control design to the computing platform
- Verifying that the requirements are satisfied by the implementation
- Validating that the solution actually meets stake-holder needs
- Fine tuning the final implementation by calibration

It is clear that all these steps are interdependent and that many different approaches are possible in ordering and/or parallelizing these tasks.

2.1 Model based design and rapid control prototyping

Typical for modern tools is that they support graphical specifications of control functionality. Classical methods of control theory can be used to identify and optimize the control characteristics of the function. As the graphical specifications provide an executable semantics, the control function can be simulated, either by stimulating it with appropriate input values or by having the function in a closed loop with plant and environment models. The graphical specifications include basic components or model elements which can be hierarchically decomposed. In describing the behaviour of each component there are typically several alternatives; a native behavior description can be used (for instance using pre-defined transfer function or state space components) or the behavior of the component can be defined by encapsulating programmed algorithms (e.g. coded in C-code). The components can be treated as white or black boxes.

Furthermore, the graphical specification of the control function can be extended by an appropriate I/O interface specification; in practice these specifications are just other components that include code that directly can access I/O devices. The resulting model can be compiled (where code is generated) and executed either on a PC directly or on a separate real-time hardware, each providing connections with the actual controlled system. This stage is called rapid control prototyping. It is an established method for fast design iterations.

In these early stages of the development process a control engineer will usually focus on the functional and behavior of the control function but not on details of a later software implementation on an electronic control unit. Modeling guidelines are important to ensure proper use and maintenance, but can also be required to ensure the proper behavior and the generation of efficient production code.

The design models form a basis for communication, analysis and code generation. Code generation and some specific analysis techniques are described in the following.

2.2 Target code generation

The use of code generation has increased significantly only over the last few years [Reuter et al., 2004]. There are several motives for introducing code generation in the first place including reducing the development time, ensuring consistency among

models - code - and documentation, reduction of programming faults, and ease of porting.

Important requirements on code generation technology are on the other hand to ensure

- efficiency in terms of memory and speed, and
- dependable code generation, providing testing and analysis techniques to ensure that the generated code behaves as intended

Using this technology, a control engineer will deliver an executable graphical specification as a reference for further simulations and as a basis for the generation of code for implementation.

Efficient code generation means that a minimum of execution time, RAM and ROM resources as well as stack size is required to run the code on an embedded processor. This minimizes chip size and costs. But production-quality code generation involves more than just these benchmarks. There are soft criteria that are equally important, including human readability of the code, traceability between code and model and target awareness of the code. Modern code generators typically support also the automatic generation of the software documentation.

Code generators use a range of techniques to meet such requirements including standard and inter-block optimizations and the use of code pattern libraries. For reasons of efficiency, some production code generators also have the ability to perform processor- and compiler-specific optimizations. FIG.3 illustrates the difference between optimized vs. non-optimized code-generation.

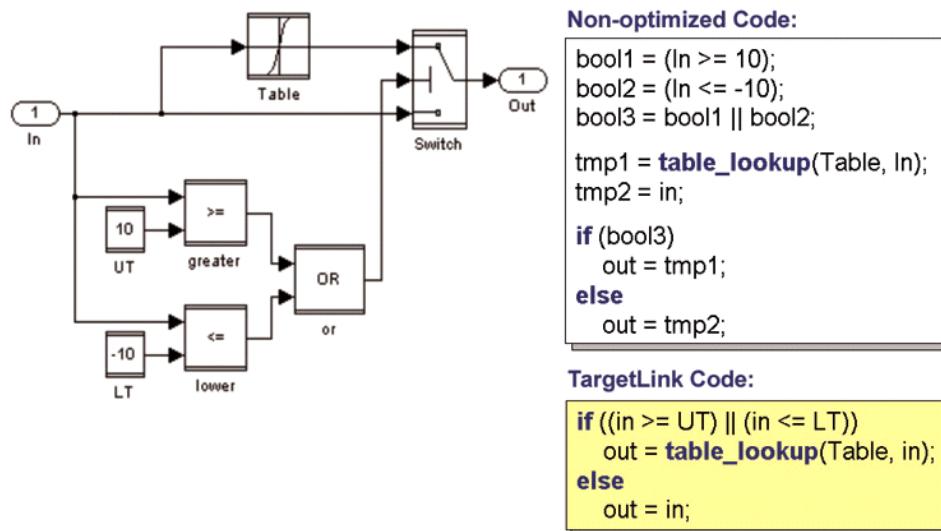


Figure 3. Code generation example; from Simulink to C-code (Courtesy of dSPACE).

Dependable code generation is closely related to the provision of complementary facilities for ensuring that the generated code is correct. For example, even though the control system model may have been verified, bugs in code generators, compilers and even hardware can cause erroneous behaviour. Examples of available testing environments supporting the progression and profiling, from models to implementation, are given in section 2.4.

In the context of this report, it is interesting to note that a code-generator in effect provides an interface between control designers and implementation engineers; it can thus serve as a co-design tool. This is particularly the case for the types of problems considered in this report, where resource constraints appear.

A control system design could in the simplest case correspond to a few discrete-time equations. In practice, a controller contains both state machine logic and such equations. In transferring such a control system design to software there are many decisions to be made and many trade-offs are possible. It is often desirable to minimize the required amounts of memory (e.g. allowing only on-chip memory to be used) and to reduce the execution time of a given algorithm (reducing the delay from sampling to actuation). At the same, it is desirable to use a cheap processor while providing the accuracy needed for computations. Unfortunately, these requirements are contradicting. While reduced memory consumption can be achieved, it typically comes at the cost of increased execution time and reduced accuracy. While execution times of algorithms can be reduced, this in turn typically requires more memory and reduces the accuracy of requirements, etc. Full automatic code generation is very difficult unless some of these constraints can be relaxed.

An example of a code generator that support designers in dealing with such trade-offs is Targetlink from dSPACE. In doing so, the tool provides support for evaluating control performance, execution time and memory consumption for given implementations in terms of choices of variable sizes, scaling of variables etc. For fixed point processor implementations, Targetlink supports scaling tasks and automation wherever possible.

2.3 Analytical verification

There are several theoretical approaches towards analytical verification of control systems. These approaches are strongly linked to the types of models, or models of computation, that they are developed for. As illustrated in Figure 4, these models could be characterized in terms their notion of data and time. Control theoretic approaches are strong in handling continuous time data and continuous time, for example allowing formal model based analysis of robustness and stability with respect to sensor noise and defects (or just modeling errors) in the controlled plant. The theory is also well developed for period (time-triggered) discrete-time systems whereas there is less work addressing discrete-data (quantization) and even less so for discrete-event systems, [ARTIST2 Control cluster roadmaps, 2006].

More recently, formal techniques with a computer science background have also been made available within in control engineering tools or by exporting models to other tools providing such facilities. Such techniques are applied to the discrete-event system parts (lower right hand part of Fig. 4) of control systems, see e.g. Ranville (2004) and the ARTIST roadmap, [Bouyssounouse and Sifakis, 2005]. The term ‘model checking’ was introduced in the early eighties. The capabilities of the technology have advanced significantly over the years, and it is now, for example, commonly applied in the area of hardware verification. Model checking is a technique that relies on building a finite state machine model of a system of interest and checking that a desired property holds in that model. One example of the type of

question that can be addressed is as follows: Can the ABS feature be requesting a decrease in engine torque at the same time as the cruise control feature is requesting an increase in engine torque? The use of model checking then requires formulating this type of requirement in the provided formal specification language and a system model that is amenable to analysis.

	Continuous data	Discrete data
Continuous time	$dx/dt(t) = Ax(t)+Bu(t)$	$dx_q/dt(t) = A_q x_q(t)+B_q u_q(t)$
Discrete time	$x(kh+h)=\Gamma x(kh)+\Phi u(kh)$	<pre> graph LR S1((S1)) -- "a, x:=0" --> S2((S2)) S2 -- "b, (x<2)" --> S1 In(()) --> S1 </pre>
Discrete event	$x(k+1)=\Gamma(k)x(k)+\Phi(k)u(k)$	

Figure 4. Model categories according to how data and time are represented. Theoretical approaches are less well developed for hybrid systems, where different models are mixed (e.g. a system containing several continuous systems where the connections and activations between them is controlled by a state machine).

A central problem in applying model checking is the scalability of the techniques, suffering from very large state spaces. This problem becomes even more cumbersome when dealing with embedded control systems that are composed of variables, continuous in both range and time, and incorporating a mixture of discrete-event and continuous time systems. Moreover, there has most probably also been a lack of training and in developing appropriate user interfaces. In adopting this technology, methodology and model extraction become important, to actually address the most relevant problems (subsystems, functions) and at the right level of abstraction. The use of such tools is right now emerging in embedded control systems and there is plenty of research in attempting to extend the technology, ([Bouyssounouse and Sifakis, 2005], [Carloni et al., 2004]).

2.4 Simulation and testing based verification and validation

The model-based development approach also brings new opportunities to perform tests⁷ at various stages within the development – not only at the end after having built the actual system.

⁷ Note that the use of the word testing here encompasses tests performed on models, physical systems/components or a combination thereof.

As soon as a model for the control function, a plant model and models of other pertinent aspects of the environment, such as disturbances, are available these models can be used in a closed control loop. The strength of testing through simulation is (at least) four-fold:

- There are few limitations in the types of systems that can be tested, e.g. works for non-linear systems and for hybrid systems
- The test conditions can be well defined and the tests are repeatable
- The tests can be automated.
- The tests can support a wide variety of purposes including verification in early development stages as well as analysis of failures during maintenance.

A corresponding challenge is that of defining relevant test scenarios and also in appropriately managing the test process. However, once such an environment is set up, it can be reused, not only for other systems/products but also incrementally during development. It is for example today common to support so called software-in-the-loop (SIL) simulation, where the basic setup is the same but where the control function part of the model has been replaced by the corresponding production code. The behaviour of the production code can then be compared with the behaviour of the function model.

A final step in testing is so called hardware-in-the-loop (HIL) simulation. Here, the real control unit is embedded within a testing environment that typically contains real-time simulation, electrical components emulating parts of the environment and some real physical components. Real-time plant models are required for realistic simulations. HIL simulation covers a broad range of test procedures, including

- correct interaction of networked components,
- test of diagnostic functions and communication,
- electrical fault simulation,
- simulation of not yet existing control units, so-called rest-bus simulation, systematic generation and injection of electrical and logical faults.

3 Tools from closely related domains

There are many areas and disciplines with tools that have some relation to the type of co-design tools being studied in this survey. FIG. 5 outlines a number of related areas. The characterization is for illustrative purposes mainly but does indicate that the areas are more or less connected and overlapping (all overlaps are not shown!). The purpose of this section is to briefly outline some these related areas and their connection to the tools studied in this report. Some examples are given. The tools covered in section 4 originate from several of the communities shown in FIG. 5.

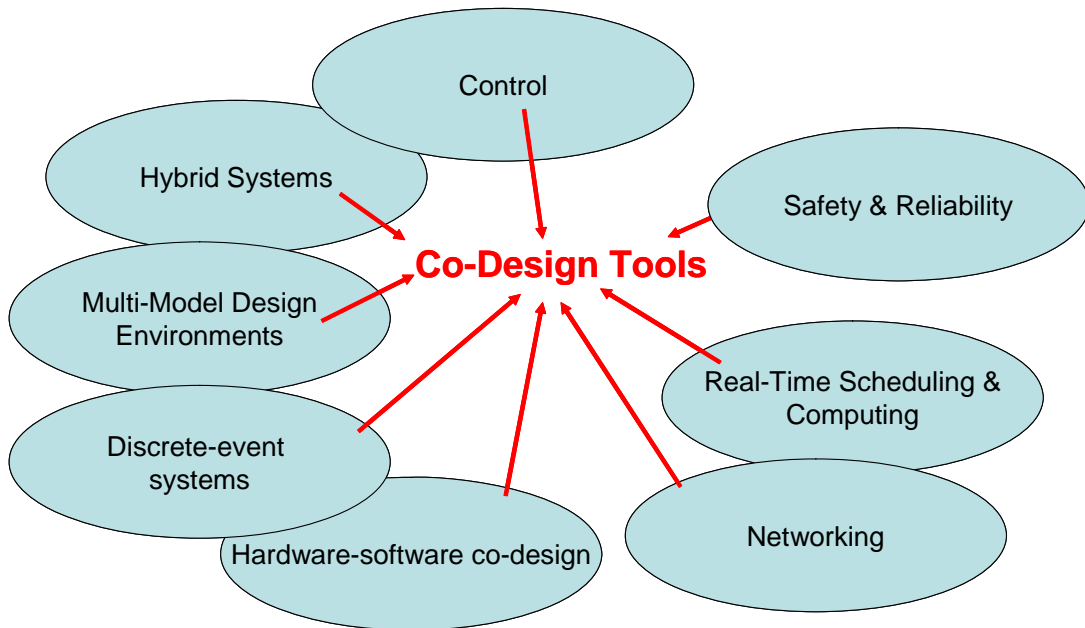


Figure 5. Sample research areas/disciplines with tools with some relation to the co- design of control systems and their implementation. Note that several of these areas address other dimensions of co-design.

3.1 Hardware-software co-design

Tools from the area of hardware-software co-design have many similarities to the types of tools studied in this report, where the main difference is the type of functionality considered (control functions vs. more or less general software). In common are the goals to provide support to designers that assist in mapping functionality to a platform while highlighting relevant trade-offs.

Tools in this area often utilize hardware/software modeling languages such as SystemC and VHDL, which can be seen as a kind of high-level languages including concurrent behavior. There are several examples of commercial design environments

(such as those from companies like Xilinx⁸, Cadence⁹ and Mentor¹⁰) that provide integrated design environments where both software and hardware, and the mapping of software onto the hardware, can be specified. The area is intensively evolving.

An interesting research project in this category is Metropolis¹¹ which has the aim to support design from specification to implementation on hardware and software platforms. Two important parts of Metropolis are its methodology and meta-model (tools are also available). The meta-model serves the purpose to allow different models of computation to be defined. The meta-model allows behaviors to be defined using the basic capabilities of actions, constraints, and their refinements. Quantity constraints enable the specification of performance and cost constraints such as time and power (more information on Metropolis is available in the Hybrid systems tool survey, [Carloni et al., 2004]).

3.2 Multi-model design environments and hybrid systems

Very closely related to hardware-software co-design area, is that of multi-model design environments. A typical example here is Ptolemy II. Ptolemy enables the definition of several model of computation, their assembly as a number of concurrent components and finally the definition of how these heterogeneous models interact. Ptolemy is described in more detail in section 4 since it provides pre-defined models of computation supporting co-design.

There are also modeling languages that in themselves provide a multitude of structural and behavioral descriptions. Examples in this category include UML2 and the AADL. While the UML was initially intended for general purpose software systems, work is being undertaken by the OMG¹² to provide more capabilities for non-functional aspects, e.g. for describing timing behavior and fault-tolerance, by defining UML profiles (an overview of some of these activities are provided in the ARTIST roadmaps, [Bouyssounouse and Sifakis, 2005].). The AADL on the other hand is a language dedicated to support the implementation of control systems, [AADL, 2004]. The AADL standard was developed based on the MetaH effort. A somewhat similar but broader effort is the development of the EAST-ADL, a description language for automotive systems [EAST-ADL, 2004] with some relation to the Autosar¹³ initiative. Both the AADL and the EAST-ADL provide constructs and properties required for analysis of safety, reliability and timing, as well as explicit models of software components.

A closely related and partly overlapping area is that of hybrid system tools. These are tools specialized in the modeling, simulation and formal analysis of systems composed of continuous and discrete event systems (manifested by combining differential equations with state machines). There are many tools that provide some support for hybrid systems (and possibly also other models of computations).

⁸ <http://www.xilinx.com/>

⁹ <http://www.cadence.com/>

¹⁰ <http://www.mentor.com/>

¹¹ <http://www.gigascale.org/metropolis/>

¹² <http://www.omg.org/>

¹³ <http://www.autosar.org/>

Example tools include Matlab/Simulink with Stateflow, Modelica/Dymola, Sildex and HyVisual. See [Carloni et al., 2004] for an excellent survey of hybrid system tools.

Control systems are more or less intrinsically hybrid systems. However, whereas the hybrid community studies the behavior of interacting continuous and discrete event subsystems, the co-design tools surveyed in this report focus on the interaction between the control system and its implementation. These types of tool categories therefore strongly complement each-other. However, it is often feasible to develop support for co-design functionality on top of the capabilities provided by hybrid system tools. An example of this is Matlab/Simulink from the Mathworks on top of which for example the TrueTime tool is built, further described in section 4.

Many of the hybrid system tools provide code generation. Some also provide explicit support for embedded systems design and in this sense contain ingredients from several of the above mentioned areas. As an example, consider Sildex, a tool-set for formally specifying and designing control and data-oriented real-time embedded systems. Sildex (which is produced by TNI¹⁴) targets safety-critical embedded software applications and is based on the synchronous language Signal. To describe each component, the user can choose from different styles: data-flow style, state machines, truth tables, or Grafsets. It is also possible to import Simulink diagrams or to write components in the C language.

The Sildex environment conforms to the description of model based design in Section 2.1 in supporting graphical components, their connections and hierarchy. Sildex provides two features for validating a specification diagram. Through simulation it is possible to execute the embedded code generated by the compiler and to study the evolution of the program's state machines and data flows. As a complement, there is a formal proof mechanism of safety properties (analytical verification of state machines as mentioned in Section 2.3). Code generation is support from the models to C and ADA code.

An advantage of Sildex is its foundation on a mathematically well defined language, facilitating the application of various analysis and synthesis techniques. The current focus of SILDEX is on embedded software design and the tool lacks the capability of modeling generic hybrid systems, [Carloni et al., 2004]).

Finally, another class of related efforts are those that instead focus on the integration of different domain tools and/or the management of the superset of information treated by different tools. In systems development there is for example the need to manage the dependencies and to provide traceability between related information artefacts, from requirements documents/models, over designs to implementation. There may also be the need for stronger interactions between domain models, for example the need to support co-simulation between tools describing different behaviors (communication entities) and to support the allocation of functional (or software) models to hardware, when these are described in different tools. A typical scenario is also when a particular design model, e.g. a functional behaviour model, contains basic information that is required for many different types of analysis (e.g. allocation as described above, for safety analysis such as FMEA, or for formal verification purposes). In many cases there is a need to provide model

¹⁴ <http://www.tni-world.com/>

transformations, decide on appropriate model exchange formats and to deal with tool APIs. Representative model integration approaches, and their relation to multi-domain modelling languages are surveyed by [Chen, et al., 2006].

An example of a multi-domain tool environment for mechatronics products and with an emphasis on product development is CAMEL-View. It provides an interactive way to build up models of complex mechatronic systems which include different system domains, such as multi-body, hydraulic, control-engineering and discrete systems. With CAMEL-View the developer is supported by an extensible database comprising predefined components which is similar to Matlab/Simulink. The integration of models taken from Matlab/Simulink and other tools is also supported. For today's design it is also important to include the 3-D graphical description which is a core feature of CAMEL-View. The graphical description can be imported from numerous CAD systems, like OpenInventor, VRML, DXF or IGES. They are reduced automatically for animation purposes. For the analysis the specific model can automatically be transferred to a mathematical representation and to optimized C code especially for real-time simulation and Hardware-in-the-Loop applications. Besides this, CAMEL-View allows to export the model to Matlab/Simulink as an mdl-file including s-functions [iXtronics, 2006].

3.3 Discrete-event systems

This class of tools focuses on event-triggered dynamic systems. In this category there are tools both from the control system community as well as from the computer science community, sometimes allowing limited modeling and analysis of hybrid systems. There are also related tools developed within the Petri Net community.

The tools typically provide simulation, while some of them also provide facilities for formal reasoning about so called liveness and safety properties. A liveness property refers to a condition that will eventually come true. A safety property is a hazardous condition that is desirable to avoid; formal analysis of safety properties can thus be very important for systems with strict safety and reliability requirements (this type of verification is strongly related to analytical verification as discussed in Section 2.3).

There is a multitude of tools supporting discrete-event systems including UML tools and some control engineering tools (e.g. Stateflow as part of Simulink). Extensions of pure state machine formalisms to handle time include timed automata. Timed automata have been used for modeling hybrid systems and real-time systems. It is possible to use such automata for modeling the environment, the application software as well as the system platform. Timing analysis problems can be formulated and solved using model checking [Bouyssounouse and Sifakis, 2005].

Discrete event systems are often used as a basis for developing application specific modeling and analysis capabilities, for example for analysis of network protocols or real-time software. In some cases they have also been extended to included continuous-time modeling capabilities.

3.4 Networking tools

There is a broad range of tools that support modeling and analysis in the area of networking. One class of such tools is typically built upon discrete event simulators,

making it possible to model and simulate the behavior of communication protocols. One example of such a tool is NS-2, frequently used in telecommunication applications.

Networking simulators have been extended to incorporate continuous dynamics (to model plants), thus effectively reaching into the hybrid systems area. There is a close relation to the real-time scheduling and computing area; when dealing with distributed systems the approach taken with resource management (e.g. scheduling) has to cater for both computing and networking resources. This area covers those approaches with a starting point from the network.

Other tools under this heading are dedicated to the configuration of communication systems, including signal interfaces, protocol handling of data and communication scheduling. Examples in this category include tools from Volcano communication technologies¹⁵, Vector Informatik¹⁶ and TT-Tech¹⁷.

As an example of tools supporting networks, and in particular distributed systems design, consider the tool-chain provided by the company TT-Tech. This primarily targets distributed systems that are based on the so called Time Triggered Protocol (TTP). The tool-chain also provides an interface to Simulink from the Mathworks.

Once the control application is designed and tasks are assigned to the nodes of the system, the TTP communication messages that need to be exchanged must be defined. The designer completes the cluster design process by configuring the communication system (e.g. TDMA round duration, transmission rate, type of communication controller). All design data created above can subsequently be used by the cluster design tool for TTP-based systems. This tool constructs the TDMA communication schedule and stores it in a message descriptor list which includes the entire configuration of the communication schedule. This configuration is loaded into the communication controller in the implementation phase. The node design divides the application algorithms of the subsystems into tasks and specifies them. Configurations for certain operating systems can also be defined. It is then possible for the designer to invoke a Simulink code generator to produce application code for the tasks and to download it.

3.5 Real-time scheduling and computing

In the real-time research community, a number of tools have been developed for modeling and analysis purposes. These tools allow multi-tasking systems, and sometimes distributed systems to be modeled and their timing behavior to simulated, analytically assessed or schedules to be generated. For early examples of such tools see Audsley et al. (1994) and Storch and Liu (1996). An example of a more recent tool from this area that has been extended continuous dynamics capabilities is the RTSIM tool, described in more detail in section 4.

When dealing with distributed systems, there is also a need to define the allocation of functions to the nodes of the system and the partitioning of this functionality into

¹⁵ <http://www.mentor.com/products/sm/volcanoautomotive/>

¹⁶ <http://www.vector-informatik.com/>

¹⁷ <http://www.tttech.com/>

tasks. The AIDA tool-set is an example of a tool that supports such architectural design and that moreover has been integrated with control design tools in order to evaluate the resulting control performance. The AIDA tool-set is described in section 4.

AIRES¹⁸ - Automatic Integration of Real-time Embedded Software – is a tool prototype developed within the MOBIES project at the University of Michigan. AIRES was developed to support the analysis of timing and schedulability of embedded software. The tool also has some synthesis capabilities.

In the AIRES tool, the embedded software under development is represented with a component, software architecture and run-time model. The AIRES tool can be used either as a stand-alone analysis and design assistance tool or along with other design tools. The implementation of AIRES includes a graphical modeling environment, a meta-model, and analysis packages. The Generic Modeling Environment (GME) was chosen as the graphic modeling environment for AIRES. The meta-model is implemented as a modeling paradigm in GME and exported as an XML file for sharing among different tools. The analysis algorithms perform component allocation, timing assignment, priority assignment, schedulability, and end-to-end response time analysis. In the prototype, interfaces to design models in Rational Rose and Matlab Simulink/Stateflow have been implemented. Two flavors of the AIRES tool have been implemented: one for avionics applications and one for automotive applications

Another approach to real-time systems programming is represented by Giotto [Giotto, 2003]. Giotto as well as xGiotto have been developed at UC Berkeley from 2000 to 2004, see [Giotto, 2006] for links to a number of papers on Giotto and its tools.

GIOTTO focuses on distributed embedded control systems and as a programming language has a basis in the notion of logical execution time (LET). Giotto programs connect periodic software tasks to sensors and actuators, and specify the exact times when sensors are read and actuators are updated, independently of the number, speed, and utilization of host computers. The duration from reading a sensor to updating an actuator is referred to as the LET of a task if the task is connected to that sensor and actuator. Equivalently, the LET of a task determines the time the task executes logically, i.e., the time the task takes from reading input to writing output, independently of the time the task actually computes. The execution of Giotto and LET programs in general is correct (time-safe) if all tasks compute in real time less or equal than their LET. The LET semantics explicitly distinguishes logical from physical task execution in order to trade off average-case performance for determinism. Even if a task computes in less than its LET, its output is delayed until its LET has elapsed. In the LET paradigm, more and faster host computers will provide more resources not to execute existing programs faster but to accommodate additional computation and programs without changing the real-time behavior of existing programs. LET programs are thus composable with respect to real-time behavior even on distributed systems. LET programming also supports model-based embedded software design in the sense that control models designed in, e.g., Simulink, that use LET semantics can be translated into LET programs written in, e.g., Giotto, and then compiled into executable code that approximates the behavior of the original models in real time. Giotto and other LET programs have been compiled

¹⁸ <http://kabru.eecs.umich.edu/aires/>

into portable real-time code targeting the Embedded Machine, e.g., to control a model helicopter, and into so-called schedule-carrying code for increased efficiency.

In Giotto, the LET of a task must be equal to its period, i.e., the task can only read a sensor at the beginning of its period and update an actuator at the end of its period. The advantage of this rather restrictive model is that checking time safety of Giotto programs is fast even in the presence of multiple so-called modes, and precise if all modes are reachable by mode switching. Giotto programs may specify modes and when to switch modes. Giotto modes are essentially different configurations of tasks, sensors, and actuators. Checking time safety of programs with multiple modes is non-trivial because modes may be switched even before the execution of a mode has been completed.

More recent work on extending Giotto has essentially focused on two directions. xGiotto supports mixed sets of time- and event-triggered LET tasks at the expense of checking time safety fast, [XGiotto, 2004]. HTL supports hierarchical program composition where task LETs may be different than task periods at the expense of checking time safety precisely, [HTL, 2006]. In other words, there is an expensive but precise time safety check for xGiotto, and a fast but imprecise time safety check for HTL, i.e., if the time safety check of an HTL program fails, the program may still be time-safe. However, if the time safety check succeeds, tasks in lower levels of the program's hierarchy may be replaced by other tasks without re-checking time safety. The ongoing work on HTL is a collaborative effort between UC Berkeley, EPFL in Switzerland, the University of Salzburg in Austria, and the Technical University of Timisoara in Romania. A prototype tool chain is available at <http://htl.cs.uni-salzburg.at>.

3.6 Safety and reliability

Tools with an origin in the safety community have evolved rather independently of the other mentioned tools and provide support for example for fault-tree, failure mode effects and hazard analysis. Tools supporting reliability analysis are on the other hand related to tools in the discrete-event area (compare for example Markov chains), and also have a strong connection to formal analysis tools.

There seems to be very few tools that address the intersection between control design and safety/reliability tools. Some exceptions in this regard include the following:

- Bridges between control design and safety tools were developed in the SETTA project, allowing failure modes to be defined within Simulink and export for the generation of fault-trees, [Papadopoulos et al., 2001].
- In work at KTH, based on the experiences with co-simulation [El-khoury, Törngren, 2001], a Simulink library was developed that supports fault-injection in terms of bit-flips in all types of blocks, signals and constants. The library has been used to evaluate the effects of transient hardware faults on control system robustness and in devising control algorithms that are resilient to such faults, [Norberg and Törngren, 2003].

However, there exists a lot of related work in industry with proprietary tooling environments for these purposes.

4 Overview of selected tools

This section is organized as follows. Sub-sections 4.1-4.8 describe the different tools mentioned previously. 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 (or semantics)
- Tool automation
- Extensibility
- Availability

4.1 AIDA

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 6 shows an example of a data-flow diagram with four functions and the related data flows connecting them.

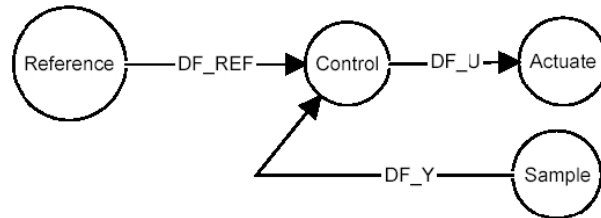


Figure 6. An example of an AIDA data flow Diagram

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 7 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 6 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 messages to CAN-frames; and triggering of process executions.

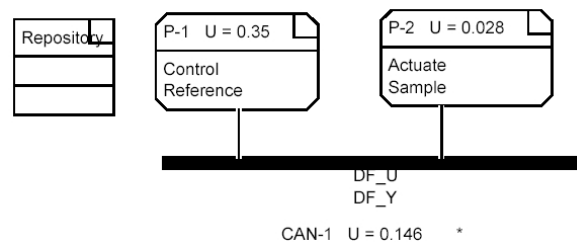


Figure 7. An example of an AIDA hardware structure diagram

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.

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

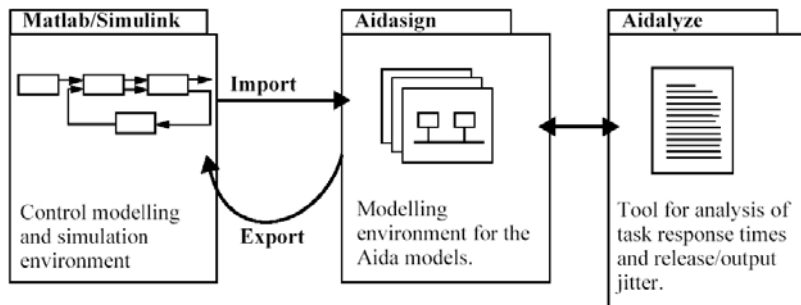


Figure 8. Architectural overview of the Aida toolset, highlighting the three parts: The interface with MATLAB/Simulink; the real-time system modelling environment (Aidesign); and the response time analysis tool (Aidalyze).

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 8 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 maximum execution times while a data flow is simply described with the number of bytes that it communicates. See Figure 6.

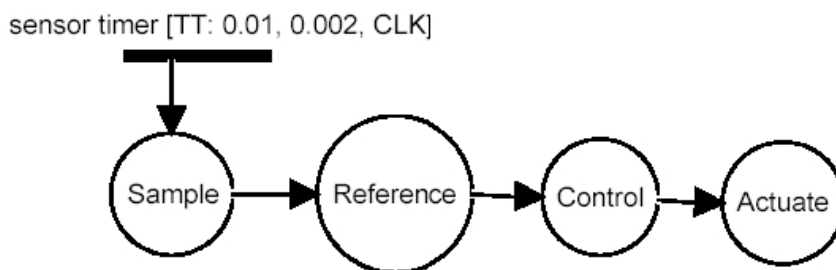


Figure 9. An example of an AIDA function timing and triggering diagram.

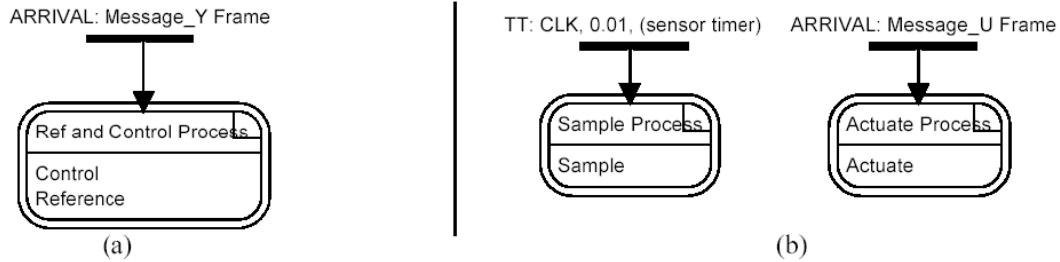


Figure 10 Process timing and triggering diagrams for processors P-1 (a) and P-2 (b) of Fig. 7.

- *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 9 shows an FTTD where the execution sequence of the functions in the example application of Figure 6 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 7.
- *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 10 shows the PTTDs for the two processors in Figure 7. 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.
- *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 11 shows the very simple PiTTD for the Ref and Control process executing in processor P-1.

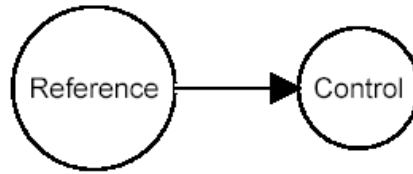


Figure 11 The process internal timing and triggering diagram of the Ref and Control process.

- *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 12. It defines two inter-process messages: *Message_U* and *Message_Y*.

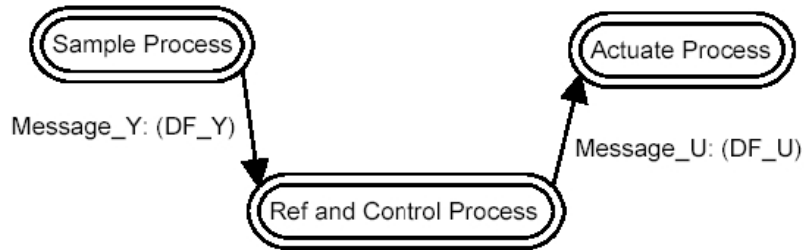


Figure 12 The process structure diagram for the example system.

- *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 13 shows how the messages defined in Figure 12 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 10.

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 Aidesign 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, Aidesign is easily extended with more models when needed. Furthermore, tools performing 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.

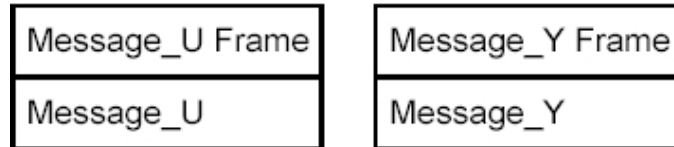


Figure 13 A communication link diagram defining the CAN-frames in the system.

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 Aidesign.

Availability Developed in-house KTH. Available upon request.

4.2 Jitterbug

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.

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.

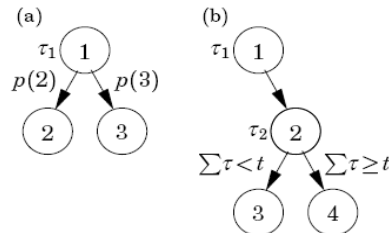


Figure 14 Alternative execution paths in a Jitterbug execution model: (a) random choice of path b) choice of path depending on the total delay from the first node.

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 15, 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 $H1$, $H2$, and $H3$. The system $H1$ could represent a periodic sampler, $H2$ could represent the computation of the control signal, and $H3$ could represent the actuator. The associated timing model says that, at the beginning of each period, $H1$ should first be executed (updated). Then there is a random delay τ_1 until $H2$ is executed, and another random delay τ_2 until $H3$ 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 15 are given in Figure 16.

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

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

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

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

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

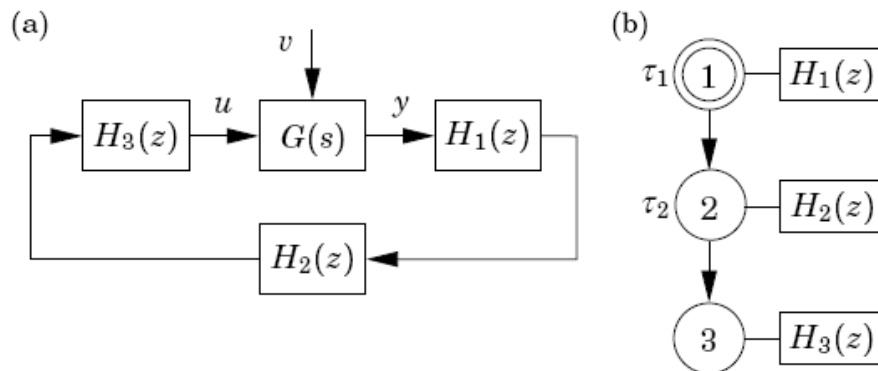


Figure 15 A simple Jitterbug model of a computer-controlled system: (a) signal model and (b) timing model. The process is described by the continuous-time system $G(s)$ and the controller is described by the three discrete-time systems $H1DzE$, $H2DzE$, and $H3DzE$, representing the sampler, the control algorithm, and the actuator. The discrete systems are executed according to the periodic timing model.

<code>G = 1000/(s*(s+1));</code>	Define the process
<code>H1 = 1;</code>	Define the sampler
<code>H2 = -K*(1+Td/h*(z-1)/z);</code>	Define the controller
<code>H3 = 1;</code>	Define the actuator
<code>Ptau1 = [...];</code>	Define delay probability distribution 1
<code>Ptau2 = [...];</code>	Define delay probability distribution 2
<code>N = initjitterbug(delta,h);</code>	Set time-grain and period
<code>N = addtimingnode(N,1,Ptau1,2);</code>	Define timing node 1
<code>N = addtimingnode(N,2,Ptau2,3);</code>	Define timing node 2
<code>N = addtimingnode(N,3);</code>	Define timing node 3
<code>N = addcontsys(N,1,G,4,Q,R1,R2);</code>	Add plant, specify cost and noise
<code>N = adddiscsys(N,2,H1,1,1);</code>	Add sampler to node 1
<code>N = adddiscsys(N,3,H2,2,2);</code>	Add controller to node 2
<code>N = adddiscsys(N,4,H3,3,3);</code>	Add actuator to node 3
<code>N = calcdynamics(N);</code>	Calculate internal dynamics
<code>J = calccost(N);</code>	Calculate the total cost

Figure 16 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.

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/>

4.3 ORCCAD

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) aspect 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 (TR), 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.

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:

- Basic actions (RT) are performed using periodic control loops;
- Multi-rate control is supported using communicating modules (6 predefined protocols)
- Higher level actions (RP) run on a discrete events time scale;
- The runtime code is assumed to run on top of a preemptive/ fixed priority kernel

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 run time code is automatically generated as a multi-tasks executive linked to the RTOS.
- Simulation and formal verification can be performed for validation.

Tool Architecture. The Orccad toolset consist of dedicated human- machine interfaces (module and RT editor for control laws specification and code generation). It also contains a HMI and code generator for the application specification (discrete event based spec. based on Esterel). Run time libraries for several off-the-shelf RTOS are provided (e.g. Linux/Posixthreads, RTAI...).

Tools based on Petri nets modelling and (max, plus) analysis allow for the structural and temporal verification of the network of synchronized control modules (assuming fixed given execution times for the modules) [Simon and Benattar, 2005].

The verification tools associated with the synchronous language Esterel allow for the formal verification of the system behaviour as well as its crucial properties, such as liveness and safety properties. Control purpose dedicated GUI have been written to help the control designer in the verification properties design and diagnosis.

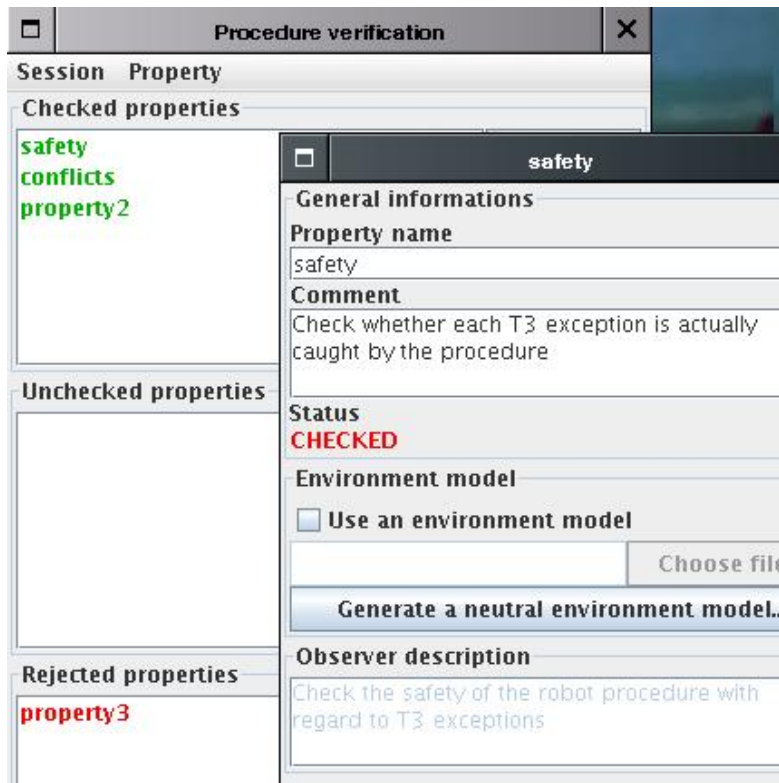


Figure 17 The robotic application: a two degrees of freedom arm.

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 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 18.

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.

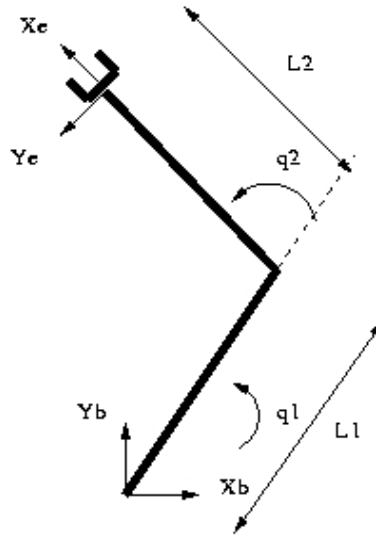


Figure 18 The robotic application: a two degrees of freedom arm.

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 *AppliArmX*. 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.

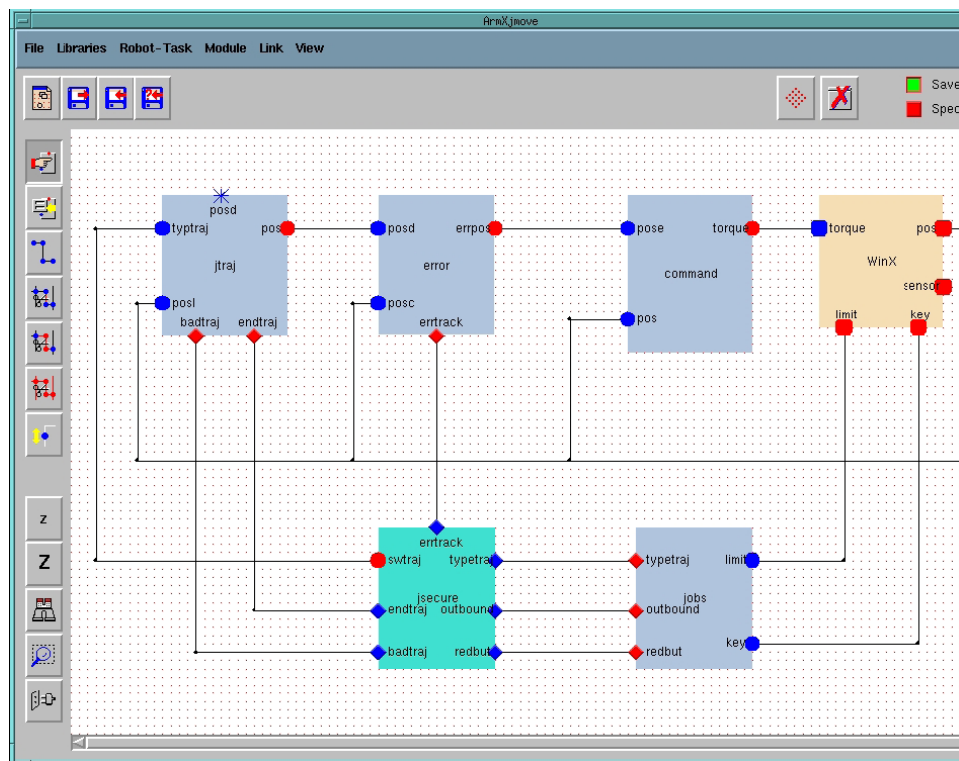


Figure 19 The *ArmXjmove* robot task.

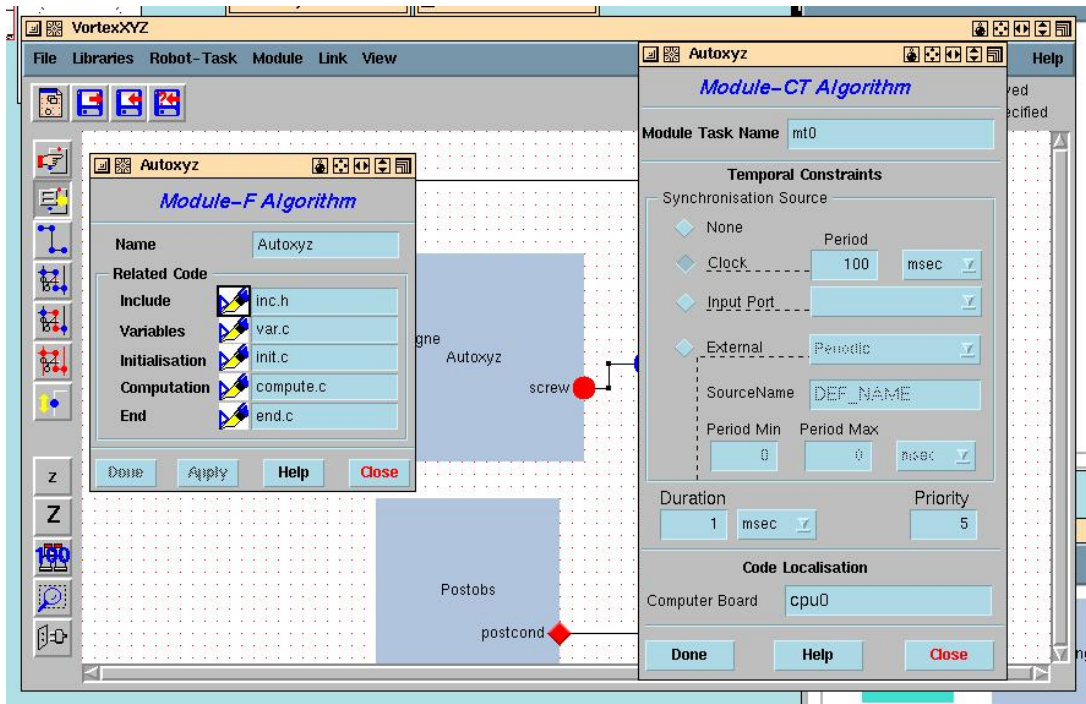


Figure 20 Functional and temporal spec. of a control module.

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 gathered and translated into Esterel which is then further translated into C code. Runtime libraries are provided to finally translate the systems calls and link all the C files (control laws and Esterel control automata) with the target RTOS.

New runtime *libs* can be rather easily written for preemptive/fixed priority RTOS, in particular for *Posix* compliant ones.

Extensibility. Real-time simulation can be provided by calling a numerical integrator (running the process ODE model) in the control task drivers (but requires enough computing power to be executed).

Flexible scheduling. During control design one of the module task can be customized to work as a feedback scheduler [Simon, Robert, 2005]. Its effective use requires some instrumentation at kernel level (e.g. threads execution on line measurement).

Availability. Unfortunately not longer freely available.

4.4 Ptolemy II

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 21.

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 are often represented graphically as in Figure 21.

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. Note that Giotto has evolved as a tool itself (see Section 3.5).

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 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.

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.

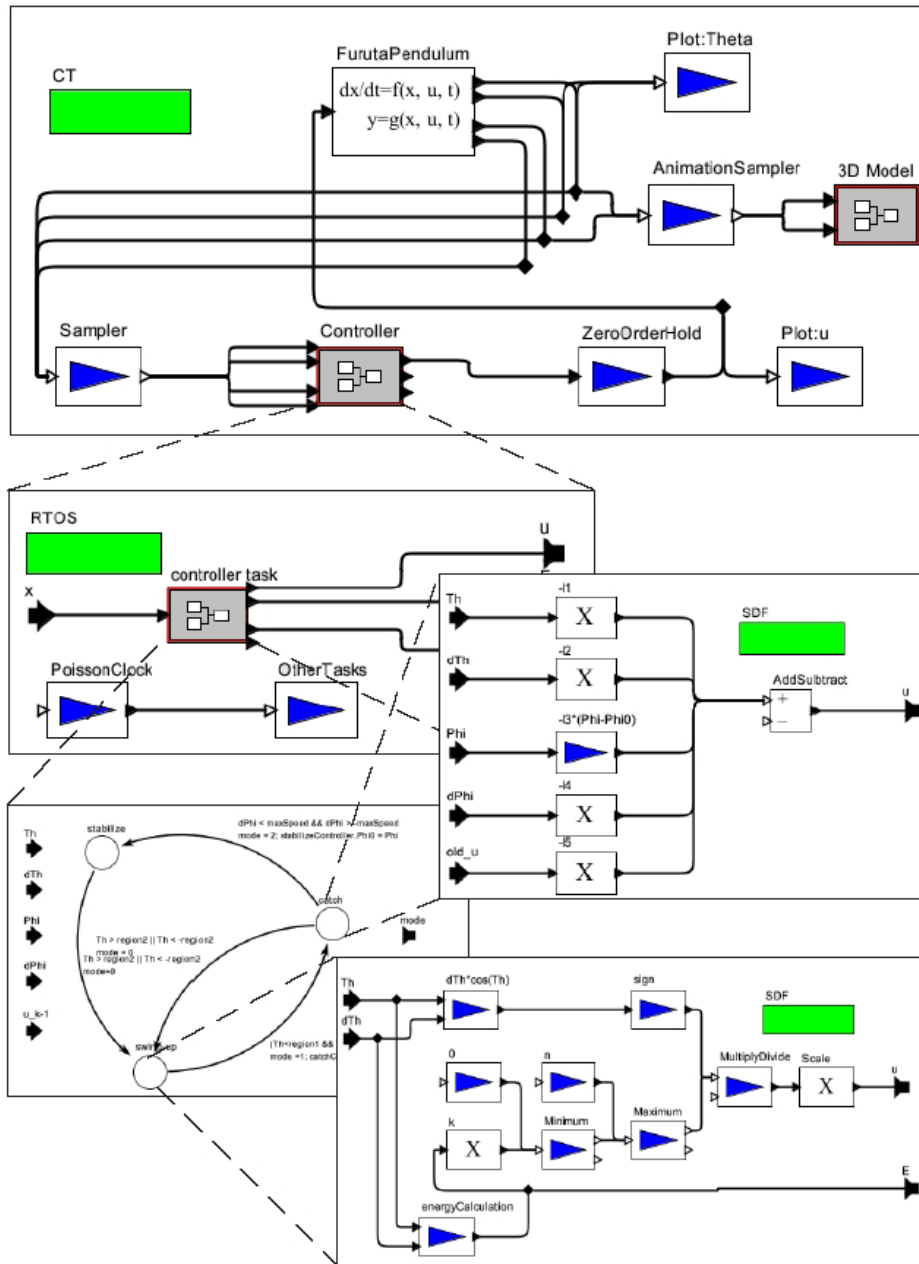


Figure 21 An integrated simulation model of an inverted pendulum process in Ptolemy II (from [Liu et al., 2002]). The top level contains actors for the pendulum process and the controller and utilizes the continuous-time model of computation. The controller is implemented as a task in the TM domain (here called RTOS). In addition to that, the different states of the controller are modeled as synchronous data flows (SDF).

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 _le 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>

4.5 RTSIM

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 HWSW architecture, see Figure 22.

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.

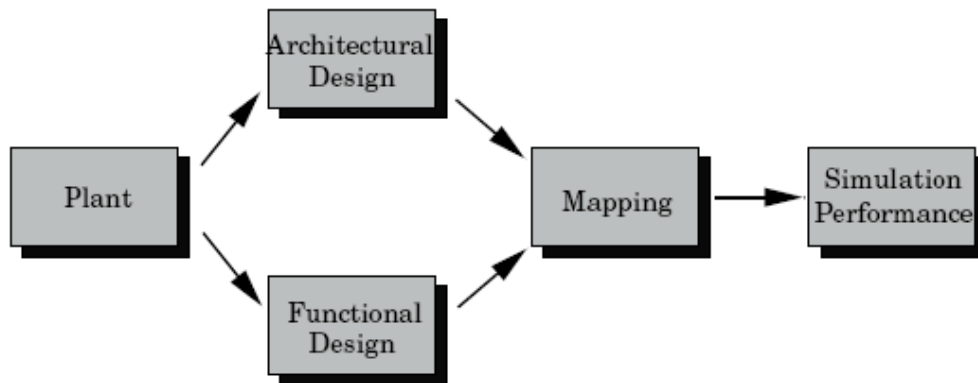


Figure 22 The design of a real-time control simulation using RTSIM.

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 23 shows an example of functional model of an inverted pendulum control system.

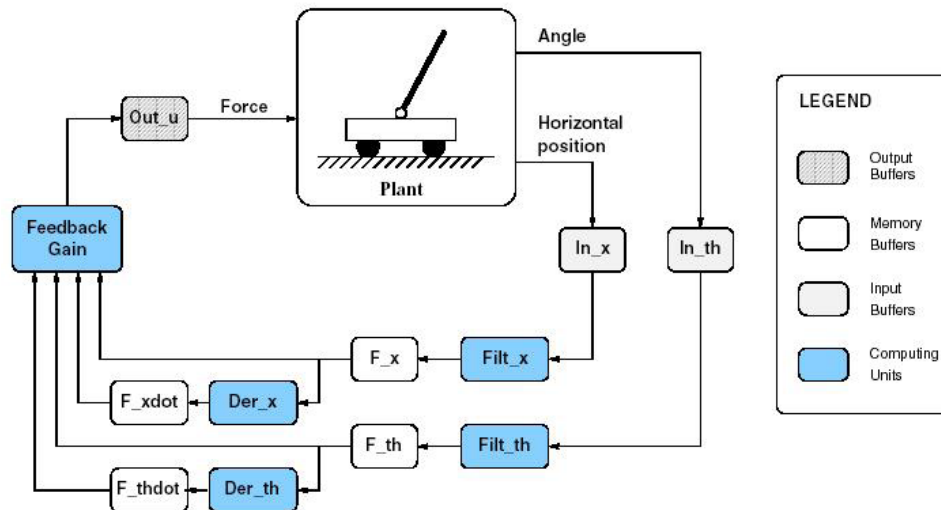


Figure 23 Example taken from [Palopoli et al., 2002] of a functional design for an inverted pendulum system. Input buffers are used to model sensors and output buffers to model actuators. Computing units exist for filtering and derivative actions and to compute the control signal.

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.

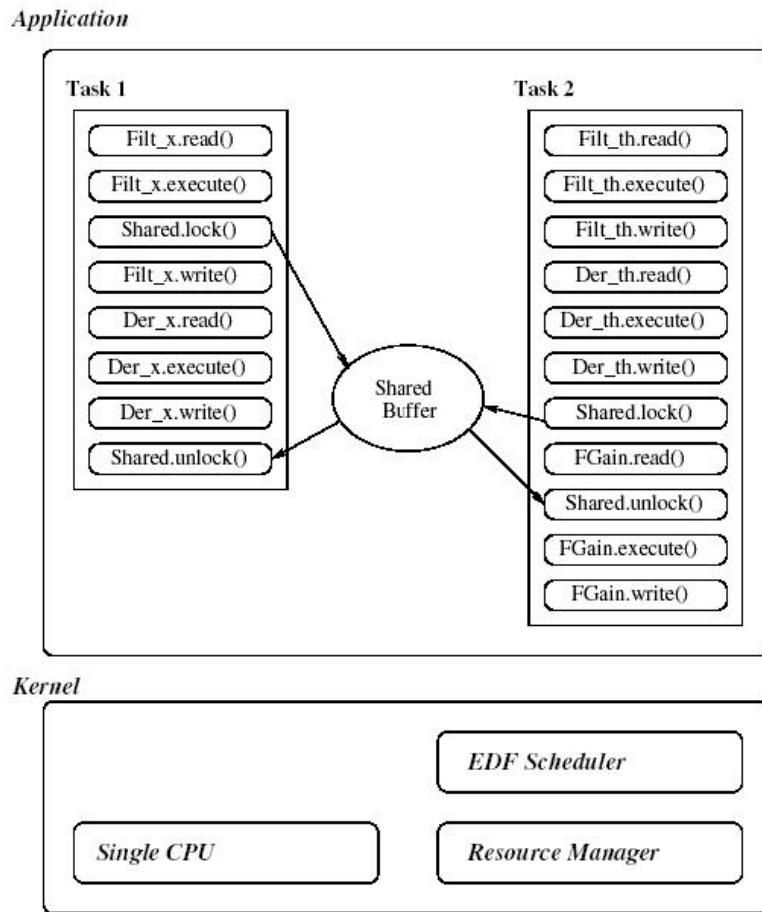


Figure 24 Example of an architectural design for the system from Figure 23. Here it is assumed that the horizontal position of the cart is obtained from camera images, whereas a potentiometer is used to acquire the angle. Therefore, the architecture uses two tasks for the controller computations.

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.

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 responses 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/>

4.6 SynDEx

Tool Overview

The SynDEx tool supports rapid prototyping of reactive application algorithms implemented on distributed heterogeneous hardware architectures [Pernet and Sorel, 2003; Grandpierre et al., 1999; Lavarenne et al., 1991; Forget et al., 2004]. SynDEx, based on the AAA methodology [Sorel 1994], lets the designer specify both the application algorithm and the distributed hardware architecture in a graphical environment, and then automates the mapping and scheduling of functions (called operations) and data-dependences between functions (called data-dependences) on the processors (called operators) and communication media (bus, link, crossbar, etc). During the mapping and scheduling process, which can also be manual, the hardware architecture as well as the application algorithm can be modified to better match the timing and resources constraints. When a sufficiently good solution has been found, SynDEx generates, using executive kernels depending on the processor type, executable codes that can be downloaded and run in real-time onto the distributed target.

Algorithms are specified in SynDEx as conditioned data-flow graphs that are indefinitely repeated. The graphs describe directed data-dependences (edges) between operations (vertices), and thus forms a directed acyclic graph. The graphs are conditioned because there may be sub-graphs of operations that are only executed when some value occurs on a specific conditioning input of the sub-graph. This mechanism is equivalent in the data-flow model to the control structure *If...Then...Else* or *Case...Of...*. In addition some sub-graphs of operations may be finitely repeated a certain number of times. This mechanism is equivalent in the data-flow model to *For i=1 to N Do...*. An operation can be hierarchically decomposed into sub-graph of operations. Non decomposable operations are called atomic. The algorithm model has a formal semantics equivalent to synchronous language Signal semantics, and can therefore be verified with tools for this purpose.

Figure 25 shows the specification of a very simple algorithm, *algorbasic*, that contains two constant operations (*cste1* and *cste2*) that represent constant integers. Two sensors produce constant data for two operations (*add* and *mul*) that perform addition and multiply, each of them produces a data for an actuator operation (*visuadd* and *visumul*).

Hardware architectures are also specified as graphs but that are not directed. Each architecture graph consists of two types of components interconnected via edges representing bi-directional connections. A component may be either an operator (processor, FPGA, ASIC), which sequences operations, or a communication medium, which sequences data-dependences. Figure 26 shows an architecture example, *biProc*, that consists of two processors (*root* and *pc1*) interconnected via a communication medium (*link(uTCP)*).

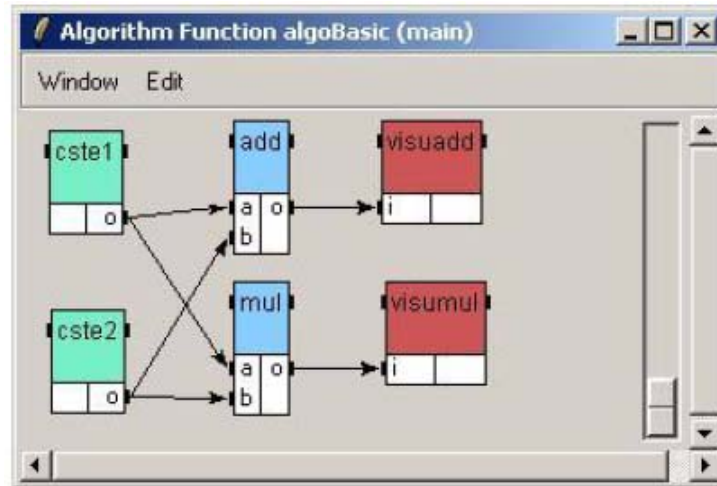


Figure 25 The algorithm graph *algoBasic*.

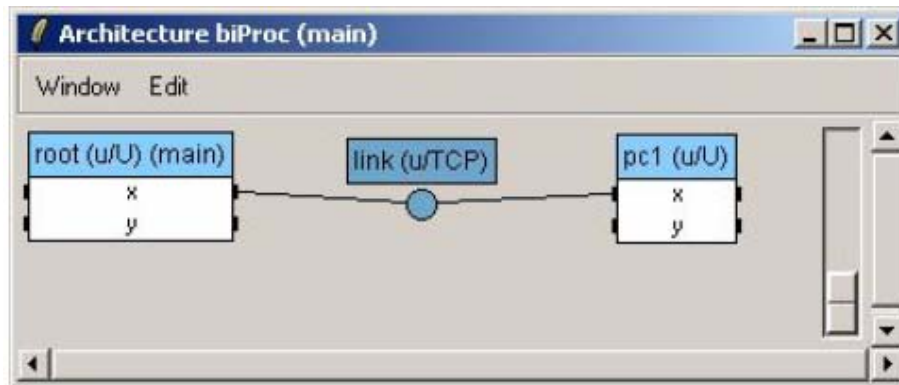


Figure 26 The architecture graph *biProc*.

The automated adequation supported by SynDex is performed by a heuristic, because the corresponding problem is of NP-hard complexity, that both maps operations (resp. data dependences) to operators (resp. to communication media), and schedules the operations and data-dependences on their respective components. It is based on a multi-periodic scheduling and mapping analysis which aims on the one hand at satisfying the real-time constraint, that is for each operation a deadline equal to its period, and on the other hand at minimizing the global execution time of the algorithm onto the distributed architecture. This heuristic utilizes the worst execution times (WCET) of operations and data-dependences, onto the operators and the communications media. Both types of executions are assumed to be indivisible. An operation may have several WCETs due to the different types of operator it may be mapped onto, it is the same for data-dependences (heterogeneous architecture). The multi-periodic scheduling and mapping analysis guarantees that the resulting implementation will maintain the partial order associated to the data-dependences of the initial algorithm graph leading to a distributed code executed in real-time without any deadlock. The result of the adequation heuristic, called an implementation model, is also a graph. It is visualized in a timing diagram that shows the parallel execution

and data transmission on all components in the system. Figure 27 shows a timing diagram of the schedule generated by the adequation heuristic 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 mapping constraints, that the designer may impose, were included to force some communication via the TCP communication medium in the simple example. Note that the constants do not appear in the timing diagram since they do not take any execution time.

Given the implementation model, SynDEX is able to automatically generate a macro-code independent of the architecture. The distributed executable code is built by the Gm4 macroprocessor from this macro-code and libraries of architecture-dependent primitives that compose executive kernels. One such kernel is needed for each supported processor type.

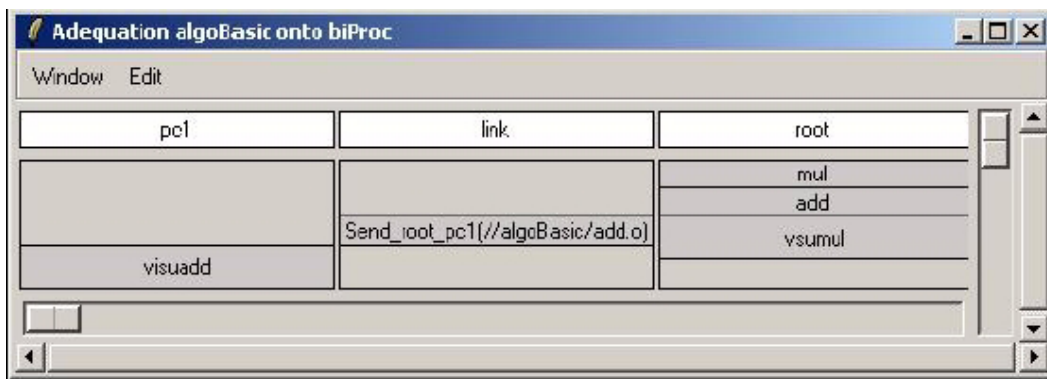


Figure 27 Timing diagram generated by the adequation algorithm.

Comparative Aspects

Scenarios Supported. SynDEX is intended to be used for rapid prototyping of application algorithms such as control, signal and image processing algorithms. The graphical interface is used to specify the algorithm and the distributed hardware. Then, when the automated mapping and scheduling has been performed, the designer has the possibility to modify the algorithm and hardware descriptions to better take advantage of the resources and thus reduce cost. The automated mapping and scheduling is performed after each modification and when the implementation has converged to a satisfactory result, executable code can be generated. The algorithm can also be formally verified using other tools suited for that purpose.

Development Stages and Activities Supported. The toolset is intended to be used in early stages when the application algorithm and 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 on the same architecture.

Qualities/Constraints Addressed. The automated mapping and scheduling step is mainly focused on finding a solution that optimizes the usage of available resources taking into account the timing constraints and the mapping constraints of operations and data-dependences to components that have been specified by the designer.

Methodological Considerations. The methodology supported by SynDEx is called AAA - Algorithm Architecture Adequation - and it follows the steps outlined above. These include: specification of the algorithm possibly through interfaces with domain oriented languages such as the synchronous languages (Esterel/SyncCharts, Lustre/Scade, Signal/Polychrony), Scilab/Scicos for modelling and simulation of hybrid dynamic control systems, and model driven graphical tools based on UML2.0 ; specification of the heterogeneous target hardware architecture ; and automated implementation of the algorithm onto the architecture, using the adequation process which involves spatial mapping and scheduling in time. Finally, an executable code may be generated, loaded and executed on the target hardware. This methodology on the one hand allows top-down or bottom-up approaches, and on the other hand to refine progressively the different prototypes to the final product. The formal semantics guarantees the distributed implementation is compliant to the algorithm specification, and possibly its previous simulations. These issues associated with automatic code generation decreases significantly the development life cycle of safety critical applications.

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 of operations, operators (processors, ASIC, FPGA) and communication media, including types for, e.g., mathematical operations and TCP communication links. The different steps : specifications, adequation (mapping and scheduling) and code generation are performed in the same environment, including a specific editor for creating new executive kernel dependent of the processor and communication media types. When an implementation has been fully completed the results of the different steps can be saved in a file which can be used as an input for a new implementation.

These issues guarantee consistency of the development life cycle.

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-dependences to be produced and consumed by the operations. The designer also needs to describe the target hardware including the processors and the interconnecting communication media. Each operation and data-dependence is given a duration (execution/transmission time) for each operator or communication medium available in the system. In addition for each operation a period is possibly given, presently the periods must be in geometric relation. When none operation has a period the heuristic aims at only minimizing the global execution time of the algorithm onto the target architecture.

Tool Outputs. The tool derives a mapping of operations and data-dependences as well as a schedule for each operator and communication medium presented as a timing diagram. Furthermore, executable code can be generated given that specific executive kernels have been developed for the different processors of the architecture.

Modeling Content. The algorithm is described as a directed acyclic graph that is executed repeatedly with a period equal to the least common multiple of the operation periods. 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 an algorithm graph can be hierarchically decomposed into sub-graphs. An operation may have several parallel sub-graphs and the selection of the sub-graph to execute for any given invocation, is controlled by a conditioning-dependence associated to the corresponding sub-graph of operations. Hence, a data flow graph can conditionally execute different sub-graphs on different repetitions. Also an operation may be repeated several time according to ratio between the dimensions of the inputs and their corresponding outputs. Furthermore, each operation and data-dependence is associated with one duration parameter for each possible operator or communication medium in the system. A period may also be associated to each operation.

SyncCharts a state diagram language that is similar to Statecharts but with a stronger semantics compliant with the deterministic real-time scheduling of SynDEx may generate algorithm graph representing the state diagrams specified and simulated with SyncCharts [Pernet and Sorel, 2003]. Thereby a SyncChart diagram after translation can be connected to a SynDEx algorithm graph already specified.

Scilab/Scicos a hybrid dynamic control system modeller and simulator, a free software similar to Matlab/Simulink, may generate a SynDEx algorithm graph for the specified models [Sorel 2005]. This allows to execute the hybrid dynamic models in real-time on a distributed architecture described with SynDEx being consistent with the corresponding simulation made with Scilab/Scicos on a working-station.

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 automates the distributed code generation 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 executive kernels.

Availability. SynDEx is available, with a complete documentation including user and reference manuals and tutorials, for download at <http://www.syndex.org>

4.7 TORSCHE

Tool Overview

TORSCHE (Time Optimisation of Resources, SCHEduling) is a MATLAB-based toolbox including various scheduling algorithms, that are used for various applications as high level synthesis of parallel algorithms, optimized production of manufacturing

lines, etc. Using the toolbox, one can easily and quickly obtain an optimal code of computing intensive applications running on specific hardware architectures. The tool can also be used to investigate application performance prior to its implementation and to use these values (e.g. the shortest achievable sampling period of the filter implemented on given set of processors) in the control system design process performed in Matlab/Simulink. The main contribution of the toolbox, which is built on well-known disciplines of the graph theory and operation research, is to make it easy to apply this type of reasoning to a wide range of problems. Many of them are combinatorial optimisation problems, and as such they are challenging from the theoretical point of view.

TORSCHÉ offers a collection of MATLAB routines that allow the user to formalize the scheduling problem, while considering appropriate configuration of resources (e.g. HW architecture performing filter algorithm), task parameters (e.g. deadlines, release dates, preemption) and optimisation criterion (e.g. makespan minimisation, maximum lateness minimisation). The toolbox enables to solve these problems by their reformulation or to solve them directly while choosing appropriate scheduling algorithm. The input data of the problem instance are typically represented by an oriented graph and the output data are represented by a Gantt chart. The input data might be automatically generated from the problem description (e.g. equations of the filter algorithm) and output data, the schedule, may be used to automatically generate an implementation of embedded system (e.g. parallel code for dedicated processing units).

Comparative Aspects

Scenarios and Development Stages Supported. TORSCHÉ is intended mainly as a research tool to design and evaluate different off-line scheduling algorithms. Use of this toolbox is double, first it can be used as it is to schedule instances of different problems, second the toolbox objects may be used to design new scheduling and optimisation algorithms. Further the toolbox may be used for more tailored applications of embedded systems that are interfaced to third party development tools (e.g. FPGA development tool chain).

Activities Supported. TORSCHÉ makes it possible to synthesize a schedule under various resource constraints, task parameters and optimisation criterions. The tool can also be used to perform response time analysis of fixed-priority scheduled tasks.

Qualities/Constraints Addressed. The task is given by the following parameters:

- Processing time, p_j , is time necessary for task execution. (called also Computation time)
- Release date, r_j , is the moment at which a task becomes ready for execution (called also Arrival time, Ready time, Request time).
- Deadline, \tilde{d}_j , specifies a time limit by which the task has to be completed, otherwise the scheduling is assumed to fail.
- Due date, d_j , specifies a time limit by which the task should be completed, otherwise the criterion function is charged by penalty.

- Weight expresses the priority of the task with respect to other tasks (called also Priority).
- Processor specifies dedicated processor at which the task must be executed.

Methodological Considerations. See scenarios.

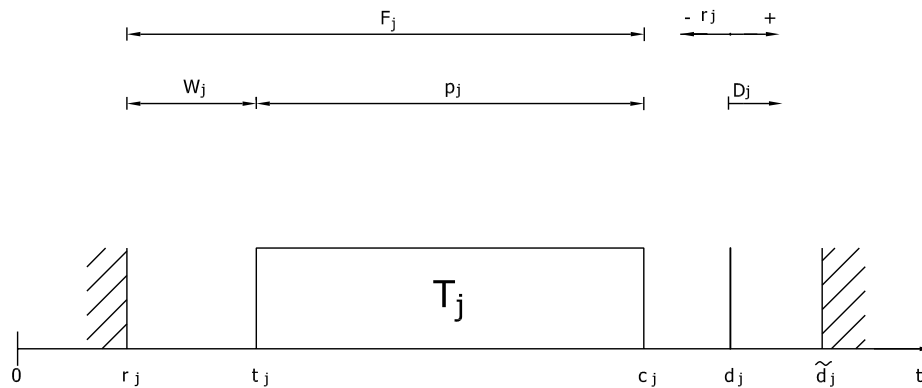


Figure 28 Task parameters

Tool Architecture. TORSCHÉ is written in Matlab object oriented programming language and it is used in Matlab environment as a toolbox. Main objects are *Task*, *TaskSet* and *Problem*. Object *Task* is a data structure including all parameters of the task as processing time, release date, deadline etc. Objects of a type *Task* can be grouped into a set of tasks and other related information as precedence constrains can be added. Object *Problem* is a small structure describing classification of deterministic scheduling problems in Graham and Blazewicz notation [Blazewicz, 2001]. These objects are used as a kernel providing general functions and graphical interface, making the toolbox easily extensible by other scheduling algorithms.

Tool Inputs. The task is represented by the object data structure with the name *task* in Matlab. This object is created by the command with the following syntax rule:

```
t1 = task([Name,]ProcTime[,ReleaseTime[,Deadline[,DueDate[,Weight[,Processor]]]]])
```

Command *task* is a constructor for object of type *task* whose output is stored into a variable (in the syntax rule above it is variable *t1*). Properties contained inside the square brackets are optional.

The object *problem* is a small structure describing the classification of deterministic scheduling problems in the notation proposed by Blazewicz et al. [Blazewicz, 2001]. An example of its usage is shown in the following code.

```
p = problem('P|prec|Cmax')
```

This notation consists of the three parts. The first part describes the processor environment, the second part describes the task characteristics of the scheduling

problem as the precedence constrains, or the release time. The last part denotes an optimality criterion.

Most of all algorithms use the following syntax:

```
tasksetWS = name(taskset,problem,procesors[,parameters])
```

Where

- *tasksetWS* is the input taskset with an added schedule
- *name* is the algorithm command name
- *taskset* is the set of tasks to be scheduled
- *problem* is the object of type problem
- *procesors* is the number of processors to be used
- *parameters* denotes additional parameters, e.g. algorithm strategy etc.

Tool Outputs. The schedule, assignment of tasks to processors in time, is generated as basic output of the tool. It might be displayed by simple *plot* function.

Modeling Content. As an illustration, an example application of RLS (Recursive Least Squares) filter for active noise cancellation is shown in Figure 29. The filter uses FP32, a library of arithmetic floating point modules for FPGA (for logarithmic arithmetic based solution see [P. Šůcha, Z. Pohl, and Z. Hanz'alek, 2004]). Addition, subtraction, multiplication, division and square root are executed by separate pipelined modules that require more hardware elements on FPGA, hence only one module of each kind is usually available for a given application.

RLS filter algorithm is a set of equations (see the inner loop in Figure 30) solved in an inner and an outer loop. The outer loop is repeated for each input data sample each 1/44100 seconds. The inner loop iteratively processes the sample up to the N-th iteration (N is the filter order). The quality of filtering increases with increasing number of filter iterations. N iterations of the inner loop need to be finished before the end of the sampling period when output data sample is generated and new input data sample starts to be processed.

The time optimal synthesis of RLS filter design on FPGA with FP32 is formulated as cyclic scheduling on the set of dedicated processors (like separate modules of FP32). The tasks are constrained by precedence relations corresponding to the algorithm data dependencies. The optimization criterion is related to the minimization of the cyclic scheduling period w (like in an RLS filter application the execution of the maximum number of the inner loop periods w within a given sampling period increases the filter quality).

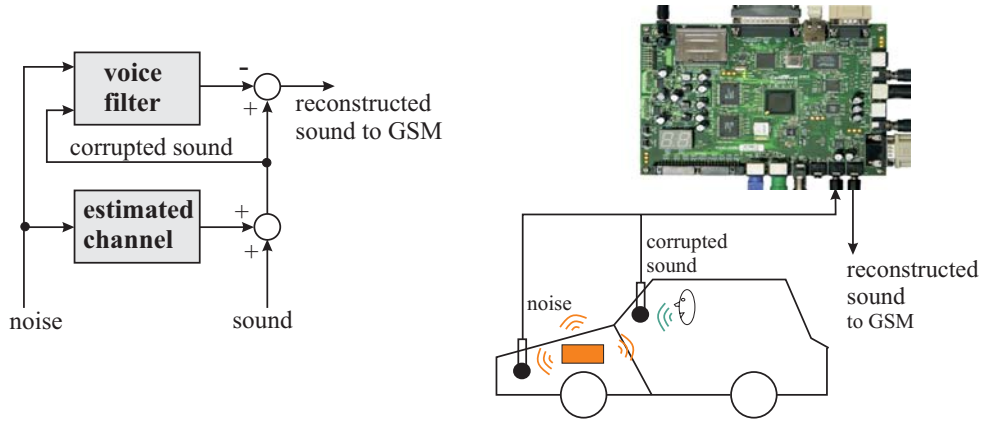


Figure 29 An illustration of active noise cancellation – an adaptive RLS filter estimates parameters of changing channel in order to reconstruct original clear sound.

Operations in a computation loop can be considered as a set of n generic tasks $T = \{T_1, T_2, \dots, T_n\}$ to be performed N times where N is usually very large. One execution of T labelled with integer index $k \geq 1$ is called an *iteration*. Let us denote by $\langle i, k \rangle$ the k^{th} occurrence of the generic task T_i , which corresponds to the execution of statement i in iteration k . The scheduling problem is to find a start time $s_i(k)$ of every occurrence $\langle i, k \rangle$. Figure 30(a) shows the inner loop of RLS algorithm.

Data dependencies of this problem can be modelled by a directed graph G , see Figure 30(b). p_i associated to node i is the processing time of task T_i . Edge e_{ij} from the node i to j is weighted by a couple of integer constants l_{ij} and h_{ij} . In fact, l_{ij} represents minimal distance in clock cycles from a start time of task T_i to a start time of T_j and it is always greater than zero. On the other hand, the height h_{ij} specifies a shift of the iteration index related to the data produced by T_i and consumed by T_j . Therefore, each edge e_{ij} represents the set of N relation constraints of the type $s_i(k) + l_{ij} \leq s_{j(k+h_{ij})}$.

In this model, the length of edge e_{ij} is greater or equal to processing time p_i assigned to node T_i . Therefore, the processor is occupied by the task T_i during processing time p_i , but the task T_j may start at least l_{ij} time units after the start time of T_i . Therefore, related length l_{ij} specifies the *precedence delay* from task T_i to task T_j (for more details on this issue see [Přemysl Šůcha and Zdeněk Hanzálek, 2006]).

The precedence delays are useful when we consider pipelined processors. The processing time p_i represents the time to feed the processor and length l_{ij} represents the time of computation. Therefore, the result of a computation is available after l_{ij} time units.

The corresponding task labels are indicated above each arithmetic operation in Figure 30(a). Figure 30(b) shows corresponding directed graph G . The schedule presented in Figure 31 was found by the cyclic scheduling algorithm based on iterative calls of ILP solver.

Figure 32 shows results of spectral analysis of the optimized RLS filter. The horizontal axis of each diagram represents the running time (corresponding to a time interval of 5s), the vertical axis represents the signal frequencies (up to 22kHz) and the colour represents the signal amplitude. The lower-right diagram presents the original sound, the upper-left diagram presents the noise, the upper-right diagram presents corrupted sound assuming sinusoidal changes of the estimated channel parameters, and finally the lower-left diagram presents reconstructed sound.

Tool Automation. Depending on the application scenario, the schedule can be automatically translated to the application formalism. Therefore TORSCHÉ automatically generates for example Handel-C code [Handel-C, 2005] in the case of FPGA design. Fig. 33 shows a piece of Handel-C code for the example shown above.

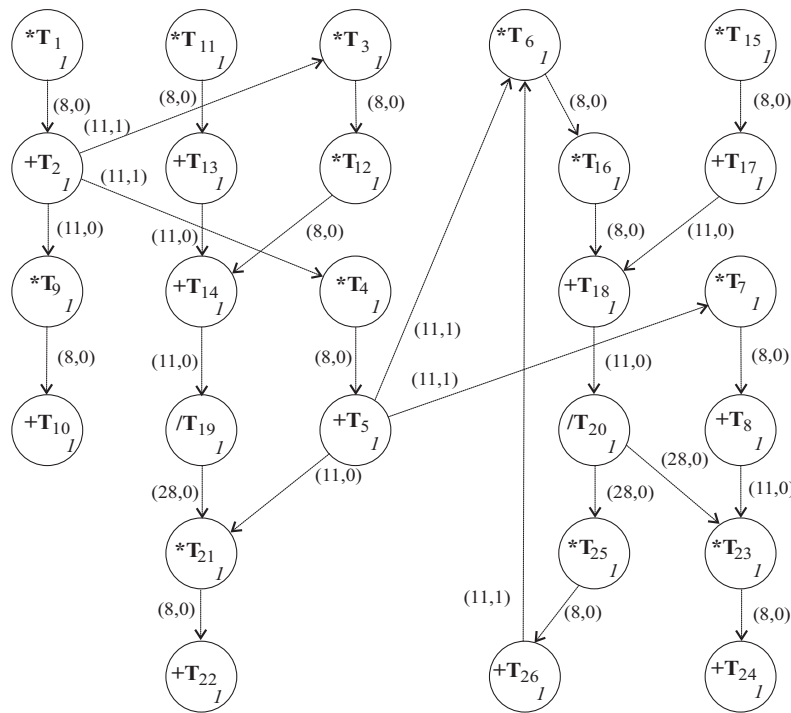


Figure 30 a) The inner loop of RLS filter. Constant N determines the filter order. **b)** Corresponding graph G .

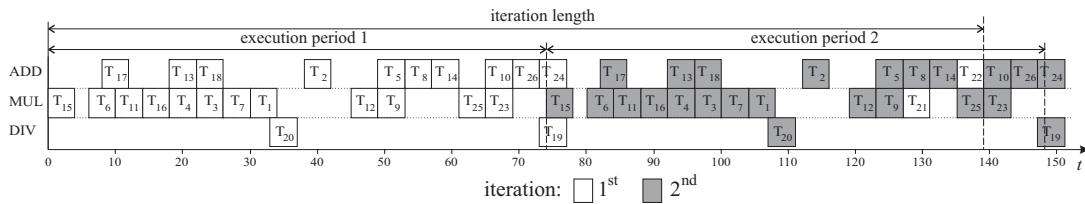


Figure 31 The schedule of the RLS filter inner loop (the period is 74).

Extensibility. Using predefined objects, the toolbox is easily extensible by various off-line scheduling algorithms implemented as Matlab functions/objects or C/C++ code.

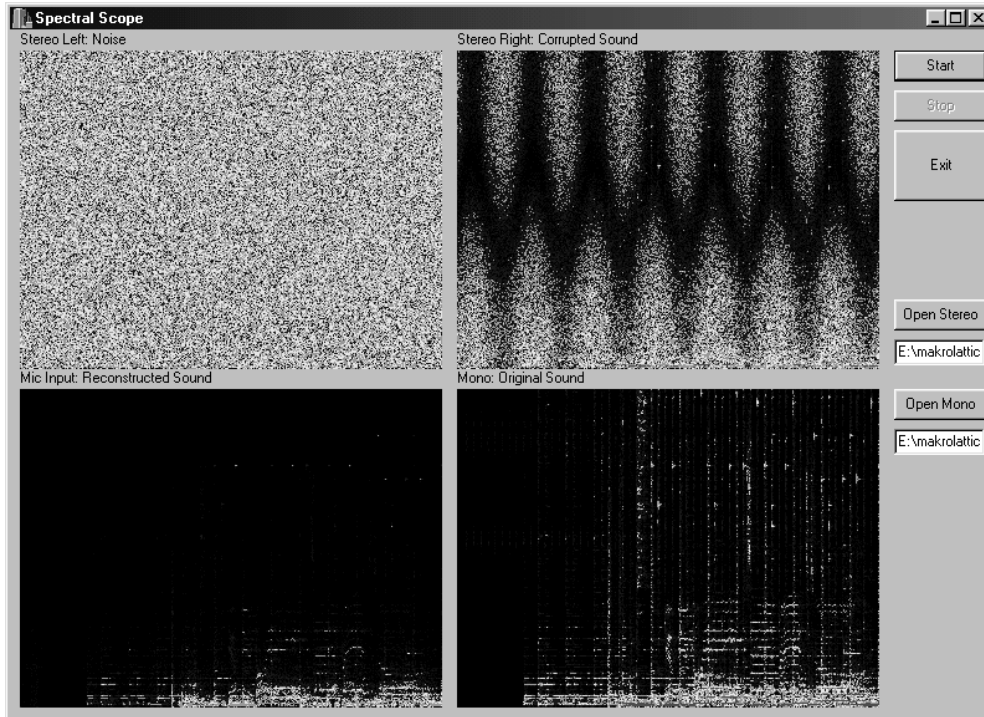


Figure 32 Spectral analysis of input and output signals of the optimized RLS filter.

Availability. TORSCHE is available for download at <http://rttime.felk.cvut.cz/scheduling-toolbox/>

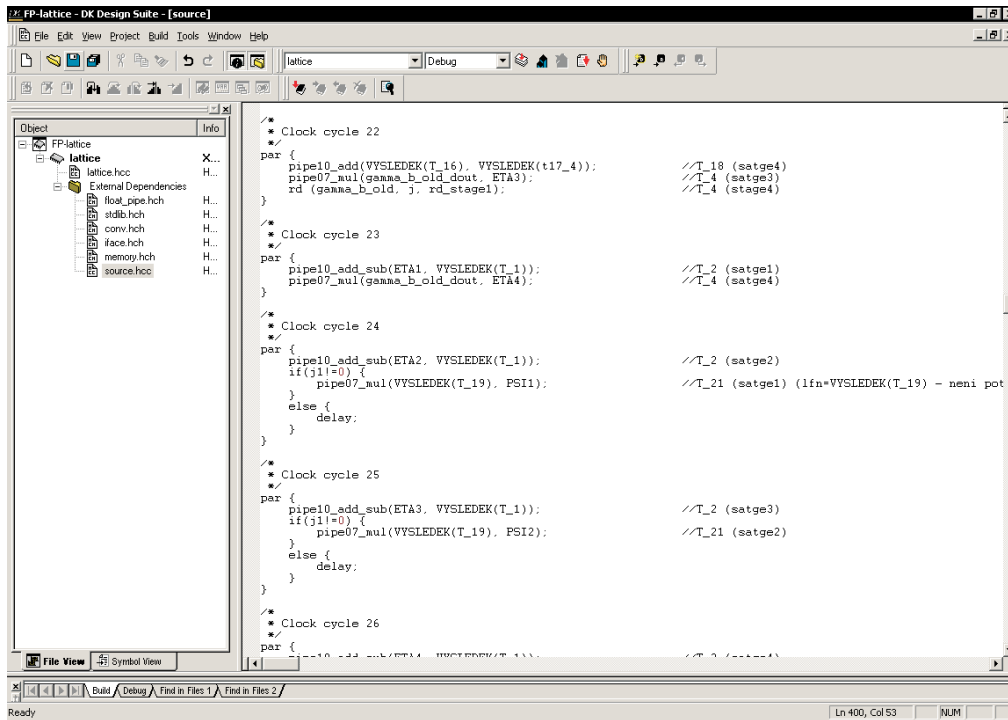


Figure 33 Generated Handel-C code

4.8 TrueTime

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 behaviour of a multitasking real-time kernel executing controller tasks. The tasks are controlling processes that are modelled 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 34. 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.

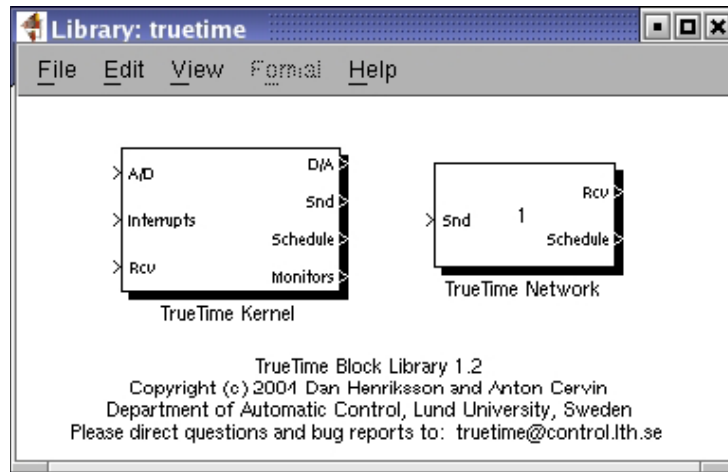


Figure 34 The TrueTime block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

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 34, 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 AD-DA 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 output 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.

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 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).

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 35.

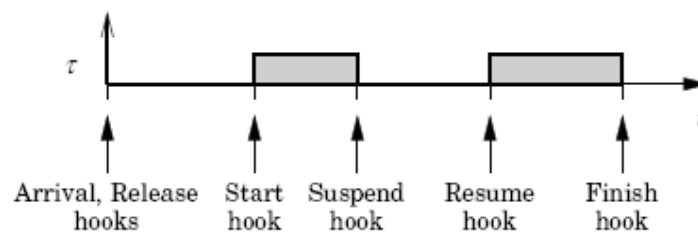


Figure 35 TrueTime scheduling hooks.

The network block is configured through the block mask dialog, see Figure 36. 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.

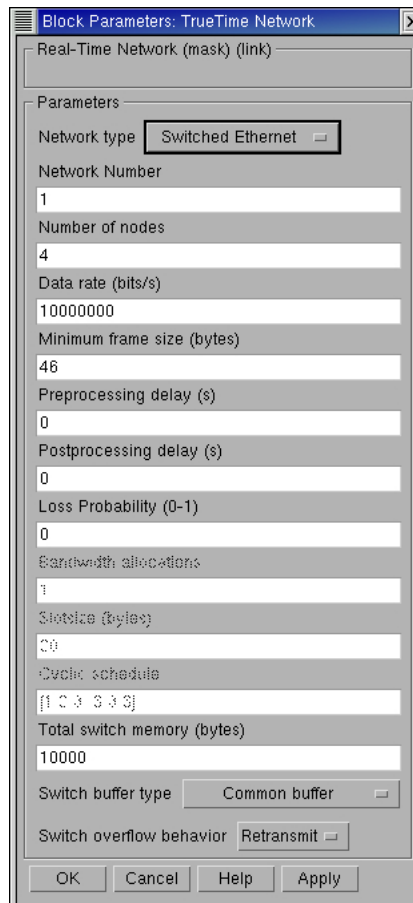


Figure 36 The dialog of the TrueTime Network block.

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.

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 task. 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 37.

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.

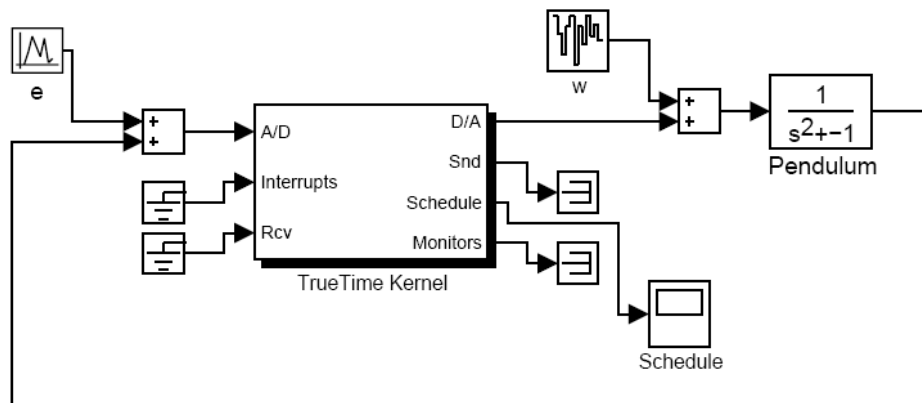


Figure 37 A TrueTime computer block connected to a continuous pendulum process.

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 37). 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.

Listing 1 Example of a simple TrueTime initialization function

```
function example-init
ttlnitKernel(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);
```

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

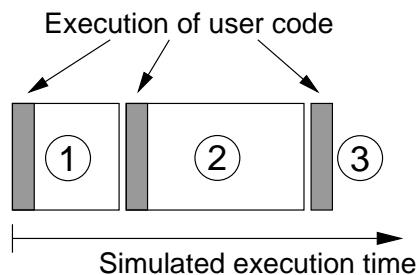


Figure 38 The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

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 executions 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 38. 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`.

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
```

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 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/>

5 Discussion: trends and challenges

To cope with the system complexity efficiently there has over the years been a constant trend to raise the abstraction levels at which the systems are being programmed and modeled. More than 20 years ago the programming of ECS was predominantly carried out using assembly languages. Then the paradigm shifted to high-level programming languages with the idea to provide programmers with more powerful tools. These tools would relieve the programmers of the burden of knowing the implementation hardware in detail, thus giving them the possibility to work more efficiently by focusing on the applications (a kind of separation of concerns). During this paradigm shift, concerns were raised whether the compilers would be able to produce efficient and reliable code. Entering the paradigm of model based development (MBD) the same concerns are now being raised with regards to code generation from models.

MBD related trends in embedded control systems include the incorporation of formal methods in tools, development of modeling guidelines, increased tool support for distributed systems and function/platform integration, and standardization of modeling languages, platforms and architectures.

Considering the requirements on embedded systems many efforts aim at improving the dependability of these systems. One way of reducing the probabilities for faults in a complex system is to reduce the manual labour and individual freedom included in the development chain. Manual labour is known to be error prone. This means that different initiatives to increase formalization are ongoing. For example, guidelines for C-coding have been issued by the automotive software organization, Misra¹⁹, and guidelines for modelling are being developed [Mathworks, 2004]. MBD has an important role in this dependability effort; code generation from models is seen as one way to improve software quality [Shigematsu, 2002]. Synthesis can be expected to be introduced at all levels (from drivers over RTOS to applications). The integration of application code generation with RTOS configuration is another closely related issue.

Contemporary computer aided control engineering (CACE) tools support a large body of control theoretical approaches for analysis and synthesis of dynamical systems (both continuous and discrete-time). In addition, because of the hybrid nature of control systems, computer science techniques for formal verification such as model checking are also emerging in CACE tools [Ranville 2004].

MBD tool support for ECS is today mainly limited to single processor systems, however support for distributed systems in model based environments is on its way as indicated by several research and industrial efforts. Such tools enable functions to be co-designed with the distributed computer platforms they are implemented on.

In general, supporting distributed systems development is a complex task where there are many challenges; still only partially tool support exists. Many of those challenges are less of a technical nature; not only are many protocols in use, but nodes part of a distributed system are developed by many different companies. Standardization and appropriate processes, including agreements between system integrators and subsystem suppliers are very important.

¹⁹ <http://www.misra.org.uk/>

Contemplating on the surveyed tools, the tools from related areas as well as the industrial tools, it does seem that extremely capable tool-sets could be formed by integrating many pieces of functionality available in the different tools. This would provide a range of capabilities with respect to modeling, analysis and synthesis, covering a large number of aspects (safety, control performance, power consumption etc.).

One approach to support co-design is that of formulating it as a synthesis problem, where, given design constraints, a solution is synthesized satisfying these constraints. This corresponds to a top-down approach. However, if the approach fails to find a solution for the given constraints, an iteration loop is required where some of the constraints (e.g. the control performance specification) are changed. Another more basic approach is to provide support for what-if type analysis where, for example, the control performance for a particular type of implementation can be evaluated, thus supporting solution space exploration. Of course, the approaches are complementary in that more advanced optimization approaches can be built on-top of the basic approach. It is perhaps representative that there are still few synthesis tools – this is a new research area.

There are several challenges facing the further development of the type of co-design studies in this report. One area for further research is that of filling in the theoretical gaps that today hamper co-design. Another area is that of providing support for the integration and management of all the different types of models being used in development.

Finally, the introduction of tool chains supporting model based control engineering is not unproblematic and is strongly related to and by affected by organizational, process and technology constraints. Introducing tool chains causes a reliance and dependence on particular tool vendors and requires training of personnel.

6 Conclusions

Designing a real-time control system is essentially a co-design problem. Choices made in the real-time computer system 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 important co-design tools available while at the same time characterizing the state of practice for industrial tools and tools in related areas.

The tools presented are in general specialized 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, and SynDEX 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.

7 Acknowledgements

This work has been supported to a large extent by the European Commission through the ARTIST2 Network of Excellence.

The authors would like acknowledge inputs and comments from the developers of the tools mentioned in this report. Additional inputs by Joachim Stroop for chapter 2, and by Achim Rettberg for a description of the CAMEL-View tool are acknowledged. Jianlin Shi, KTH, is acknowledged for providing feedback on and assisting in the editing of the manuscript.

8 References

- AADL – 2004. Architecture Analysis Description Language – SAE Standard AS-5506, Nov. 2004 (<http://www.aadl.info/>)
- ARTIST roadmaps – 2005. B Bouyssounouse,; J Sifakis, (Eds.). Embedded Systems Design - The ARTIST Roadmap for Research and Development Series: LNCS, Vol. 3436 2005, Springer.
- ARTIST2 – 2006. The ARTIST2 Network of Excellence on Embedded Systems Design. <http://www.artist-embedded.org/FP6/>
- ARTIST2 Control cluster roadmaps, 2006. Roadmaps produced by the Control for Embedded Systems Cluster within the ARTIST2 network of excellence. <http://www.md.kth.se/RTC/ARTIST2/publications.html>
- 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.
- Bhatt, D., V. Thomas, and J. Shackleton (1996): A methodology and toolset for the design of parallel embedded systems.. ACM SIGPLAN OOPS Messenger, 7.
- Blazewicz J., K. Ecker, G. Schmidt, and J. Weglarz (2001). Scheduling Computer and Manufacturing Processes. Springer, second edition, 2001.
- Bollella, G., B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull (2000): The Real- Time Specification for Java. Addison-Wesley.
- Carloni Luca, Maria D. Di Bebedetto, Alessandro Pinto and Albert Sangiovanni-Vincentelli (2004). Modeling Techniques, Programming Languages Design Toolsets and Interchange Formats for Hybrid Systems. Deliverable of the Project IST-2001-38314 COLUMBUS - Design of Embedded Controllers for Safety Critical Systems. http://www.columbus.gr/documents_public.htm
- 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 LUTFD2TFRT--7604--SE. Department of Automatic Control, Lund Institute of Technology, Sweden.
- Chen et al. (2006). DeJiu Chen, Martin Törngren, Jianlin Shin, Henrik Lönn, Sebastien Gerard, Mikael Strömberg, Karl-Erik Årzén. Model Based Integration in the Development of Embedded Control Systems – A Characterization of Current Research Efforts. In Proceedings of the 2006 IEEE Computer Aided Control Systems Design Symposium, October 2006. Accepted for publication.
- Control Systems Society (2004): CACSD history. Home page: <http://www.robotic.dlr.de/controlcacsdcacsdhistory.shtml>
- Crnkovic I., U Asklund. and DA Persson (2003), Implementing and integrating product data management and software configuration management, Artech House Publishers, 2003.
- EAST-ADL, 2004: Freund U., O Gurrieri, J Küster, H Lonn, J Migge, M-O Reiser, T Wierczoch and M. Weber (2004). An Architecture Description Language for developing Automotive ECUSoftware. INCOSE 2004.
- El-khoury Jad, Ola Redell, Martin Törngren (2005). A Model and Tool Integration Platform for Multidisciplinary Development. 31st Euromicro Conf. On Software Engineering and Advanced Applications (<http://www.idt.mdh.se/euromicro-2005>), Porto, Portugal, August 30th - September 3rd, 2005.
- 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.
- Handel-C. Language reference manual. <http://www.celoxica.com/>, 2005.
- Hristu-Varsakelis, Dimitrios; Levine, William S. (2005). Handbook of Networked and Embedded Control Systems. 2005, ISBN: 0-8176-3239-5.
- Henriksson, D. and A. Cervin (2003): TrueTime 1.1 Reference manual. Technical Report ISRN LUTFD2TFRT--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 MATLABSimulink.. In Proceedings of the Nordic MATLAB Conference. Copenhagen, Denmark.
- HTL (2006). A. Ghosal, T.A. Henzinger, D. Iercan, C.M. Kirsch and A.L. Sangiovanni-Vincentelli. A Hierarchical Coordination Language for Interacting Real-Time Tasks. Proc. ACM International Conference on Embedded Software (EMSOFT), ACM Press, 2006.
- 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 UCBERL M0325. Department of Electrical Engineering and Computer Science, University of California Berkeley, CA.
- Giotto (2003). T.A. Henzinger, C.M. Kirsch and B. Horowitz. Giotto: A Time-triggered Language for Embedded Programming. Proceedings of the IEEE, Vol. 91, No. 1, pages 84--99, Jan. 2003.
- Giotto, 2006. <http://embedded.eecs.berkeley.edu/giotto>.
- iXtronics (2006). www.ixtronics.de
- 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.
- MathWorks (2004). The MathWorks Automotive Advisory Board,

<http://www.mathworks.com/industries/auto/maab.html>, accessed Sept. 2005.

Motus L. and Rodd M. (1994). Timing analysis of real-time software. Pergamon Press. ISBN 0 08 0420257

Norberg, J. and M. Törngren (2003): Fault injection into control algorithms.. Technical Report TRITA.MMK 2003:37, ISSN 1400.1179, ISRN KTHMMKR-0311-SE. Department of Machine Design, KTH, Sweden.

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

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>.

P. Šúcha, Z. Pohl, and Z. Hanzálek. Scheduling of iterative algorithms on FPGA with pipelined arithmetic unit. In 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004), Toronto, Canada, 2004.

Přemysl Šúcha and Zdeněk Hanzálek. Scheduling with start time related deadlines. In IEEE Conference on Computer Aided Control Systems Design, Taipei, September 2004.

Ranville S. 2004. Case Study of Commercially Available Tools that Apply Formal Methods to a Matlab/Simulink/Stateflow Model. SAE World Congress 2004. Detroit, MI. March 8-11. 2004. SAE 2004-01-1765.

Reuter J. Analysis and Comparison of 3 Code Generation Tools. SAE World Congress 2004. Detroit, MI. March 8-11. 2004. SAE 2004-01-0702.

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 KTHMMK.989.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.

Shigematsu T. (2002). Software Quality Management Applied to Automotive Embedded Systems. SAE Convergence 2002. Transportation Electronics. Detroit, MI. October 21-23. 2002. SAE 2002-21-0017.

Simon D. and Benattar F. (2005), Design of real-time periodic control systems through synchronisation and fixed priorities. Int. Journal of Systems Science, Feb. 2005, Vol. 36, no. 2, p. 57-76.

Simon D., Robert D., Sename O. (2005), Robust control/scheduling co-design: application to robot control. RTAS'05 11th IEEE Real-Time and Embedded Technology and Applications Symposium, March 2005, San Francisco.

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.

- Sorel, Y. (1994): Massively Parallel Systems with Real Time Constraints, the Algorithm Architecture Adequation Methodology}. In Proceedings of Conference on Massively Parallel Computing Systems, MPC'S'94. Ischia, Italy.
- Sorel, Y. (2005): From modeling/simulation with Scilab/Scicos to optimized distributed embedded real-time implementation with SynDEX. In Proceedings of the International Workshop On Scilab and Open Source Software Engineering, SOSSE'05. Wuhan, China.
- 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.
- Premysl Sucha, Zdenek Hanzalek (2006). Scheduling of Tasks with Precedence Delays and Relative Deadlines, Framework for Time optimal Dynamic Reconfiguration of FPGAs. In IEEE International Parallel & Distributed Processing Symposium. New York: IEEE Press, 2006, s. 170. ISBN 1,4244,0054,6
- Törngren Martin, Mats Andersson, Björn Wittenmark, Jan Torin and Jan Wikander (2001). *Integrated Real-time Computer and Control System Architectures - DICOSMOS2 - final report*. Internal report, Department of Machine Design, KTH, 2001.
- Törngren, M., and Larses, O. (2005). Maturity of model driven engineering for embedded control systems from a mechatronic perspective. In Model Driven Engineering for Distributed Real-time Embedded Systems. Edited by: Sébastien Gérard, CEA, France Jean-Philippe Babau, INSA Lyon, Lyon, France Joel Champeau, ENSIETA, France. ISBN: 1905209320. Publication Date: August 2005.
- Törngren Martin, DeJiu Chen, Ivica Crnkovic, (2005). Component based vs. Model based development: A comparison in the context of Vehicular Embedded Systems. 31st Euromicro Conf. On Software Engineering and Advanced Applications (<http://www.idt.mdh.se/euromicro-2005>), Porto, Portugal, August 30th - September 3rd, 2005.
- Martin Törngren, Dan Henriksson, Karl-Erik Årzén, Anton Cervin, Zdenek Hanzalek (2006). Tools Supporting the Co-Design of Control Systems and Their Real-Time Implementation; Current Status and Future Directions. In Proceedings of the 2006 IEEE Computer Aided Control Systems Design Symposium, October 2006.
- 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.
- XGiotto (2004). A. Ghosal, T.A. Henzinger, C.M. Kirsch and M.A.A. Sanvido. Event-driven Programming with Logical Execution Times, Proc. International Workshop on Hybrid Systems: Computation and Control (HSCC), Springer, LNCS series, Vol. 2993, pages 357--371, 2004.