A DESIGN FRAMEWORK FOR REACTIVE AND TIME-TRIGGERED EMBEDDED SYSTEMS VIA THE UML-SYSTEMC BRIDGE

NGUYEN DANG KATHY

(B.Eng. (Hons.), Hochiminh City University of Technology)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE
2009

Acknowledgments

I am so blessed to be surrounded by many supportive and caring people. Without their guidance and support, this thesis would not have been possible.

First of all, I would like to thank my supervisor Prof. P.S. Thiagarajan for his patience, guidance and encouragement all these years. His valuable advice has helped me become better in various areas such as critical thinking, research writing and presentation skills and definitely made me a stronger researcher.

I also would like to thank my thesis committee members, Prof. Samarjit Chakraborty and Prof. Wong Weng Fai for their time and valuable feedback. Special thanks to Prof. Wong Weng Fai for always being helpful and supportive during my PhD. I would also like to acknowledge my earlier committee member Prof. Abhik Roychoudhury for his feedback during the initial stage of my research.

My senior Yang Shaofa and labmates RamKumar Jayaseelan, Dang Thi Thanh Nga, Pan Yu, Ge Zhiguo, Unmesh Dutta Bordoloi, Raman Balaji, Edward Sim Joon, Ankit Goel, Phan Thi Xuan Linh, Vivy Suhendra, Huynh Phung Huynh, Liu Shanshan, Ioana Cutcutache, Gu Yan, Deepak Gangadharan and Sun Zhenxin have been very kind and supportive. I will miss the time we discussed about work, had fun, and traveled to conferences together. Especially, thanks to Sun Zhenxin and Geoffrey Koh Yeow Nam for working with me as co-authors of some of my papers.

I appreciate Adam and Stuart for their advice which brought lots of changes in me. And I treasure the association with Poh Eng, Jack, Alvin, Ken, Sew Pheng, See and Shi Sheng. I get a lot of happiness, inspiration and experience every time we gather.

Thanks Lan Uyen for always listening to me. We have been friends for about twenty years and her unexpected appearance in Singapore during the last year of my PhD has brought me much joy and motivation.

And last but not least, I would like to express my deepest gratitude to my parents, my brother and my husband.

Con khong the nao noi het duoc long biet on cua con doi voi ba me. Con duoc den ngay hom nay la nho ba me. Cam on gia dinh luon tin tuong va dong vien con. Con luc nao cung tu hao ve gia dinh minh.

Cam on Bear da o lai Vietnam thay chi cham soc cho ba me (va Cucky nua) de chi yen tam hoc hanh. Luan van nay cua chi khong la gi neu so sanh voi nhung gi Bear da lam cho gia dinh, cong ty va tat ca dong su, nhan vien cua Bear.

Ong xa oi, anh da luon o ben canh em tu ngay dau tien em sang Singapore. Trong nhung luc em gap nhieu kho khan nhat hay hanh phuc nhat, anh van luon cung chia se voi em. Cam on anh da va se cung di con duong nay voi em.

Thanks all for being there in this wonderful and colorful journey.

Contents

Summary

Embedded systems are increasingly complex due to the large number of internal components and their interactions. This calls for more effective design methods. System level design methodologies have been proposed in this context as the means to cope with complex large scale embedded systems.

The aim of this research is to use UML notations to support system level design of systems in which control flow is event-triggered or time-triggered. We use SystemC as an intermediate representation to do design validation.

Our main contributions are:

- The identification of a subset of UML using which the structure, behavior and requirements for a system can be captured. In addition, we identify the necessary UML extension mechanisms and the level of abstraction to facilitate the efficient SystemC-based simulation.

- A translation framework in which the UML model can be used to generate SystemC code automatically. The generated SystemC code has been proven to offer good simulation speed.

- The first steps towards tool-supported model association in which UML-based test cases and requirements can be validated at the SystemC level and simulation traces can be displayed at the UML level.

- Case studies to confirm the efficacy of our design approach both in event-triggered and time-triggered settings.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Design of embedded systems

An embedded system-on-a-chip (SoC) is a single integrated circuit consisting of all the functional and communication components of a computing system. SoCs are being deployed on a broad range of applications such as multimedia, automotive, home control applications and consumer electronic devices.

A typical SoC may contain one or more programmable microprocessor(s), memories, peripherals such as timers, buses, external interfaces such as USART (Universal Synchronous/Asynchronous Receiver/Transmitter), SPI (serial peripheral interface), etc. In addition, there is software that runs on the microprocessor(s), operating systems and middle-wares.

The design of an SoC must fulfill strict requirements, regarding its functionality and performance but also non-functional requirements such as low cost, low power consumption, small size, and time-to-market. A good design method should aim to strike a balance between these design metrics. Most importantly, a design method should be able to help designers do initial analysis of a system's functionality and performance, before moving towards implementation and production.

Embedded systems are becoming increasingly complex due to the large number

of internal components and their interactions. Moreover, the components in a system often come from different suppliers, making their integration more difficult. This creates an enormous challenge for SoC design.

1.1.1 Conventional design methods

The conventional design methods for embedded systems usually have informal specifications. They cannot capture the requirements precisely and completely. Hardware components are refined from the specification to the implementation. Software components are subsequently implemented and integrated with the hardware components to get a complete system. It is only after this process that the functionality and design metrics are evaluated, and the design bugs are discovered. Once a bug is discovered, designers have to identify the design layer and the location of the bug within this layer and fix it. This usually consumes a large amount of effort.

Moreover, the partitioning of a system into hardware and software components depends on the intuition and experience of the system architects. Thus, design metrics often cannot be optimized.

Consequently, designers seek a design method in which specification can be captured more precisely and completely. In addition, the method should enable hardware/software integration and verification to be performed in the initial design phases so that design bugs can be discovered and fixed early. Moreover, designers would like to reuse previous designs to a large extent. This helps to reduce the non-recurring engineering cost of designing a system. Reuse and early hardware/software integration and verification can also help reduce design costs and shorten the time-to-market.

1.1.2 System level design

System level design is being pursued to meet the aforementioned requirements. The idea of system level design is to specify - without distinguishing between software and hardware realizations - the desired functionality with a coherent view of the whole

system. In this approach, functionality and design metrics can be evaluated at a high level of abstraction before the hardware/software partitioning occurs.

Designs at high levels of abstraction are simpler and more understandable compared to those at lower levels of abstraction. Thus, one can focus on important aspects of the design and create a system model quickly. The model will be easier to modify and maintain. Further, high levels of abstraction offer faster simulation speed although with some loss of accuracy.

The high level models are usually platform independent. Through refinement steps platform specific models are generated from higher level models for verification and implementation. Tools to support the required model transformation automatically are definitely needed.

Since system level design requires functionality of a whole system to be modeled, it is a challenge for designers. Graphical models enable designers to have a comprehensive visualization of the system under design, especially with the increasing complexity of embedded systems nowadays. In addition, graphical notations make it easier for designers to communicate with each other. Hence system specification is often captured using graphical models.

Often, it is not easy to reuse pre-designed components since their descriptions will involve many implementation specific details. . High levels of abstraction bring reuse to another level where the specification of the component being reused is independent on the detailed implementation. Thus reuse becomes more effective.

Another important aspect of system level design is the separation of computation and communication. Having different design blocks/modules for communication and computation components increases reusability and the simultaneous development of the components. We refer the reader to [86, 87, 62] for more detailed discussions on system level design.

In summary, system level design methods for embedded systems seem inevitable given the technological trends and the accompanying economic pressures. Two popu-

lar approaches for system level design are the Y-chart and its extension - the platform-based design.

The Y-chart based design

The Y-chart based design is usually used in the evaluation of alternative architectures [76, 124, 75]. It involves constructing the application (behavior) and architecture (platform) models separately. The behavior model is then mapped to the architecture model to build a performance model which will produce performance numbers. Based on the performance evaluation, designers can modify the application and/or architecture models or redo the mapping to improve performance. This procedure is repeated until satisfactory performance is obtained.

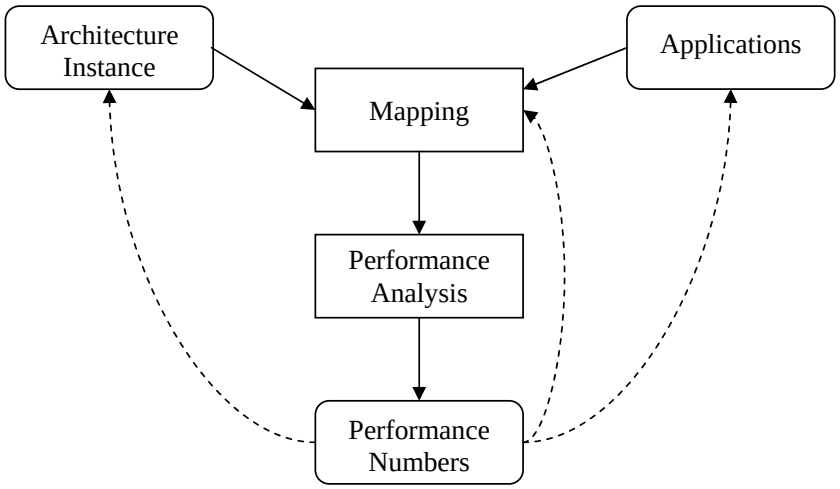

Figure 1-1: The Y-chart approach [76]

The Y-chart approach identifies three key aspects that play important roles in finding a suitable design, namely the application models, the architecture models and the mapping strategies. It enables the reuse of application and architecture models by having libraries of application and architecture components. In addition, it is a quantitative approach for architecture evaluation and design space exploration. Therefore, the Y-chart approach is a potential basis for a rigorous design methodology.

Another design approach which includes consecutive refinement steps, each of which is the spirit of the Y-chart methodology, is the platform-based design.

Platform-based design

Platform-based design (PBD) is a hierarchical design methodology that starts at the system level [36]. The top layers are the highest levels of abstraction where unnecessary implementation details are hidden. The design is carried out as a sequence of "refinement" steps that go from the initial specification towards the final implementation using platforms at various level of abstraction [108].

A platform is defined to be a library of components that can be assembled to generate a design. The library includes both communication and computation blocks. These blocks are supposed to have been verified earlier and are ready to be integrated through interfaces to form a model.

Fig. 1-2 shows the PBD process where the triangles on the left hand side represent the functionality and the ones on the right hand side represent the platforms. The functionality is mapped into a platform which consists of components chosen from the library. The mapped functionality in the middle is then considered the *function* in the next refinement step, which subsequently will be mapped to another platform. The process is repeated until all the components are implemented in their final form.

In order to apply the Y-chart and platform-based approaches, some important questions need to be answered. First, how to model the applications/functions and the architectures/platforms. The model must aim to satisfy the system level design's purpose, namely high levels of abstraction, design reuse and separation of communication and computation. Second, how to map the application model to the architecture/platform model, such that the mapped model can be validated and analyzed. And third how to do validation and performance analysis for the mapped application executing on the architecture/platform.

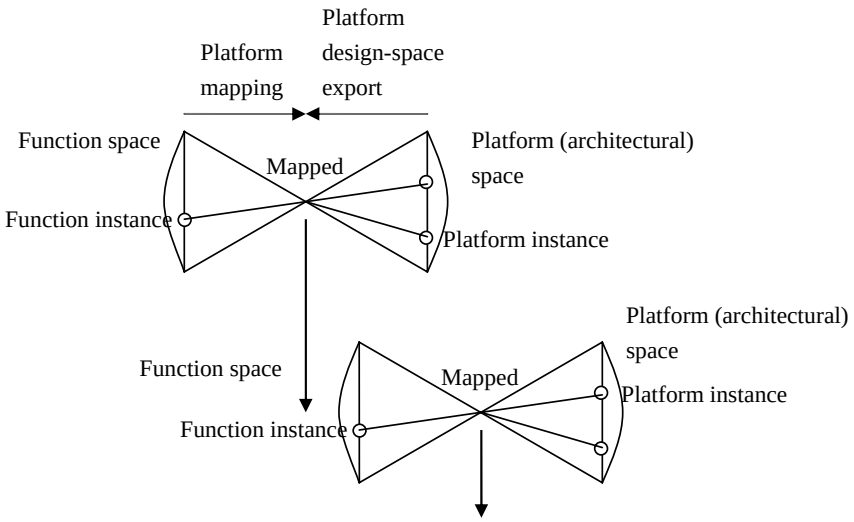

Figure 1-2: The platform-based design process [108]

1.2 Execution paradigms

In order to validate the design by simulation, the models need to be executable. There are two main execution paradigms for embedded systems: reactive (event-triggered) and time-triggered [126].

A reactive system is a computer system that interacts with its environment in an ongoing fashion. Moreover, the reactive systems changes its actions, outputs and conditions/status in response to external stimuli or trigger events. Thus they are considered to be event-driven systems. At the same time, the control and communication of each internal component are also triggered by events from the environment and other components. The points of time at which these events occur will be usually unpredictable.

In time-triggered systems, the steps of computation are triggered by the passage of time. In other words, all actions in a time-triggered system are carried out at certain predesignated points in time. Thus, time-triggered systems will have predictable temporal behaviors. Hence they are increasingly used in automotive, aerospace and railway applications, where failure to issue controlling commands promptly can have

disastrous consequences. However this requires that nodes in a distributed time-triggered system must have a common notion of time. This is achieved through synchronized clocks supported by a time-triggered communication platform which implements a time-triggered protocol.

Some systems integrate the two execution paradigms, thereby aiming at getting the best of two worlds: time-triggered dependability and event-triggered flexibility [126, 96, 114]. The differences in the dynamic behaviors of reactive and time-triggered systems demand different approaches in modeling and simulation for system level design.

1.3 Contributions

This thesis aims to explore the usage of existing standard notations and languages, namely UML and SystemC for system level design of reactive and time-triggered systems.

First, we identify the UML subset which can be used to capture the design of reactive and time-triggered systems and equally important, the levels of abstraction in which a system's structure and behavior should be captured. We base our choice of UML notations and diagrams on their ability to capture the aforementioned aspects of a system and to generate executable code so that validation can be carried out. We determine the roles and semantics of the chosen notations and diagrams in our model-driven system level design framework.

Second, we use SystemC as an intermediate representation for validation purpose. The SystemC executable representation can also serve as a high level model of the design before hardware and software components are implemented. We build a translator which translates the UML models to executable SystemC programs for simulation. To the best of our knowledge, ours is the first tool - at the time it was created [91] - which generates executable SystemC code from the UML model of a

system which includes both structure and behavior specification.

Third, we explore how to test the design at high levels of abstraction. We model the test case at UML level, and we generate a SystemC test driver. The test driver is actually a stimuli generator that triggers the model under test to carry out the test specified at UML level. In addition, the simulation trace is reflected back at UML level and compared with the requirement in order to guide the designers in their testing process. This is a first step towards supporting model association and model-based testing.

As mentioned earlier, we consider the design of both reactive (event-triggered) and time-triggered systems. They need to be treated differently in terms of the used UML notations, code generation and simulations approach. For event-triggered UML model, our framework synthesizes SystemC communication channels for event exchange. For time-triggered system, we support both the modeling of time-triggered applications and the lifting up of the communication platform to the high level UML model. A SystemC simulation driver is synthesized from a time-triggered UML model to speed up the simulation. Since we use UML for modeling and SystemC for intermediate representation in both settings, it should be easier to integrate the two execution paradigms.

Last, we also perform a variety of case studies to examine how our choice of UML and SystemC helps the design process. Our case studies include modeling a number of reactive and time-triggered applications and platforms. We also measure SystemC simulation speed and how our support for model association helps in identifying design bugs. Our results show that the UML-based design framework proposed here can significantly improve the design productivity. In addition, SystemC is an attractive intermediate representation. The translator from UML to SystemC built by us can serve as a tool in the high level to implementation toolset.

The framework established in this research will offer designers a means to do system level design by deploying standard notational systems and languages. Overall,

it should help designers of reactive embedded systems and time-triggered systems to bring system design to a higher level of abstraction, enhance reusability and thus improve productivity and decrease cost.

In this thesis, we do not aim to provide a comprehensive toolchain from high level design to implementation of hardware and software. Neither do we target high level formal verification, timing analysis and synthesis here. However, our framework will form a sound basis for adding these features or integrating with the tools that support them such as [82, 67] and [119].

1.4 Organization

Chapter 2 presents the overall view of our proposed design framework. We discuss the rationale for choosing UML and SystemC. We then present a UML subset which is used to capture the structure, behavior, and requirements for both reactive and time-triggered systems in our framework. Subsequently we sketch how SystemC is used to do simulation for functional and performance validation. This aspect of our work was presented in [89] and [90].

Chapter 3 describes how our framework can be applied for reactive embedded systems. We present how event-triggered computation and communication is modeled at UML level and represented at SystemC level. Then the generation of SystemC code from UML hierarchical state machines for effective simulation is discussed. These results in a preliminary form were presented in [91] and [92].

Chapter 4 shows how our framework can be customized and applied in time-triggered systems. The time-triggered applications are modeled at UML level. The communication platforms are also modeled by UML notations at a high level of abstraction. Through the case studies we show how the synthesized simulation driver significantly improves the simulation speed. The results of this work appeared in [93].

Chapter 5 presents how validation and model association can be supported. In

particular how test case and requirements are modeled at UML level, how a test driver is generated to facilitate the testing process and how the SystemC simulation trace is displayed and compared with the requirement at UML level.

Finally the last chapter presents a summary and discussion on directions for future work.

Chapter 2

System level design based on UML and SystemC

In this chapter, we first identify some of the important elements of a system level design framework. Among these, the crucial one is the high level modeling language for modeling and design. The second one is the intermediate representation that is needed for validation before further implementation. We start the chapter by reviewing system level design methods for embedded systems. Next, we present our proposed design framework and the suggested levels of abstraction to be used in this framework. Finally, we discuss our chosen high level design language-UML and the intermediate representation language-SystemC.

2.1 System level design frameworks

The need for system level design and its important aspects and challenges are discussed in [74, 108]. Some of the prominent system level design frameworks are Metropolis [19], Artemis [97], Spade [81], and Ptolemy [7]. Artemis supports the Y-chart approach and targets the multimedia application domain. Also following the Y-chart approach, Spade addresses the signal processing systems. On the other hand,

Metropolis was designed to support platform-based design in a unified framework.

First, we examine the modeling languages and notations. There are two trends of specification languages: one based on standard notational systems such as UML [48, 104, 124, 85] and its variations [9]; the other one based on new notations, such as MMM [19, 97], SystemC [10] and SpecC [95], etc. In Metropolis, the function and architecture are modeled in a representation called Metropolis Meta-Model (MMM) which like SystemC, separates communication and computation components.

In the Artemis framework, a sequential imperative application specification, written in a subset of Matlab, is converted into a Kahn Process Network (KPN). An architecture model is constructed from generic building blocks provided by a library, which contains template performance models for processing cores, communication media (such as busses) and various types of memory.

In Spade [81], the applications are modeled by Kahn Process Networks using which the computation and communication workload can be analyzed. An architecture is specified in a textual architecture description language. The mapping between applications and architectures is specified using textual language. We think that this approach is still at a low level of abstraction, where implementation-level architecture aspects such as instructions' latencies have to be specified.

We now consider simulation methods. The Metropolis framework contains a front end that parses the input metamodel language and creates an abstract syntax tree. In addition, it has a back end that translates the metamodel specification into the executable SystemC language. There is another back end which based on the simulation traces, examines whether the system satisfies some temporal properties defined in linear temporal logic (LTL) or the Logic of Constraints (LOC).

In Artemis, the simulation for performance analysis is done in Sesame [51] which is based on Pearl and an extension of SystemC. In Spade [81], the application simulation is based on Pamela [72] multi-threading environment. The architectural simulation model is based on TSS (Tool for System Simulation), a Philip's in-house architecture

modeling and simulation framework. A trace-driven simulation technique is employed to co-simulate an application model with an architecture model.

Ptolemy II [7] offers an environment for heterogenous reactive systems where different models of computation are supported for modeling and simulation. It is a Java-based framework and is able to generate C code for synchronous dataflow (SDF), finite state machine (FSM) and heterochronous dataflow (HDF) models.

In summary, many projects use SystemC for simulation [19, 97, 47]. Others use C and Java for simulation [7, 19]. [45] contains a survey of system level languages. We think that SystemC is able to support not only simulation but also multiple levels of abstraction; thus a SystemC intermediate representation can be further refined to implementation.

We next consider formal verification. The Metropolis framework has another formal verification back end which uses SPIN to verify LTL and a subset of LOC properties against a metamodel specification [108]. There have been works on formal verification of SystemC models at different levels of abstraction, from system level [79], transaction level [66, 73], to RTL level [63].

At UML level, there also have been research projects that involve formal verification. These works can be divided into two types, depending on the supported UML diagrams and what can be verified. First, several projects support UML class diagrams and Object Constraint Language (OCL). UMLtoCSP [30] translates UML class diagrams with OCL constraints to a Constraint Satisfaction Problem (CSP). The properties to be verified are mainly the relationships between the objects in the model. UML2Alloy [16] transforms UML class diagrams with OCL constraints to Alloy code. Alloy is a textual modelling language based on first-order relational logic [16]. It comes with the Alloy Analyzer which allows fully automated analysis.

On the other hand, there are projects that support the verification of UML class diagrams and state machines [18, 82, 109]. [18] transforms UML model consisting of class diagrams and state machines to a formal representation which can be

verified using TLPVS, an PVS-based implementation of linear temporal logic and some of its proof rules. However, orthogonal states and timed systems are not supported. The UML notations supported by vUML [82] are wider than those of [18]. They include composite states and transitions across composite states. vUML uses SPIN model checker to perform verification. The UML model is transformed into a PROMELA program. Also supporting the UML class diagrams and hierarchical state machines, [109] transforms UML models into a format usable for for VIS model checker. State space explosion is a common problem for these projects. Moreover, dynamic object creation, unbounded event queues and unbounded domain variables are also issues. Several techniques such as the symbolic analysis technique [38] and the abstraction technique of data-type reduction [39] are applied to eliminate these limitations.

We now turn to synthesis. The Metropolis framework not only offers simulation and formal verification tools but also supports refinement from higher levels of abstraction to lower levels. This platform refinement process is described in [43]. The refinement can go as far as "real" architectures such as the Xilinx Virtex II Pro [42]. In order to assure the refined models conform to the behavior of their abstract counterparts, formal refinement verification is provided [41]. The Artemis framework includes a synthesis path to VHDL code and FPGA implementation. In our framework, we have not supported synthesis to implementation. However, it is possible to connect our UML-based framework with others which provide this support.

Several frameworks are based on UML and SystemC. It will be more convenient to discuss them in a later chapter after presenting the main features of UML and SystemC.

2.2 Rationale for the UML-SystemC framework

In the recent past, a broad consensus has emerged regarding the basic principles that should govern system level design methods. These principles include that the design should start at a high level of abstraction and should deploy substantial component reuse. Moreover, they should separate computation and communication. As the complex systems usually have components containing each other and concurrent threads of execution, the modeling language must support the modeling of both hierarchical structure and concurrent behavior.

All of the above call for a high level modeling language which can be used as the starting point of the design process where the requirements as well as the system at high level are captured. The modeling language should serve as an easily understood common language among different parties involving in the system development process, including system architects, hardware and software engineers. At the system level, the modeling language must be able to capture different aspects of the system under design, namely the structure of a system and the behavior of its components, including how they interact with their environments and with each other. In addition, in order to support design validation, the modeling language must be able to capture test cases.

We have chosen UML as the modeling language to capture all the above. It is now widely accepted in the software engineering community as a common notational framework. It supports object oriented designs which in turn encourage component reuse. UML can be used to provide multiple views of the system under design with the help of a variety of structural and behavioral diagrams.

Though it was originally created to serve the software engineering community, UML is also becoming an attractive basis for developing system descriptions in the (real time) embedded systems domain [80]. In fact, many of the enhancements to the UML 2.0 are geared towards easing the task of specifying complex real time embedded applications.

UML allows standard ways of extending the language to meet the demands of specific application domains. It is a set of notations which is easy to learn and apply, which shortens the learning curve of hardware designers compared to other non-standard languages, thus saving time and effort. What UML may have to offer towards system level design methods for real time embedded systems has been studied from a number of perspectives as reported in [80].

It is necessary to do validation or simulation in order to validate the design. Due to the complexity of the systems we would like to validate, we have chosen simulation as the main approach. However, an UML design cannot be simulated without generating an executable program from it. In addition, an executable intermediate representation can serve as a common design document for the software and hardware teams which can then independently work towards a detailed implementation.

The intermediate representation should have a clean *executable semantics*, at which both the application and the platform on which the application is to be realized can be captured and related. Further, behaviors described at the intermediate level, should clearly separate the computational aspects from the communication features. It should also be compatible with the chosen modeling language above, namely UML.

SystemC has been chosen as the intermediate representation in our framework. It supports different levels of abstraction to capture a system from un-timed functional model to RTL (register transfer level). It allows both applications and platforms to be expressed at fairly high levels of abstraction while enabling the linkage to hardware implementation. Moreover, it separates computation and communication elements. Hence SystemC has the potential to provide a full fledged description of an execution platform which can serve as the target of a co-design methodology.

SystemC also supports multiple Models of Computation such as RTL model, Kahn Process Network, Static and Dynamic Dataflow. In addition, it supports different kinds of design process: top-down, bottom-up, and iterative. Furthermore, SystemC — viewed as a programming language — is a collection of class libraries built on

top of C++ and hence is naturally compatible with the object oriented paradigm that UML is based on. Moreover, SystemC can be co-simulated with other hardware description languages [31].

With SystemC being used increasingly popular in system level design, some high level SystemC models of platforms (including peripheral, bus, memory and processor) are available [37, 33, 44]. These models can be incorporated into our framework as elements in the platform library to be used for design space exploration in the Y-chart scheme (see figure 2-5).

One might wish to consider SystemC itself as the high level system description language. However, at the application level one would like to have visual notations for interacting with the end users to capture requirements easily and precisely. It is also important to be able to use standard *models of computation* (MOCs) at the initial design stages. Further, one may not wish to concretely specify the communication mechanisms and instead leave it to be defined by the underlying operational semantics of the MOCs being deployed.

We think that for large scale systems, modeling at UML level makes a big difference. First, the designers can focus on the model, not the program, which free them from many unnecessary details and allows them to focus on important aspects of the design. Moreover, the modeling at UML level and the automatic SystemC code generation makes the design process faster. It also helps to remove human errors when writing the code, especially with complicated models. In addition, different representations can be derived from UML models for various purpose, such as SystemC for simulation and another representation for formal verification.

2.2.1 Overview of the framework

Fig. 2-1 shows our overall design framework. UML is used to capture the design of a system, including its architecture and behavior, the testcases for the design and how the system is expected to behave in different scenarios. The UML model is then

translated into SystemC executable program to do simulation so that the designer can verify whether the design satisfies the requirements based on the simulation trace. This simulation trace shows different properties/aspects of the system under test, such as performance and interaction sequence which can help designers validate the design. If the simulation traces show that the design is not satisfactory, designers can go back to UML model to modify it, re-generate SystemC code and check its behavior again.

All of the above are done before implementation is carried out, namely hardware is realized and software is implemented on the target hardware. Here for our purpose of modeling event-triggered and time-triggered systems, we selected a subset of UML and gave the chosen notations the semantics that are suitable for these types of systems.

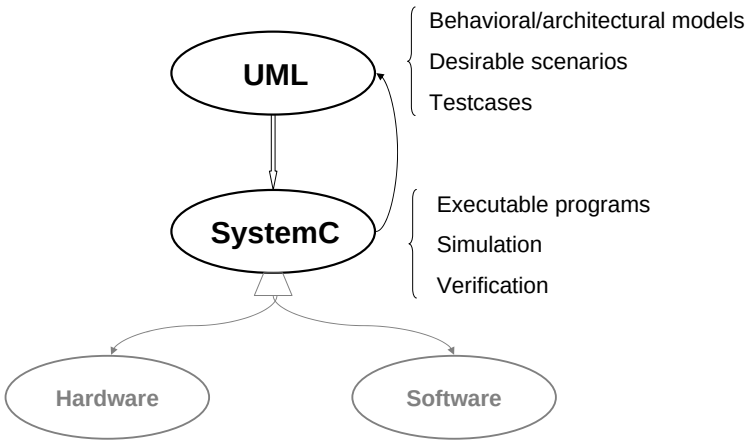

Figure 2-1: The envisioned design framework based on UML and SystemC

Fig. 2-2 shows the different levels of abstraction of a computation or communication component's model. An untimed model is an algorithm description. Designers can annotate timing information into the untimed model to get a timed model. When the model is more detailed and the timing information contains the approximate numbers of clock cycle it takes to execute, it becomes a cycle approximate model. Furthermore, the cycle accurate model can contain the behavior of the components at each clock cycle. Finally, the register transfer level (RTL) of a hardware component

describes the operation of a digital circuit with hardware registers and signals. The RTL model of a software component can be considered the software program running on the target platform.

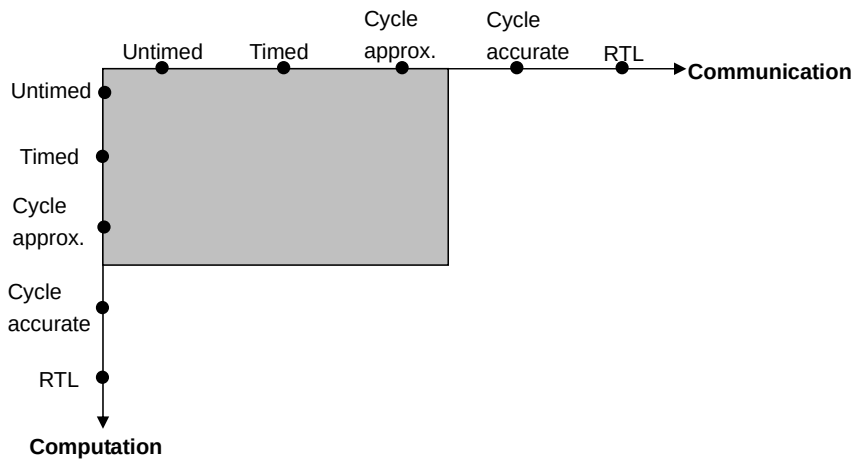

Figure 2-2: The levels of abstraction

For a system level model which consists of computation and communication components, we suggest and support the abstraction levels covered inside the highlighted rectangle in Fig. 2-2. The unnecessary details of implementation platform and techniques must be hidden. More specifically:

- Communication: no pin-level or cycle-accurate level synchronization. Communication should be done through function calls. The functions are captured in interfaces and the components are connected through ports. A component's port connects to another port with matching interfaces.

- Computation: abstract, high level data-types should be used instead of hardware data-types which operate on bits.

- Timing: clock-level timing should be avoided as this incurs much details and simulation overhead. There should be no clock synchronization if the system under design is a distributed system; all the nodes must assume that their clocks

are well-synchronized already. This ensures that designers focus on crucial characteristics of the system at high level design. Clock synchronization mechanisms can be added later in the design process.

With the supported levels of abstraction, designers can choose their own methods or paths to design and refine their system model. Fig. 2-3 and Fig. 2-4 show two of the possible refinement paths that designers can do within our framework. Note that the refinement is done manually. We do not support automatic refinement.

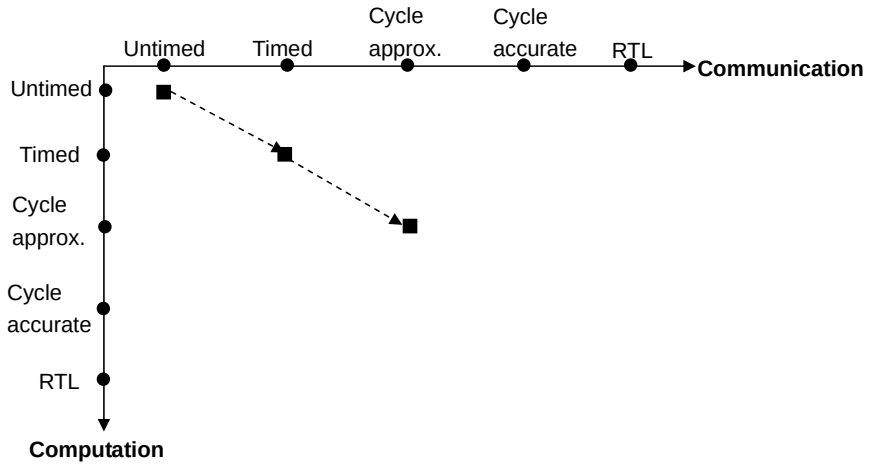

Figure 2-3: A refinement process

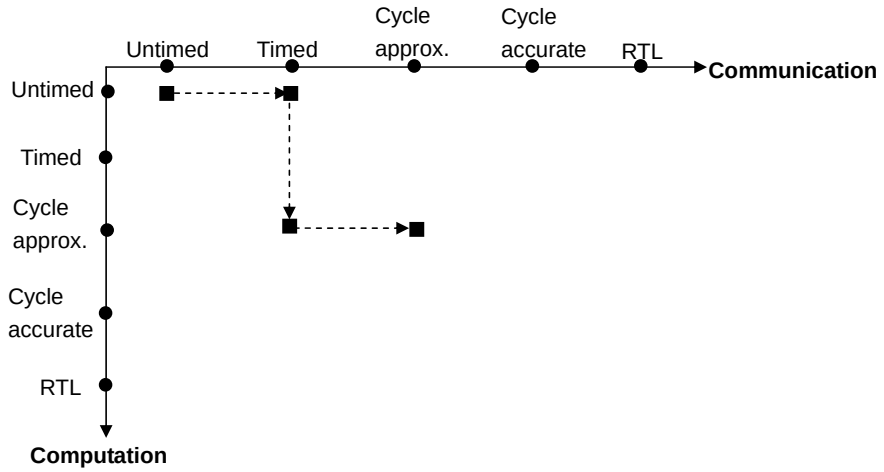

Figure 2-4: Another refinement process

Fig. 2-5 shows how our design framework fits in a Y-chart scheme. On the left hand side of the Y-chart, the behavior of the designed system can be captured using UML. Then SystemC program can be generated from this behavioral UML model to validate it. On the right hand side of the Y-chart, the potential architecture can also be captured in an architectural UML model and its corresponding SystemC code can be generated and simulated too. When both the behavioral and architectural models are ready, designers can map the behavior onto the architecture. This mapping can be done manually or with the support of some tools. Here are our suggested mapping steps which can be done by designers at UML level:

- Block mapping: decide which behavioral block is going to be realized or running on which architectural component. Then connect each behavioral component to it architectural component so that the behavioral component can use the services provided by the architectural component. Note that the components are categorized into two types: computation and communication.

- Communication interface adapting: insert adapters between interfaces of different behavioral and architectural components. This is necessary when the communication primitives used by the application model are different than the communication primitives provided by the platform. In the later steps of a development process, these adapters can be refined to device drivers and protocols.

- Operation mapping: make an application's computation block utilize the operations of the platform component that it has been mapped to during the block mapping step. Again, some adapters should be inserted if the interfaces are not matched.

After mapping, the SystemC code for the system being designed is generated. Based on the SystemC simulation trace, the correctness of the system is validated and the performance of the system is analyzed. This is because the trace contains

information about the interaction among the components in a system, input and output to the environment, the point of time at which an event or action occurs or how much time it takes to finish a function. If the system after mapping is not satisfactory, the UML behavioral model can be modified, or the UML architecture can be adjusted or changed, or the mapping can be changed.

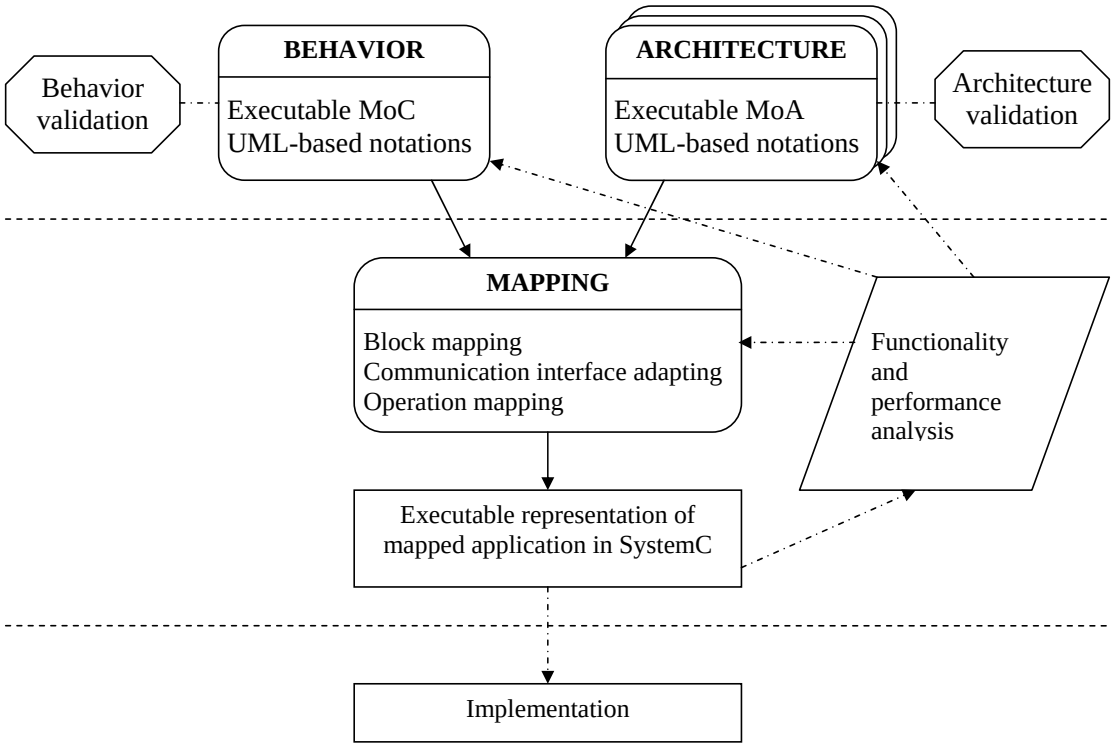

Figure 2-5: The Y-chart framework based on UML and SystemC

The above framework supports reuse. The architectural model can be designed from scratch or can re-use pre-defined components. Behavioral blocks can also be reused, e.g. the blocks representing commonly used algorithms.

Similar to the case of the Y-chart approach, our framework may fit in the platform-based design approach for the higher levels of abstraction in which both functionality and platform are modeled in UML and the mapped instance is validated by SystemC simulation before becoming the functionality for the next step.

In this research we focus on the UML modeling of the functionality and generating

SystemC code from it. Systematic architecture modeling of architecture components such as micro-controllers, ASICs, etc. is not targeted here. We also do not study the details of the mapping between the functionality and the architecture.

2.3 UML modeling

2.3.1 UML essential features

The current version of UML called UML 2.x is a large collection of diagrams and notations. It has 13 diagrams types. Here we briefly present the essential features of UML that we use in our framework. These notations and diagrams are chosen such that they can capture the structure, behavior of a system and its components, the test cases and system requirements at a high level of abstraction. In addition, they contribute to the SystemC code generation process.

Class diagrams

Each class in a model represents components of the same type, or having similar behaviors. More crucially, a class diagram depicts the classes in a model and how they are related to each other. The relationships include generalization (or inheritance), aggregation/composition and association. Generalization (or inheritance) is for a class to inherit attributes, operations and behavior of another class or implement the operations defined in another class. Aggregation/composition is to capture that an object of a class may contain one or more objects of the other classes. Association exists when an object of one class have interaction with another object of another class. In case the other class is an interface, the association relation indicates that one class will call operations defined in the interface to interact with its environment. The number at the other end of an association relation refers to the number of instances of the other class which is (are) related to an instance of this class. A class may have many relationships with different other classes.

Fig. 2-6 is an example of a class diagram which is used to capture the relation-

ships between different types of components. Each of the block is a class. The connections represent their relationships. In this case, the `simple_bus` class inherits or implements `simple_bus_blocking_if`, `simple_bus_non_blocking_if` and `simple_bus_direct_if`. The `simple_bus` class has `<<Channel>>` stereotype to indicate that this is a communication channel. The `simple_bus_blocking_if`, `simple_bus_non_blocking_if` and `simple_bus_direct_if` classes have `<<Interface>>` to indicate that they are interfaces which include only definitions of operations to be implemented by the `simple_bus` class. The association relation between `simple_bus_master_blocking` and `simple_bus_blocking_if` tells that the `simple_bus_master_blocking` will call the operations defined in the `simple_bus_blocking_if` interface to communicate with its environment.

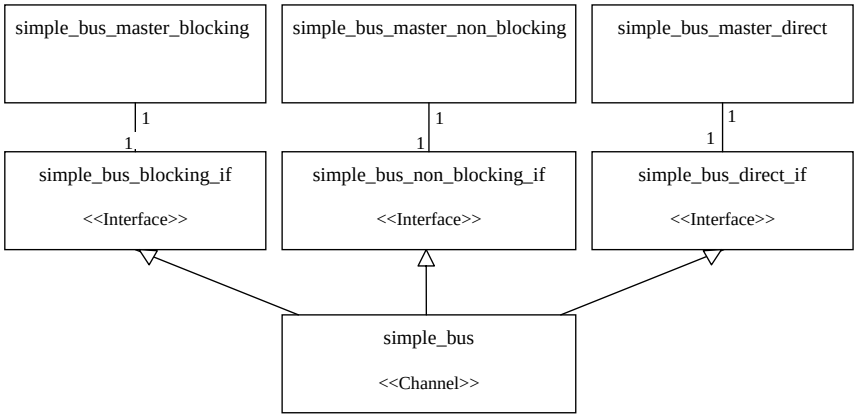

Figure 2-6: A class diagram

Structure diagrams and structured classes

In complex systems, an object of a class may contain objects of other classes and the relationships between objects may be intricate. UML provides structure diagrams to model the internal structure of classes more accurately. Moreover, they are used to capture hierarchical structure. An important concept of UML structure diagrams is ports, which are the connection points of an object to its environment. Ports come with provided or required interfaces which define the set of operations that the object provides or calls through the ports. The parts in a structure diagram represent the

internal components. They are connected through ports with interfaces. Fig. 2-7 is an example of structure diagram which shows the structure of a cluster containing the components of a brake-by-wire application, including two managers, four actuators at four wheels and different sensors. The right hand side of the figure shows the internal structure of the manager.

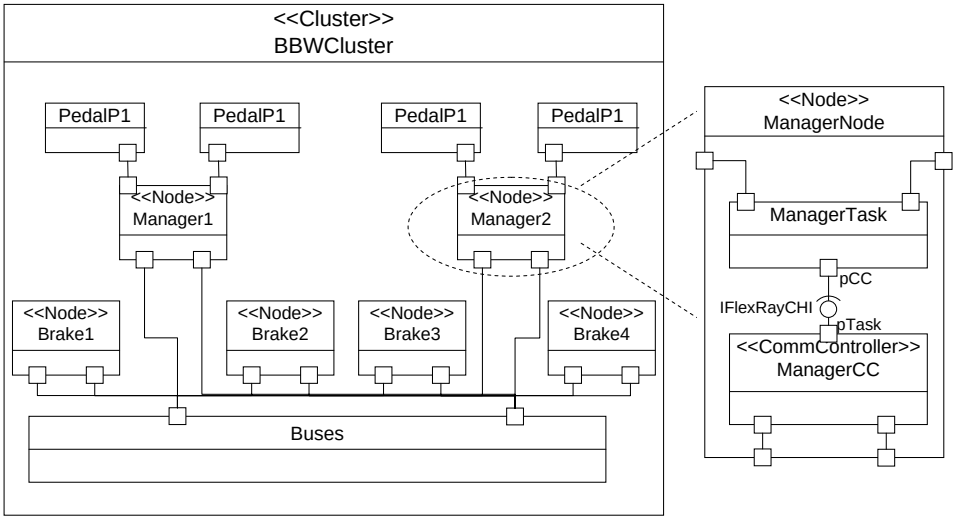

Figure 2-7: A composite structure diagram

Behavioral state machines

In UML, an object-oriented variant of statecharts [68] called *behavioral state machines* are used to model the behavior of components in a system. Behavioral state machines describe the behavior of classes. A state can be a simple state or a composite state. A composite state may consist of concurrent substates; in this case it is called an orthogonal state. On the other hand, a composite state which consists of sequential substates is called a simple composite state. Being in an orthogonal state means being in all of its substates. Being in a simple composite state means being in exactly one of its substates. Each simple composite state has an initial pseudostate and an optional final state.

Each state is associated with a set of actions on entry and actions on exit. These will be executed when the object enters and leaves that state respectively.

A transition connects a source state and a target state. The label of each transition includes a trigger event, a guard and a sequence of actions. Trigger events may be associated with data. A guard is an expression which returns a Boolean value. When an object is in a state and an event of an outgoing transition of that state occurs, the corresponding guard is evaluated. If the guard is true, the transition is taken, the sequence of actions is performed and the object moves into another state. Otherwise, the object stays in the current state. The order in which the actions of the states and transition is as follows: action on exit of the source state, then actions of the transition and finally, action on entry of the target state. In case the transition cross a state(s)'s border, the actions on exit is executed starting from the substate(s)'s to the father state(s)'s and actions on entry is executed from father state(s) to substate(s). An example of an hierarchical state machine and a cross-level transition is shown in Fig. 2-8.

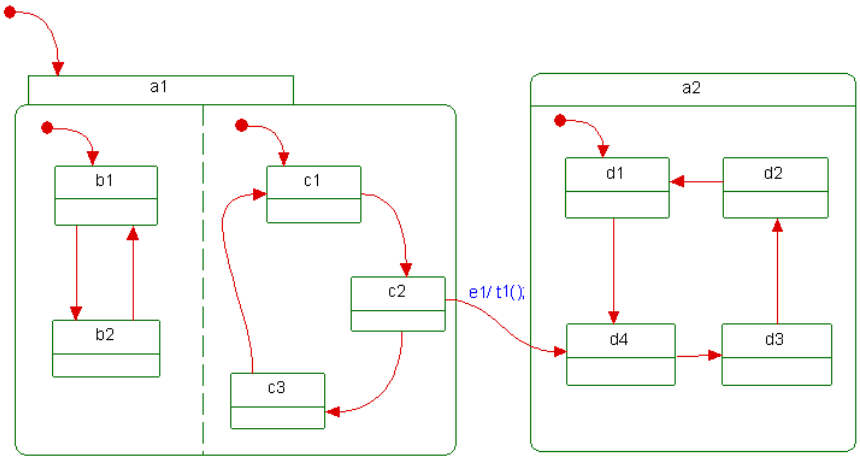

Figure 2-8: A fragment of a hierarchical state machine

Fig. 2-9 shows a hierarchical behavioral state machine of a brake actuator in which the transitions at the top level are triggered by time-out events while the transition inside `dynamic2` state is triggered by an event from another component.

Use case diagrams

A use case diagram describes the usage requirements for a system. Fig. 2-10 shows

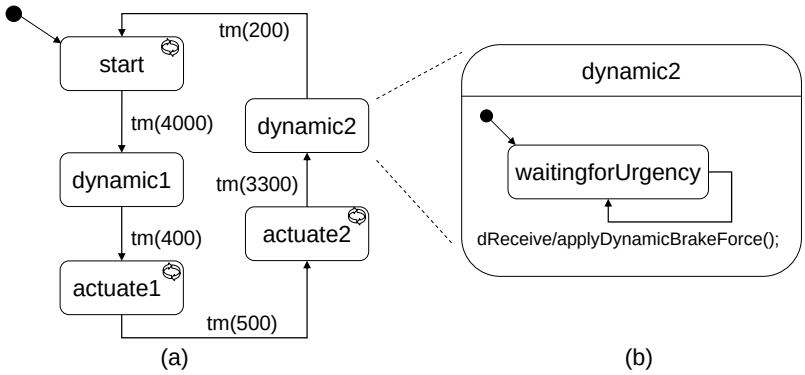

Figure 2-9: A behavioral state machine

a simple use case diagram describing the usage of an online store customer. The actor `Customer` is the user of the system. The use cases `Search for items`, `Place orders` and `Obtain help` are the actions that the `Customer` can perform on the system. In other words, they are the services that the system offers to its users.

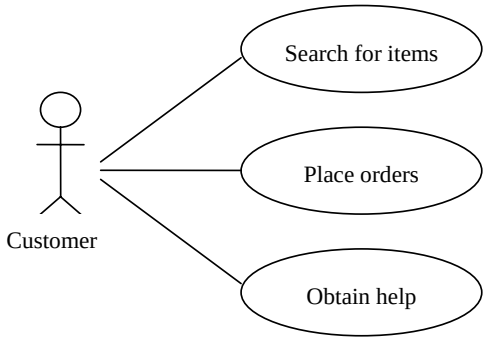

Figure 2-10: A use case diagram

Activity diagrams

In contrast to behavioral state machines which depict the behavior of objects of the same class, an activity diagram displays the activities across different objects which may belong to different classes. Activity modeling emphasizes the sequence and conditions for coordinating behaviors of different objects [94]. An activity diagram has multiple activity nodes connected together. In order to determine the order of actions, a unique token flowing through the diagram determines the activity occurring.

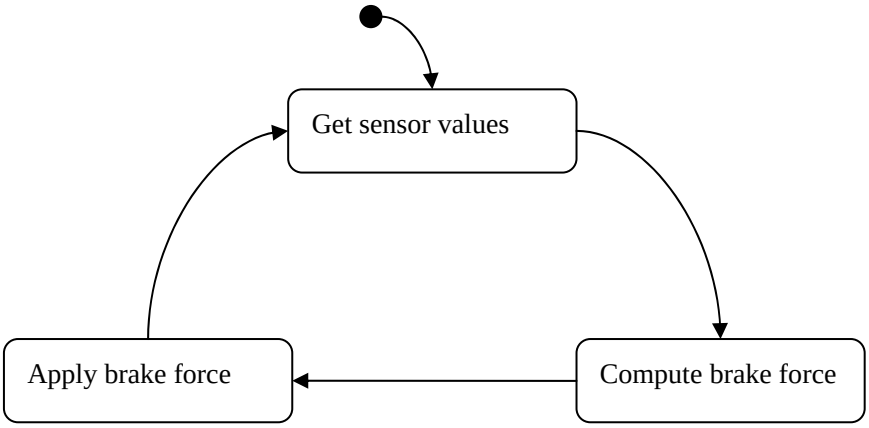

Figure 2-11: An activity diagram

The token starts from the initial node and stops at the final node. Fig. 2-11 shows a simple activity diagram describing a car's process of getting the values from the brake pedal sensor, computing the brake force and applying the force on the wheels. Here which components inside the car carrying out these activities are not specified.

Sequence diagrams

Sequence diagrams depict the messages being exchanged between different objects with the focus on the order in which the messages occur. Each vertical lifeline represents an object and each horizontal line represents a message, which can be an event sending or a function call. Time progresses from the top of a sequence diagram to the bottom. Sequence diagrams are usually used to describe usage scenarios or the interactions among the components when the system is doing some service. Fig. 2-12 is a sequence diagram with a life line represents the driver of a car, who sends requests to the internal components of the system under model.

Extension mechanisms

UML provides some extension mechanisms, namely stereotypes, tagged values, and constrains that make UML be able to support different types of systems and domains while avoiding language explosion. Stereotypes is a means to classify elements. For example, designers can introduce stereotypes "communication" or "computation"

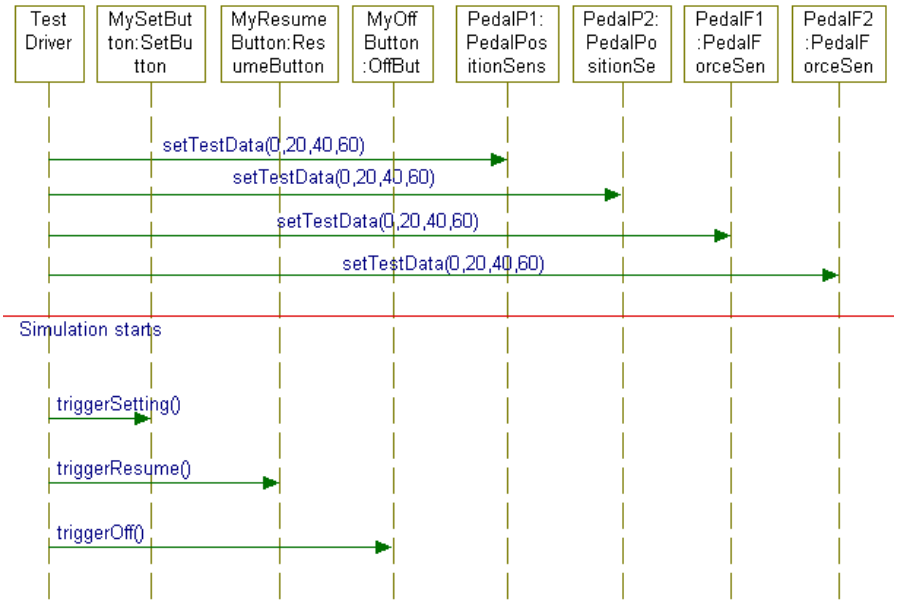

Figure 2-12: A sequence diagram

to classes to distinguish between classes that represent communication or computation components. Constraints allow some properties to be specified linguistically for a model element. A tagged value is a (Tag, Value) pair that permits arbitrary information to be attached to any model element. A UML profile is a collection of the stereotypes, constraints and tagged values defined for modeling in different domains. In our framework, we define a SystemC profile at UML level in case designers want to lift up SystemC concepts to UML. In addition, there is a profile for time-triggered communication platform to describe the platform at high level of abstraction and facilitate the code generation and enhance simulation speed.

2.3.2 UML in our framework

In our framework, we use the above diagrams for modeling and generating SystemC code. The extension mechanisms are utilized in order to distinguish different component types and thus, facilitating the code generation and simulation speed optimization process. The chosen diagrams and extension mechanisms are suitable for

our purpose of specifying a system level design model of a system and generating SystemC code.

Table 2.1 is a summary of the UML notations being used in our framework. The details on how these diagrams are used in the context of event-triggered and time-triggered systems will be presented in the respective chapters.

To be captured	**UML elements**
Structure	Class diagrams, structure diagrams, ports, interfaces, and stereotypes
Behavior	Behavioral state machines
Platform's service	Use case diagrams, activity diagrams and tagged values
Usage scenario	Sequence diagrams
Expected scenario	Sequence diagrams

Table 2.1: Summary of UML notations used in our framework

While modeling at high level of abstraction, it is necessary that the model should produce fairly accurate performance numbers for design validation and design space exploration. In our framework, designers can annotate delays in behavioral state machines as time-out events. For the accuracy of performance analysis by simulation, these delays must be carefully computed/estimated and inserted into the behavioral state machines. These delays can be obtained by different ways, simulation, analytical methods or from real execution if the component has been implemented and now it is being re-used from a library.

In object-oriented paradigm, a behavioral state machine captures the behavior of different objects of the same class. Although objects of the same class have similar behavior, the time-out events for different objects may have different values. Our framework allows designers to have variables in the time-out statements so that the delays can be adjusted for different situations or objects of the same class.

In summary, UML is a high level modeling language for system level design for embedded systems. It is used to capture various aspects of a system, such as structure, behavior, test cases and requirements. At the same time, since a UML model cannot

be simulated directly, it is necessary to have an executable internal representation to do simulation and validate the design.

2.4 SystemC intermediate representation

SystemC [62, 10] is a system level language that is suitable to serve as the intermediate representation in our framework. It is an IEEE standard language built entirely on C++. The SystemC language is suitable for system level modeling, design and verification. However, it is not limited to high level design. On the other hand, SystemC modeling can be done at different levels of abstraction. Since 1999, the SystemC user community has been growing to a large number of system design and semiconductor companies and IP providers. Tools supporting SystemC modeling, simulation, verification and synthesis have been developed by many companies. The role of SystemC in the system level design context has also been explored in detail in [86, 62].

2.4.1 SystemC essential features

SystemC separates computation and communication by having modules and processes for computation; ports, interfaces and channels for communication. Modules are the basic building blocks for partitioning a design. A module hides its data and algorithms from other modules. Modules communicate through channels. There are two types of channels: primitive channels and hierarchical channels. Primitive channels are in some sense, stateless while hierarchical channels can have internal states and control flow associated with them. As the name suggests, hierarchical channels can contain other channels, modules or processes. Interfaces specify the signature of the operations provided by channels. A module accesses a channel through a port whose type is one of the interfaces implemented by the channel.

A component interacts with its environment through port(s), each port is associ-

ated with an interface, a set of operations that are provided or required by this port depending on whether it is a `sc_export` or `sc_port` respectively.

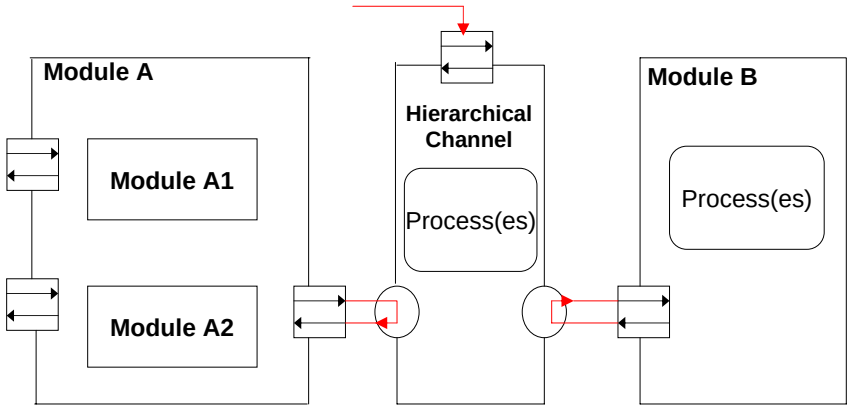

Figure 2-13: SystemC basics

Both communication and computation components, except primitive channels, have processes to capture their behavior. It may have one or more processes which can run concurrently. There are three types of processes: `sc_thread`, `sc_method` and `sc_cthread`. `sc_cthread`'s execution is triggered by clock edge, hence suitable for low level design while `sc_thread` and `sc_method` are triggered by SystemC signals or events. As opposed to `sc_method` which is sensitive to only one event, `sc_thread` can be sensitive to different events at different points of control. This makes `sc_thread` suitable for capturing the behavioral state machine model. However, the simulation speed for `sc_thread` is slower compared to `sc_method` as the SystemC simulation kernel has to keep track of the current state of the thread.

The SystemC simulation kernel [10] which comes with the SystemC library is a discrete-event simulator. It manages the progression of time and the execution of all the processes based on the events exchanged among them and the time. The simulation consists of delta cycles. Each delta cycle involves determining which processes are ready to be executed, executing the processes, updating the primitive channels and advancing the simulation time [64]. The simulation is deterministic in the sense that two simulation runs of the same program with the same input will give the same

results. However, when two processes are triggered at the same time, there is no way of telling which process runs first.

Currently there are some synthesis tools [35, 118, 117, 14, 61] for SystemC. They synthesize a SystemC behavioral hardware module into an RTL hardware description language program or a gate level netlist. These tools place restrictions on the SystemC code that can be synthesized. We refer the reader to [2] for the SystemC synthesizable subset. Here are some examples of the unsupported features. Several object-oriented features such as dynamic object creation and virtual functions are not supported. Pointers which contain address determinable during compilation are supported. Otherwise, they are not supported. Certain data types such as limited precision fixed-point types and arbitrary precision value and the SystemC/C++ methods related to them are not supported.

Celoxica's Agility compiler [35] supports register transfer level (RTL) and behavioral level SystemC models. These models are cycle accurate for communication. While computation at RTL must be cycle accurate, computation at behavioral level is timed or cycle approximate. Agility compiler feedbacks useful information such as area and delay estimation, register usage and hardware optimization information. Forte Design Systems' Cynthesizer brings the SystemC synthesizable model to a higher level - transaction level model. Cynthesizer is able to produce a report on area, performance and power. xPilot [14] synthesizes behavioral SystemC model into configurable processors, multi-cores, and multi-processors or highly customized devices, such as FPGA and ASIC. Systemcrafter SC [117] synthesizes SystemC into RTL VHDL or Verilog for Xilinx FPGAs. ESEComp [22] synthesizes ESE SystemC designs into Verilog RTL.

Other executable system level languages are SpecC [57] and SystemVerilog [60]. SpecC is derived from software programming language C. SystemC and SpecC both separate computation and communication. While SystemC supports multiple models of computation, SpecC supports only its own models of computation (parallel,

pipeline and FSM). Moreover, SystemC is a library built entirely on C++ so any C++ compiler can be used to compile a SystemC program. Meanwhile, SpecC is a new language. SystemC has wider support by EDA vendors in many commercial tools.

SystemVerilog is derived from hardware description language Verilog. SystemVerilog is more suitable for RTL design and verification. On the other hand, SystemC is better for writing abstract models and it is good for reusability and IP design.

2.4.2 Efficient SystemC simulation

Since system level design aims to deal with the complexity of systems nowadays and to improve productivity, the validation of the design should be fast. Thus, it is crucial to have high simulation speed. In order to have efficient simulation speed, it is important that designers choose the right level of abstraction, for both computation and communication. For communication, transaction level modeling should be used. As for computation, high level datatypes should be used and context switching should be minimized.

A key feature of SystemC is that communication can be modeled at a high level of abstraction often referred to as *transaction level modeling* (TLM). It is hard to pin down this notion precisely. Intuitively, communication between components is described through method calls. Here, 'transaction' stands for the exchange of data between two components of a system. This level emphasizes what data are transferred and from which locations but not the details of the specific protocol used by the communication. Hence, when a component wants to send some data to another component, the sender will just call a function provided by the receiver to transfer data. Thus, inter-component interactions are abstracted from the details of the implementation of the communication architecture and this facilitates component reuse. In addition, simulation at this level can be usually carried out at a much high speed. For a more detailed description of TLM, see [32]. At this level, the basic communica-

tion unit consists of a method call and hence the performance numbers reported will generally not be cycle accurate. However, acceptably accurate performance numbers can be obtained by inserting fairly accurate delays at the right places. One way to improve the accuracy is to insert delays obtained from analytical methods or real execution of component (if it is reused) after every critical section.

The Open SystemC Initiative (OSCI) [10] has announced the TLM 2.0 interface standard. The standard contains interfaces for loosely-timed modeling, interfaces for approximately-timed modeling, generic payload for memory-mapped buses, direct memory interface, and debug transaction interface. It enables SystemC model interoperability and reuse at the transaction level.

SystemC simulation kernel is a discrete-event simulator. It manages the progression of time and the context switching between the processes based on the events they are waiting for. Regarding simulation speed, our experience is that the number of processes doesn't matter as long as the amount of context switching is not much. Unnecessary context switching with little or no useful processing is one of the causes of poor simulation performance. The amount of context switching should be reduced by reducing the number of `sc_thread`s. In addition, the execution of a thread, whenever it is active, should be made as long as possible, because a thread that is triggered to be active many times but does not do much when it is active will consume much simulation time. On the other hand, `sc_method`s are faster than `sc_thread`s. Hence `sc_method`s should be used as much as possible, whenever a process is only sensitive to an event.

Another important factor in context switching is the role played by clocks. A SystemC model with clocks is slowed down. At high level design, a clocked model should be modified by changing the sensitivities to clock edges to waiting of time calculated from clock period. For example, if a SystemC process is sensitive to positive clock edges and do some processing after every 5 edges, it can be modified to wait for a time equivalent to `5*clock_period`.

2.5 Summary

In this chapter we have presented our system level design framework based on UML and SystemC. We have discussed the reasons why UML and SystemC are chosen. In particular, we have gone into the UML diagrams and notations selected to capture different aspects of a system which allow designers to model requirements, systems' structure and behavior as well as testcases. In addition, the model can be used to generate executable SystemC code automatically. And we have discussed how to have a system level SystemC model that offers fast simulation as well as fairly accurate performance numbers.

Existing SystemC models can be reused in our framework. If a designer has a SystemC model of a component, she/he needs to specify the connection between this component and other components inside the system at UML level. She/he does not have to specify the behavior of this component. After generating SystemC code of the whole system, the designer can just compile the SystemC code for the existing component with the newly generated code to simulate the whole system.

One might consider Rational Rose Technical Developer (previously known as Rational Rose RealTime) [8] and Rhapsody [100] as the system level design tool. They are able to generate C, C++ and Java code from UML models. However, SystemC is a better candidate for a system level design language than C, C++ and Java thanks to its features (presented in section 2.4). In addition, there are existing system level models in SystemC which are ready to be re-used and integrated into a new system.

We have shown how our framework can fit in the Y-chart and platform based approach, although more research should be done on how to do the architecture modeling and mapping. While our framework is simulation-based, it is possible to combine it with analytical frameworks, such as WCET so that the delays inserted in behavioral state machines are computed based on WCET. The implementation of our framework makes it easy to integrate with other analytical or formal verification frameworks. The details of the implementation will be discussed in the later chapters.

Chapter 3

UML-based design for reactive systems

In this chapter, our design framework for reactive systems is presented. We focus on how reactive systems are modeled in UML. We also show the techniques for synthesizing executable SystemC code from UML hierarchical state machines and the SystemC communication channels from UML event communication specifications. This work has appeared as a conference paper [91] and a book chapter [92]. We begin by reviewing related work on UML and SystemC and executable code generation from behavioral state machines.

3.1 Related work

UML-SystemC translation

Closely related to our approach, UML and SystemC have been used together in [121, 125, 29, 110]. Our use of stereotypes for SystemC components is similar to those proposed in [121, 103, 29]. [125] proposes an extension of UML based on ROOM and RoseRT. The extension includes stereotypes for modules, channels, interfaces and ports. In the design framework proposed in [125], UML is used to model structure

only. Then SystemC model is developed for functional and performance evaluation. Similarly, [29, 17, 113] use UML merely to capture the *structural* aspects of the system under design. Thus, only SystemC skeleton code is generated. [110] also generates SystemC skeleton code from UML model. Furthermore, it provides a path to hardware implementation by taking the generated SystemC code and user input methods to generate code in hardware description languages such as VHDL and Verilog.

In contrast to the above approaches, our approach provides for the full fledged use of state machine diagrams — including C++ code associated with the actions — and hence can capture *system behaviors* exhibiting concurrency at the UML level.

A UML profile for SystemC [104, 49, 102, 23] brings SystemC concepts for both structure and behavior up to the UML level. Executable SystemC models are generated from UML models. However, designers need to know SystemC concepts in order to use this framework. Moreover, this work does not bring the design to a higher level of abstraction because the design being done here is actually a SystemC-based design.

Program synthesis from state machines

An important aspect of code generation from UML models is to generate executable programs to simulate the behavior of hierarchical state machines. This is not trivial since hierarchical state machines will include concurrency as in our framework. There are two ways to achieve this:

- In the first approach, a state machine becomes a set of data structures coupled to an execution engine, as in [123]. States are stored in a hierarchy tree and transitions in a hash table. The set of active states in which the object is residing comprises a configuration. The execution engine based on the event trigger chooses the transitions to be executed, executes the actions and moves to another set of active states. This approach avoids the exponential code growth. The translator only has to generate the data structures corresponding to the state machine. The engine is the same for different state machines. So in this

case the translator's task is simply to generate the data structures from the state machines.

- In the second approach, a state machine itself is mapped into code, usually through switch statements or function pointers [100, 15]. The code is executed to simulate the state machines. There is no execution engine. The state tree is encoded in a nested switch statement or in a class hierarchy with virtual methods [123]. The latter is referred to as state pattern [15]. Although the approach in [15] generates more understandable (readable) code, it is not effective because there are so many objects created at the beginning and it is not scalable (since it used state pattern, each state is an object).

The difference between our framework and the above related works is that we generate executable code which will be running on a SystemC discrete-event simulation kernel which manages the progression of time and supports concurrency. Thus, we can take advantage of the simulation kernel to reduce our effort. However, we still have to take care of the synchronization and communication of the concurrent execution. We use the techniques discussed in section 2.4 to speed up the simulation of hierarchical state machines.

Research issues

The first problem to be targeted in this chapter is how to have UML capture both the structure and the event-triggered behavior of a system so that executable SystemC code can be generated automatically from the UML model. This problem involves several issues. First, among a number of UML diagrams and notations, which ones are most suitable for our purpose. Another important issue is the semantics given to UML notations in order to model event-triggered platforms and applications, especially for the sending and receiving of events.

Another issue is that from UML event communication specification, how to generate SystemC communication channels. This synthesis must make sure that the chosen model of computation at UML level is executed correctly and effectively.

The generated SystemC code may be simulated many times to test and debug a design. Hence, it is crucial that the SystemC code generated from UML is effective in terms of simulation speed. In addition, designers should be able to further refine or synthesize an abstract design towards implementation. This requires the generated SystemC code to be readable and understandable.

3.2 The UML design pattern

In this section we discuss the UML notations are for modeling both structure and behavior and their semantics. We use the Rhapsody tool [100] to capture the UML specification. However, our approach does not specifically depend on this tool.

To bring out the main aspects of our modeling method, we will use the simple bus model available in the SystemC package [10] as a running example. In this system, there are three masters, namely a blocking, a nonblocking and a direct master. In addition, there is a bus and two memory slaves, one fast and one slow. A master initiates transactions on the bus to access a memory. Fig. 3-1 shows the block diagram for this example.

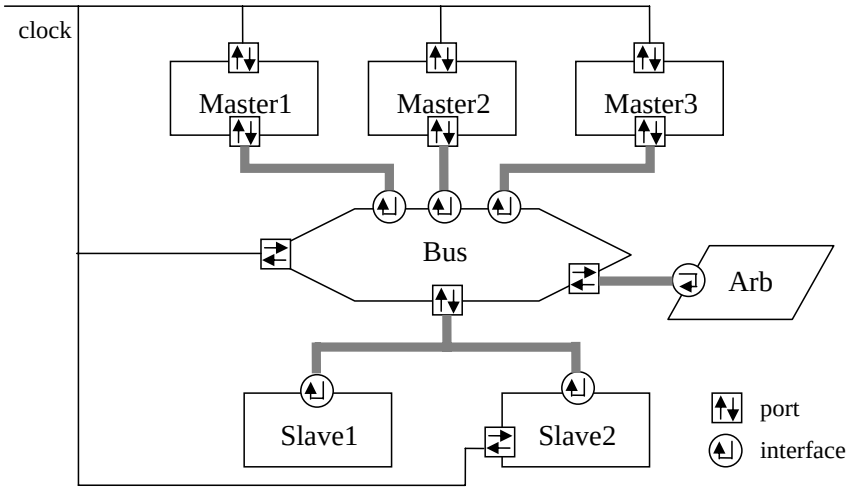

Figure 3-1: The block diagram of the simple bus example [64]

3.2.1 Structure

The key aspect of modeling structure is to capture the relationships among the components in a system. These relationships can be grouped into the following categories:

- Communication relationships between components.
- Containment relationships between the sub-components and the containing components.

In most systems, usually there are components that have similar behavior. Moreover, for fault tolerance, important components are usually replicated. Hence, in order to reduce the modeling effort, the common attributes and behavior of the similar components are modeled as a 'virtual' component. It is considered the abstract or general component. The 'real' components implement this 'virtual' component. Implementation relationships do not exist physically in real system but necessary in object-oriented modeling to better organize the models and to promote re-use.

We use the class diagrams in the usual way to describe the above relationships. In the simple bus example, the masters send requests to the bus by function calls. So does the bus to the slaves. A special type of classes called *interface* is used to capture the functions that a class provides for other classes to call. Fig. 3-2 shows a fragment of the class diagram of the simple bus example, in which we can see that different masters use different interfaces to access the bus. Thus the bus implements these three interfaces.

Classes can be related by the following relations:

- Generalization (or inheritance): when a class implements an interface, it inherits that interface. Moreover, an interface and class can inherit another interface and class respectively.
- Aggregation/composition: an object of a class contains object(s) of other class(es).

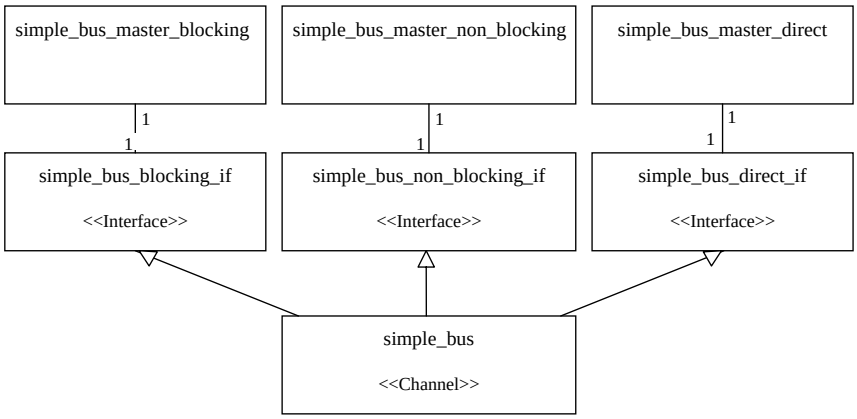

Figure 3-2: A class diagram

- Association: classes that exchange messages with each other are associated to one another. We model messages by UML events with or without arguments. Furthermore, a module may have an association relationship with an interface when it accesses a channel through this interface.

The aggregation/composition relations are modeled more clearly in structure diagrams which describe the internal sub-objects of object(s). Fig. 3-3 shows a structure diagram of the class which represent the bus system.

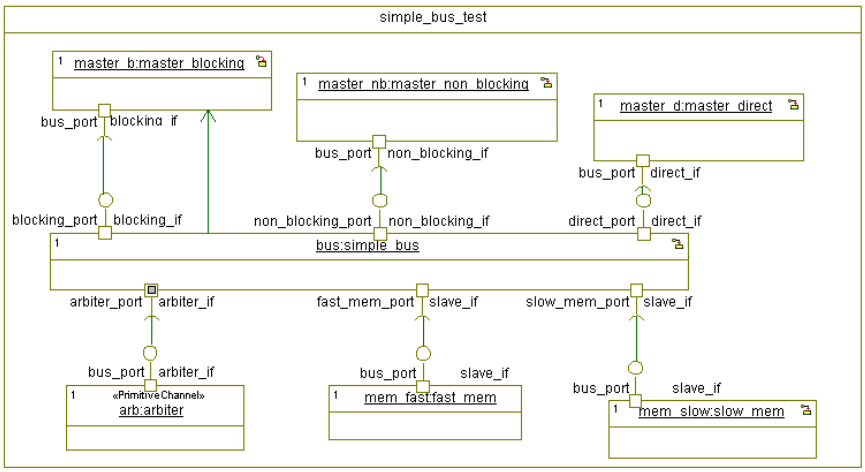

Figure 3-3: The structure diagram for the simple bus example

3.2.2 Behavior

In UML, an object-oriented variant of statecharts [68] called *behavioral state machines* are used to model the behavior of components in a system. Behavioral state machines describe the behavior of classes. Two main concepts in behavioral state machines are states and transitions.

A state can be a simple state or a composite state. A composite state may consist of concurrent substates; in this case it is called an orthogonal state (AND state). In contrast, a composite state which consists of sequential substates is called a simple composite state (OR state). Being in an orthogonal state means being in all of its substates. Being in a simple composite state means being in exactly one of its substates. Each simple composite state has an initial pseudostate and an optional final state.

Each state is associated with a set of actions on entry and actions on exit. These will be executed when the object enters and leaves that state respectively.

A transition connects a source state and a target state. The label of each transition includes a trigger event, a guard and a sequence of actions. Trigger events may be associated with data. A guard is an expression which returns a Boolean value. When an object is in a state and an event of an outgoing transition of that state occurs, the corresponding guard is evaluated. If the guard is true, the transition is taken, the sequence of actions is performed and the object moves into another state. Otherwise, the object stays in the current state. The actions of the states and transition are executed in the following order: action on exit of the source state, then actions of the transition and finally, action on entry of the target state. In case the transition cross a state(s)'s border, the actions on exit is executed starting from the substate(s)'s to the parent state(s)'s and actions on entry is executed from parent state(s) to substate(s). An example of an hierarchical state machine and a cross-level transition is shown in Fig. 3-5.

Modeling concurrency is an important part of a system specification and this is

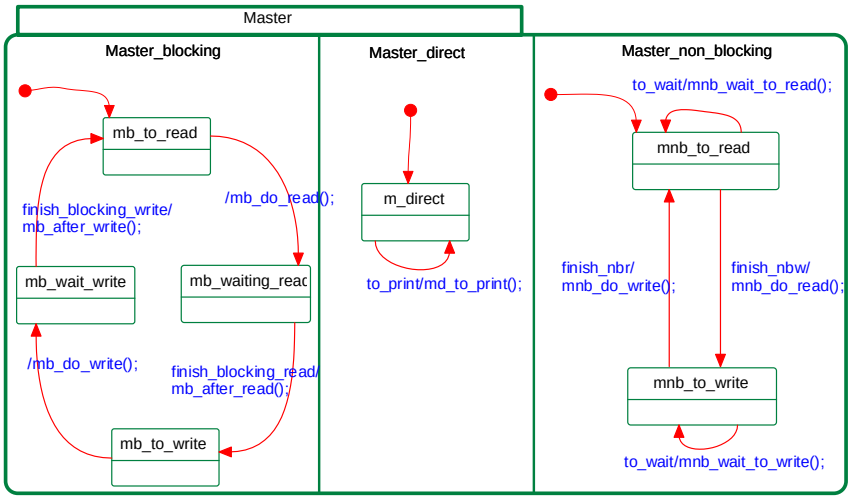

Figure 3-4: An orthogonal state

achieved with the help of orthogonal states. Fig. 3-4 shows a state machine diagram of a master which is a combination of the three masters described above. This is a derived version of the simple bus model in the SystemC package; we have combined the three masters into one master state machine diagram. The orthogonal state (AND state) `Master` has three substates, each of which is a simple composite state (OR state) which in turn has a set of simple leaf states that have no internal structure.

The transitions are not necessarily between two states having the same parent. Instead, transitions can cross a state's or many states' border. In this case, the nested substates will be exited before their composite states and the composite states will be entered before the nested substates. As an example, in Fig. 3-5, suppose the active states are (a1, b2, c2) and event *e1* arrives. Hence the transition t1 is enabled. As a result, states b2 and c2 are exited simultaneously (their actions on exit are executed), followed by action on exit of state a1, then the action of the transition, action on entry of state a2 and finally action on entry of state d4.

The actions associated with a transition or a state can be C++ statements or a function call whose body (in the form C++ code) is to be provided by the user. This code decides the level of abstraction for computation. The action could also

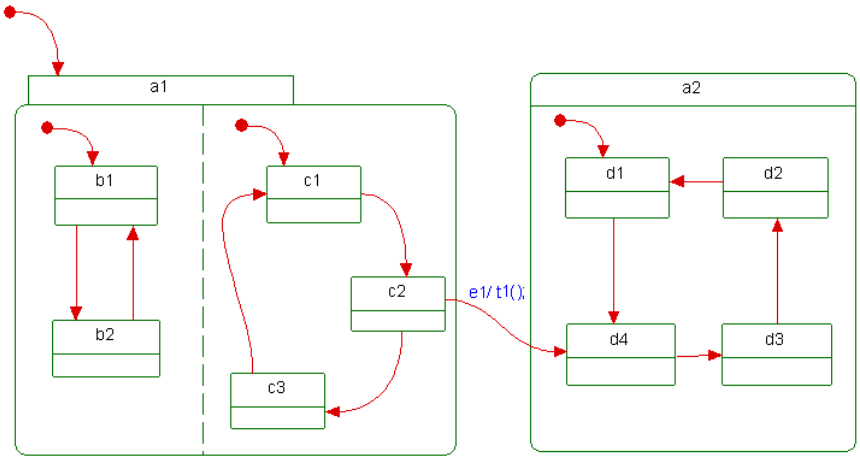

Figure 3-5: A fragment of a hierarchical state machine

correspond to sending an event to another state machine diagram (describing the behavior of a different class). In addition, the action could be calling an interface method through a port. Moreover, in the actions, we support specification of clock sensitivity or delays in terms of clock cycles or time units through C++ macros. This gives the designers an option to have timed models. Furthermore, this allows users to provide annotations of timing information for performance estimation and architectural exploration. For transaction level modeling (TLM) implementations, we do not restrict the C++ code associated with the actions in anyway. Although clock sensitivity is supported, it is not encouraged at this high level of abstraction due to simulation speed trade off.

A special statement for actions of states and transitions is sending out an event. This is an asynchronous send, i.e. an object can continue after calling this statement without waiting for the receiver to receive it.

Fig. 3-4 shows an orthogonal state named `Master` consisting of three states: `Master_direct`, `Master_blocking` and `Master_non_blocking`. We describe the behavior associated with the `Master_blocking` state. First, it goes from the initial state to `mb_to_read` state. Since there is no trigger event and guard for the transition, the function `mb_do_read` is called and the state `mb_waiting_read` is entered. In state

`mb_waiting_read`, when the event `finish_blocking_read` arrives, `mb_after_read` is performed and state `mb_to_write` is entered. Other states and transitions can be interpreted similarly.

In UML, events can be used as a mean to trigger a transition in states (of the same object or another object) or at the same time send some data associated with it. However UML does not define specifically how the events should be processed. In our framework, we require events to be sent point-to-point and not broadcast. An event is sent immediately when the sender issues the command and is consumed immediately if the receiver is waiting for it. Otherwise it is buffered at the receiver side. For each object, there are different queues for different types of events so that the order in which the events of different types are sent does not matter the execution of the receiver. We chose a queue for each event type instead of a queue for all the events sent to an object because at high levels of abstraction, it is hard to pin down exactly the point of time at which an event occurs. Fig. 3-6 shows a simple state machine of an object which transitions are triggered by event *e1* from another object *O1* and *e2* from another object *O2*. At high levels of abstraction, the order in which e1 and e2 are sent cannot be defined accurately. Suppose we use a unique queue for all event types of the object and e2 is enqueued first while the object is waiting for e1, a deadlock occurs. Hence, we separate the queues for different event types. This allows more behavioral scenarios compared to the case of a unique queue, but it is good for the early steps in the design process. The later refinement in a design process will narrow down the behavioral scenarios.

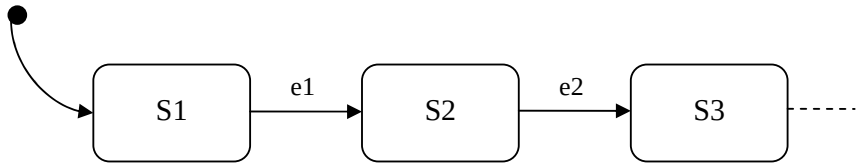

Figure 3-6: A simple behavioral state machine

There is a special type of events called time-out events which are expressed as

tm(t). After an object enters a state which has a transition with time-out event tm(t), after t units of time, the object must leave this state and take the transition.

In case where more than one transitions are enabled, one of them will be selected. In general, if t1 is a transition whose source state is s1, and t2 has source s2, then if s1 is a direct or transitively nested substate of s2, then t1 has higher priority than t2.

An event occurrence can only be taken from the event queue and dispatched if the processing of the previous event is fully completed. This is called run-to-completion processing. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal activities completed. The same conditions apply after the run-to-completion step is completed.

In addition to UML models which are independent of SystemC, we also support designers to lift up high level SystemC design to UML level by providing a simple SystemC profile at UML level. If designers are used to SystemC and would like to have UML as a front-end, they can exploit a SystemC profile at UML level that we provide. Through this, designers can have SystemC structural components specified at UML level and use behavioral state machines to model their behavior, then generate SystemC code automatically. Class notations are also used to define and distinguish between the various features of SystemC lifted up to the UML level using the stereotype mechanism. This is an extension mechanism of UML that allows one to define virtual subclasses of UML meta classes with new meta attributes and additional semantics. Using this, users can define a class as a module, an interface, a primitive channel or a hierarchical channel. In the case of the simple bus example, the bus is a hierarchical channel which implements the three interfaces. This simple SystemC profile provides a convenient way to model architectures. Moreover, if there is an existing SystemC component, the information about the type of the component (module, interface or channel) can be specified at UML level.

3.3 From UML to SystemC

After a system is modeled in UML, its executable SystemC code is generated by a translator in our framework. The translator is based on the semantics we chose for the UML notations and it generates executable SystemC code. In this section we explain this code generation process, including the mapping from UML to SystemC, how the translator is implemented and simulation speed is optimized.

3.3.1 SystemC code generation

The skeleton of the SystemC modules, interfaces, channels and their relationships are generated from class diagrams in a straightforward fashion. The instances/objects are created/initialized based on structure diagrams. We support the initialization of multiple instances of a type (module, primitive channel or hierarchical channel). However, they cannot be created dynamically since SystemC does not support dynamic instantiation; the structure of a system is determined at elaboration time before simulation starts. The functionalities corresponding to the modules and channels are generated from their state machine diagrams.

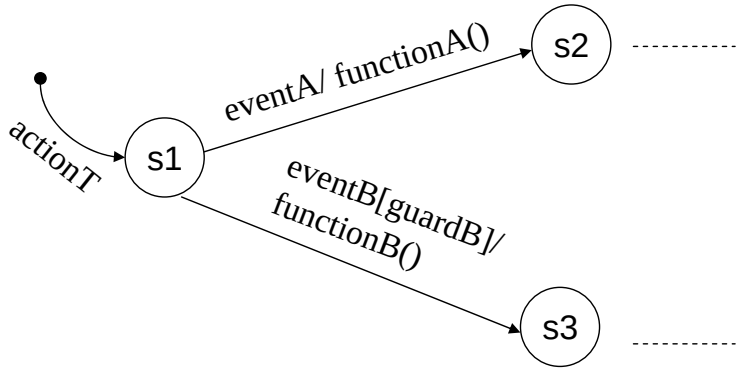

Figure 3-7: A fragment of a behavioral state machine

As mentioned earlier, we support hierarchical state machines including nested states and cross-level transitions (transitions that connects states of different nested

levels). Thus the mapping of hierarchical state machines to SystemC code is not trivial, as the following requirements must be satisfied:

- The orthogonal composite states require the simulation of concurrency.
- For the transitions involving composite states, the execution of the actions on entry, actions on exit of the states and the actions of the transitions must be in the correct order as defined in section 3.2.
- The run-to-completion requirement and the event processing order described in section 3.2 must be complied with.

Our mapping from UML to SystemC takes into account the simulation speed and the readability of the generated code in order to make it easy for designers to further synthesize to implementation.

Since SystemC `sc_thread`s can be sensitive to different events at different points of control, they are most suitable for capturing simple composite states.

We support the concurrency of behavioral state machines by having multiple SystemC threads in a SystemC module. Each OR state in the state machine is mapped onto a SystemC thread in which switch statement is used to capture the state transition among the direct children of that OR state. Each direct child is represented by a case of integer number in the switch statement, which is captured in an integer variable called state variable. Hence, only the states that are direct children of an OR state exist in the code of the switch statement.

In SystemC 2.1 which allows dynamic thread creation, the hierarchy of UML behavioral state machines can be implemented naturally by forking/joining SystemC threads when hierarchical states are entered/exited. With this dynamic scheme, threads are created when it is necessary only. Hence, at every delta cycle, the SystemC simulation kernel has to take care of the active threads only. However, our experimentations show that the cost of forking threads in SystemC is high compared to the scheduling cost of statically created threads. Therefore, we use the static

scheme in our framework, namely all the threads are created before simulation starts and triggered by events from the environment or events generated by other threads.

By default, a thread is sleeping in its initial state. It is triggered to active mode when the corresponding OR state is entered and returns to the initial state when the OR state is exited.

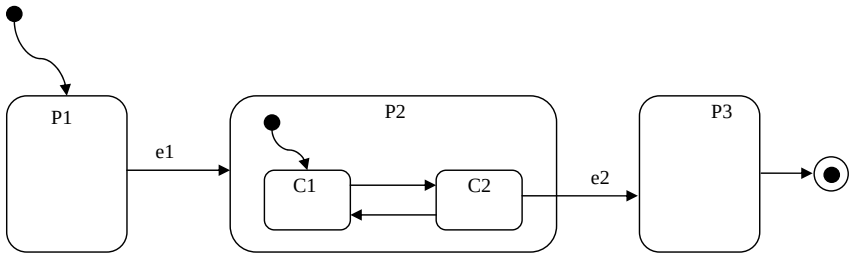

Figure 3-8: A hierarchical state machine with a simple composite state

The behavioral state machine in Fig. 3-8 (some events and actions are omitted for clarity) is mapped into two SystemC threads. The first thread captures the transitions between the states P1, P2 and P3; and the second thread C1 and C2. In the SystemC code, there will be three more states, the final state P4, P0 for the default state which has a transition leading to P1 and C0 for the default state inside state P2. Suppose the component is in state P2 and C2. If event e2 comes, the second thread will execute action on exit of C2, send a trigger to the first thread and move to C0. The first thread upon receiving the trigger from the second thread will execute the action of the transition, move from P2 to P3 and execute action on entry of P3. The trigger among the thread is done through newly created events introduced by the translator. We call them book-keeping events. All the notification of all the book-keeping events are immediate notification, so that a transition is done in within a SystemC delta cycle only, to preserve the run-to-completion semantics.

Although our approach may give rise to a large number of threads in a module, it is unlikely that at high levels of abstraction, a state machine will have a large number of nested levels or many hierarchical states.

We have examined another approach which used `sc_method` instead of `sc_thread` since `sc_method`s are faster than `sc_thread`s. In this case, instead of generating a thread for each OR state, the translator generates multiple methods, each corresponding to an active segment of the thread. This results in a large number of methods but not better simulation speed since once a method is waken up, it has to check whether it is allowed to execute at this point of time, meaning the control has reached the point this method represents. If not, it has to go into sleep mode again.

The algorithm used to generate SystemC code for a behavioral state machine consists of the following: Building the state tree; Processing the states; and Processing the transitions.

Before going into details of the algorithm, we define some important concepts which are used to explain the algorithm. For each behavioral state machine, we introduce a top-most state, which contains all other states, either directly or indirectly.

State tree: for each behavioral state machine, the state tree represents the nested structure of all the states (including pseudo-states), starting from the top-most state. Each node in the state tree contains the information about the state, including whether the state is an AND, OR or simple state, the entry and exit action of the state. Each node also has an operation to get its parent.

Active state: UML defines that a state becomes active when it is entered as a result of some transition. When a simple composite state is active, at most one of its substates is active. When an orthogonal composite state is active, all of its substates are active.

State configuration: As defined in UML, a state configuration is a set of active states starting with the top-most states down to individual simple states at the leaves.

Main source and main target of a transition: are the states that have the same direct father state (the main source and main target may be the same state) and when the transition is taken, the actions of the transition is actually executed when there is a move from the main source to the main target. Note that the main source

and main target of a transition may not be the same as the source and target of that transition.

Least common ancestor (LCA) of two states: in UML, it is the orthogonal state that contains both the states and is nearest to them based on the state hierarchy.

Least common ancestors of a transition: LCA of a transition is the LCA of the main source and main target of the transition.

Source scope: we define source scope of a transition as the set of states which change their status as the result of the exiting of states involved in this transition.

Target scope: we define target scope of a transition as the set of states which change their status as the result of the entering of states involved in this transition.

Processing the states

Upon entering a composite state, there are two possible cases:

- Default entry: the transition terminates on the outside edge of the composite state. In this case, the entry activity of the composite state is executed before the activity associated with the default transition.

- Explicit entry: the transition terminates on a sub-state of a composite state. In this case, the entry activity of the composite state is executed before the entry activity of the sub-state. This rule is applied recursively for nested sub-states if there are any.

Upon exiting a composite state, all the exit actions of the children states must be executed before the exit action of the parent state.

For the states that do not exist in the resulting SystemC code, i.e. all the direct children of AND states, their entry and exit actions are appended to their parents' entry and exit actions following this principle: entryParent+entryChildren and exitChildren+exitParent.

For each of the OR states, an book-keeping event is added to the transition from its initial state to the default state.

Processing the transitions

The simplest type of transitions is the transitions between two leaf states (simple, initial and final states) having the same parent state; because these transitions will only change the value of the state variable. For all other types of transitions, which are called composite transitions, a procedure is used to convert the transitions into the move of the involved threads.

In order to generate the code of several threads to execute a transition, the following algorithm is used for each composite transition in the transition set of each state machine.

1. Eliminate this transition from the transition set of this state machine.

2. Look for the main source, the main target and the LCA of the transition.

3. Between the main source and main target, create a transition with the trigger being an book-keeping event generated from lower level if main source is a composite state or the real event of the transition if the main source is a simple state; the guard being all children have been exited if the main source is a composite state or is the guard of the transition if the main source is a simple state; the action being the action of the transition.

4. Determine and process the source scope of the transition. For all the states in the source scope that exist in the generated SystemC code (direct children of any OR state), create a new transition from this state to the initial state at the same level in the state tree with the following properties. If this is a simple state, the guard is the guard of the transition; otherwise the guard is that all of its children having been exited. The action consists of triggering the next higher level state to exit (if the next state is within the source scope or the main source).

5. Determine and process the target scope of the transition. Target scope includes

targetSources and targetTargets. For each pair of targetSource and targetTarget, create a new transition from the targetSource to the targetTarget, with the trigger being the book-keeping event triggering the entry of that level, the guard being the source state of this transition was active at the beginning of the delta cycle, and the action being that of sending an book-keeping event to trigger the entry of the next lower level.

The algorithms to determine the main source, main target, the LCA, the source scope and the target scope of a transition are straightforward and hence we omit them.

SystemC code for event communication

As mentioned earlier, to execute the cross-level transition and ensure the action execution order, it is necessary to synchronize the threads. Some book-keeping events are inserted by the translator into the model for this purpose. These book-keeping events have higher priority than other events to maintain the run-to-completion semantics. The priority is implemented by the order in which the events are checked. When an object is waken up, it will check all the book-keeping event queues first before checking the normal event queues.

Some of the issues to be faced in implementing events are:

- the ordering of event triggering and receiving so that the actions are executed and the states are exited/entered in the correct order.
- events together with arguments to an object are queued and processed based on event types.
- an object may be waiting for several events at the same time, after waking up the object must be able to detect which event triggered it.
- if multiple composite states (threads) in a module are waiting for an event, all of them must be able to be triggered by the event and read its arguments correctly.

It's only after all the waiting threads have been triggered by the event and read the arguments that the event is dequeued. The challenge is that at translation time the translator does not know in advance which threads and how many of them will be waiting for an event.

The SystemC `sc_event`s do not satisfy the above requirements. First, when a thread is waiting for several `sc_event`s and waken up, it cannot check which `sc_event`(s) has come. And there is no mechanism to queue up the `sc_event`s and their arguments.

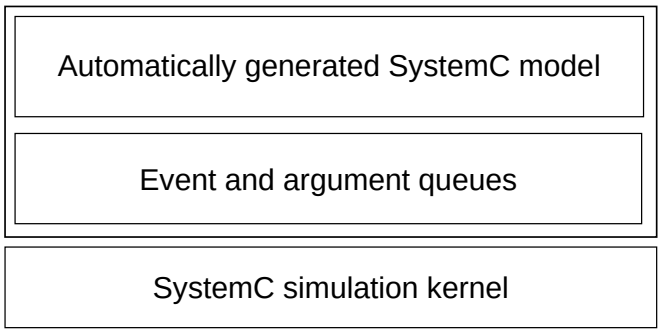

Figure 3-9: SystemC simulation layers for reactive systems

Thus, we have implemented our own event and argument queues above the SystemC `sc_event`s. We use `sc_method`s instead of `sc_thread`s of `sc_cthread`s in communication channels for better simulation speed. We also use the `request_update()` and `update()` functions of the SystemC primitive channel for the updating of the event and argument queues. This makes sure that all the threads processing an event and its arguments are able to get the correct event and arguments. The dequeuing only takes place at the end of the SystemC delta cycle, i.e. after all the threads have finished their processing.

In section 2.4 we explained transaction level modeling (TLM). In short this is an abstraction of communication so that the details implementation of the communication is not taken into account, instead each communication transaction is modeled as a function call. SystemC code at the TLM level is ideal for simulation as details of

the low level communication infrastructure are not present. In our design flow, users do not have to specify any SystemC components at UML level. They can simply work with classes or objects, state machine diagrams and model communication between objects by events with or without arguments. Such events will be implemented through SystemC primitive channels. Each module has a primitive channel to receive events sent by other modules. This primitive channel essentially acts as a buffer for incoming events to that module. When a module has an association relationship with another module, it can send messages to that module. A port is declared in the sender module to access the other module's primitive channel. Thus, the primitive channels implement two interfaces: the interface for sending events and the interface for receiving events. The SystemC code generated by our translator is at the TLM level since the senders and receivers just call functions of the primitive channels, regardless of whether or not the events have arguments associated with them.

Translator implementation As mentioned earlier, Rhapsody [100] is a tool that is used for designers to model their system using UML notations. Rhapsody saves the model in some files. These files are transformed to XML by our translator. While transforming, the hierarchical structure of information is maintained. Hence, the structure of the XML files are similar to that of the Rhapsody internal representation.

From the model that includes the state machine diagram in Fig. 3-7, an XML document is generated from the Rhapsody internal representation. Here is a fragment of the rather verbose XML code that describes the transition from state `s1` to state `s2`. It is not meant to be readable. We are including it merely to give a flavor of the XML representation.

```
<ITransition id="c3d8318a-8dd1-4f49-ab5c-2a0301f5f6db">
    <_myState>2048</_myState>
    <_name>"transition_3"</_name> -
    <_itsLabel id="e5cf97ea-6f81-4fca-9988-ea8b8e3f9dc6">
        <_itsTrigger id="e94f5561-cb5f-4009-9e92-027e060c0917">
```

```
            <_body>eventA</_body>
            <_info>""</_info>
            <_itsInterfaceItem id="c3a4c0c9-6897-47c9-8f2d-0b7a63d3a869">
                <_m2Class>"IEvent"</_m2Class>
            </_itsInterfaceItem>
        </_itsTrigger>
        <_itsGuard>NULL</_itsGuard> -
        <_itsAction id="9fc8a233-4906-420d-aff6-e6a23ed029a9">
            <_body>functionA();</_body>
        </_itsAction>
    </_itsLabel>
    <_itsTarget>GUID dcf4515d-77c4-4cd9-805d-a84909da5d21</_itsTarget>
    <_staticReaction>0</_staticReaction>
    <_itsSource>GUID04e1e2c1-b754-4d93-9304-79f4a7e73ef2</_itsSource>
</ITransition>
```


XML is a good intermediate representation for our translator because it is easy to create and extract information from XML documents. However, XML itself is not a convenient specification language for system-level designs. In particular it is textual, tedious and error-prone to be used for complex designs.

We then use our XML parser (implemented using JDOM [4]) to gather information from the XML document to build an abstract tree as an input to a template engine called Velocity [13] that generates SystemC code from predefined templates. With the help of this engine, we are able to decouple the parsing of XML document from the code generation step so that changes in the XML structure, and thus the XML parser do not affect code generation process. Further, in the later part of the work flow, we only need to work with the templates to generate code without having to deal with the verbose code of the parser. Consequently, by merely modifying the templates for one level of abstraction, we can have the templates for another level of abstraction and generate code, without touching the XML parser. Fig. 3-10 shows the work flow of our translator.

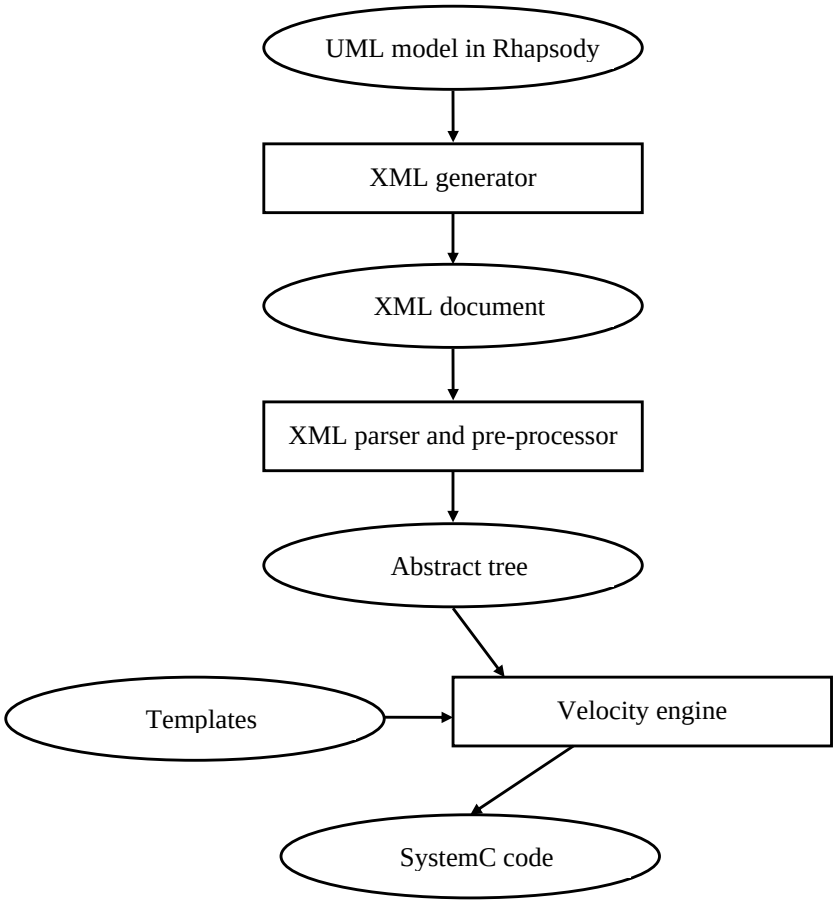

Figure 3-10: Our implementation workflow

We show here a simplified fragment of a template that is used for the state machine diagrams.

```
   switch (${orState.getName()}_state->read_temp())
   {
#foreach ($state in $orState.getChildrenState())
   case $state.getStateId():
      $state.getActionOnEntry()
      . . . ## wait for both normal and book-keeping events
#foreach ($transition in $state.getOutgoingAdminTransitions())
      if (this_uport->get_$transition.getTrigger()_flag() == true)
      {
         if ($transition.getGuards())
         {
            $state.getActionOnExit()
            $transition.getAction()
            state = $transition.getTarget();
            break;
         }
      }
   }
#end #foreach ($transition in $state.getOutgoingTransitions())
      if (this_uport->get_$transition.getTrigger()_flag() == true)
      {
         if ($transition.getGuards())
         {
            $state.getActionOnExit()
            $transition.getAction()
            state = $transition.getTarget();
            break;
         }
      }
   }
#end
```

```
#end
```

The following is the SystemC code which will be generated for the simple state machine diagram in Fig. 3-7. For the sake of readability we present this program in pseudocode form.

```
current_state = initial_state; while (current_state != final_state)
{
   switch(state) {
   case initial_state:
      wait for trigger from the top module to start behavior
      actionT;
      current_state = s1;
      break;
   case s1:
      actions on entry of s1
      wait for eventA or eventB to come
      if eventA comes {
         if (true) {  // the guard of this transition is true
            actions on exit of s1
            functionA();
            current_state = s2;
            break;
         }
      }
      if eventB comes {
         if guardB is true {
            actions on exit of s1
            functionB();
            current_state = s3;
            break;
         }
      }
      break;
```

```
   case s2: . . .  break; case s3: . . .  break;
   }
}
```

The code of `actionT`, `functionA` and `functionB` are to be provided by the users. The SystemC implementation of actions for events depends on the levels of abstraction which will be discussed in the following subsections.

3.3.2 Translation to behavioral level

We have also experimented with the generation of behavioral level SystemC code using the CoCentric tool of Synopsys [1]. We were able to access this tool at the point of time of this work. However, this tool is no longer available and supported by Synopsys. The currently available synthesis tools are presented in section 2.4. Similar to the CoCentric tool, they support only a subset of SystemC.

The Synopsys tool places rather severe restrictions at the Rhapsody level on the C++ code fragments supplied by the user. Further, one has to declare a class called `Top` to initialize all the instances since the method `new()` used to create instances is not synthesizable. One may however initialize multiple instances of the same class. The translator will create the corresponding modules and connect them according to their specified relationships.

The code synthesized at this level has to comply with the coding convention of the Synopsys tool. Restrictions are placed on the data types, constructs, instructions and SystemC classes [1]. Due to these restrictions, a simple composite state is translated to an `sc_cthread` which is only sensitive to an active clock edge. Further, communication is achieved only through signals. UML events are implemented as `sc_signal`s which can have boolean values representing the existence of the events. We show an example of the behavioral level code in the next section.

3.4 Case studies

3.4.1 A simple bus

This is a benchmark example of SystemC at the TLM level which has been described partially in the previous section. Here we use it to demonstrate how one might model a fragment of a platform at UML level and translate it into SystemC. This model uses all the four stereotypes mentioned in section 3.2, namely modules, interfaces, primitive channels and hierarchical channels. In particular, the three masters are modules that access the bus through three ports using three different interfaces. The bus is a hierarchical channel which implements the methods of the three interfaces.

For faster simulation speed, the arbiter and the fast memory are modeled as primitive channels to decrease the number of threads and thus, decrease the context switching time. The bus accesses these primitive channels through the arbiter interface and the slave interface, respectively. Fig. 3-11 and Fig. 3-3 show the class diagram and structure diagram of this example.

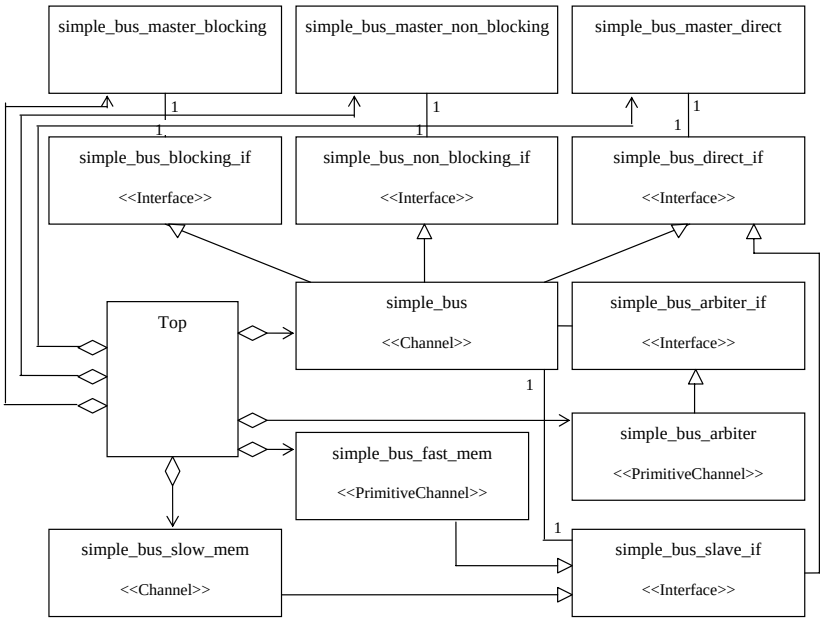

Figure 3-11: The class diagram of the simple bus example

We first captured this UML level model using the Rhapsody tool and then translated it into TLM SystemC code using our translator. We then simulated the resulting SystemC code using the standard SystemC simulation kernel. When measuring performance, we did not initialize the direct master, because it is only used for debugging. The experiments were performed on Linux Red Hat 9.0 running on CPU Intel Xeon 2.8GHz. We measured the number of CPU clock cycles for 1,000 bus transactions using the Pentium's `rdtsc` instruction. With the original code provided in the SystemC public distribution, we obtained a speed of 81K transactions per second. In comparison, with our automatically generated code from the UML model, we obtained a speed of 41K transactions per second. One reason for the slower simulation speed of our generated code is the use of `sc_thread` for all processes. The original model has the bus and the slow memory implemented as `sc_method`s. Due to the need for context switching, `sc_thread`s run slower than `sc_method`s.

3.4.2 A micro polymerase chain reaction controller

This is a simple realtime controller. Polymerase Chain Reaction (PCR) is a thermal cycle reaction used for the rapid *in vitro* multiplication of DNA samples [83]. The μ-PCR chip realizes a miniaturized version of this process where a small quantity of the DNA sample is placed in each chamber of the chip and the PCR reaction is achieved by controlling the thermal power supplied to the chambers according to an input temperature profile. A schematic diagram is shown in Fig. 3-12.

We will not describe here the PCR biochemical process in detail but instead focus on the functional model of the controller. This unit is driven by the temperature profile, which specifies the control objective, and feedback received from the chip regarding the current temperatures of the chambers. In the present version of the plant model, the effects of inter-chamber influences are ignored as a simplification. Hence there is one independent controller for each chamber. This controller periodically reads the temperature, converted into a voltage value via an analog-to-digital con-

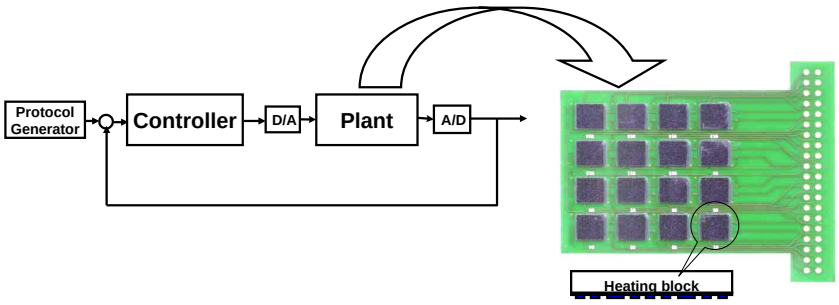

Figure 3-12: The μ-PCR block diagram

verter, of the chamber. With the help of the estimator — the control law — it then computes the output voltage required for that chamber to maintain the temperature according to the temperature profile of the current PCR thermal cycle. This voltage is then converted back into an analog value via a digital-to-analog converter, which is subsequently used to control the heating element of that chamber.

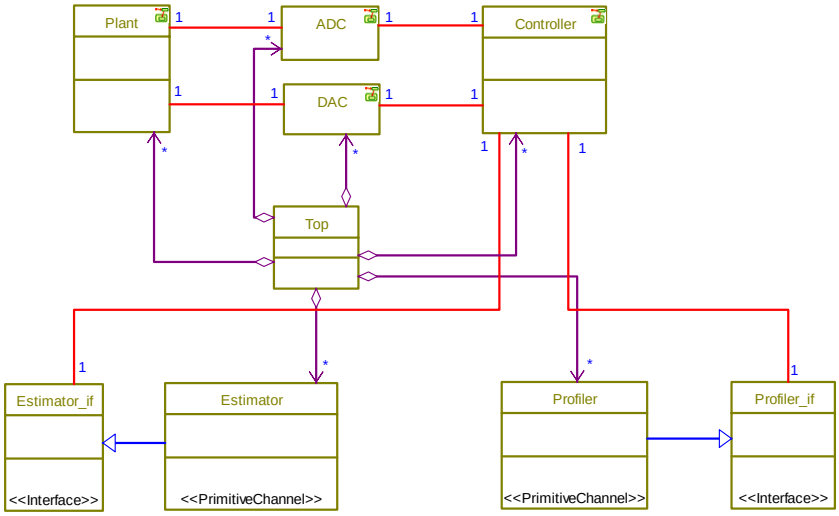

Figure 3-13: The μ-PCR class diagram

Fig. 3-13 shows the class diagram of this example. The profiler that keeps the temperature profile and the estimator that keeps the control laws were modeled separately from the controller so that we can reconfigure the temperature profile and control law easily. They were modeled as primitive channels in order to get better

simulation speed at the TLM level. The communication among the modules are cycle accurate, in the sense the status of a module's input and output are specified at each clock cycle. Yet another real time aspect is the timing diagram associated with the A/D converter. The state machine diagram of the controller is shown in Fig. 3-14.

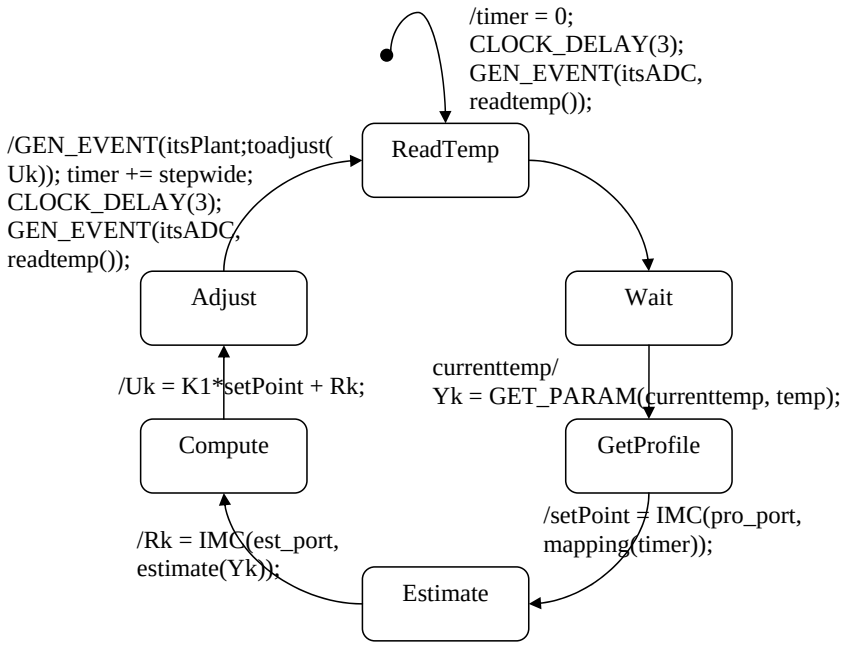

Figure 3-14: The state machine diagram of the μ-PCR controller

For this example, we have synthesized, using the CoCentric compiler tool of Synopsys, the behavioral level SystemC code generated via our translator. This application has been simulated at both TLM and behavior levels.

Following is a fragment of the code for the `sc_cthread` of the controller at behavior level. For brevity we have eliminated some `wait()` statements.

```
while (true) {
        switch (state) {
        case 106: // Read_Temp state
                //wait for ack signal from ADC
                wait_ack();
                if ( read_ack() == true ) {
```

```
                //the guard of this transition is true
                if (true) {
                   state = 118;
                   write_ack(false);
                   break;
                }
                else
                   write_ack(false);
             }
             break;
      case 110: //GetProfile state
             // no event trigger for this transition
             if (true) {
                //IMC is a macro for an interface method call
                //to the port pro_port to the profiler
                //the method mapping has an argument timer
                //the returned value of this method
                //is assigned to variable setPoint
                setPoint = IMC(pro_port, mapping(timer));
                state = 115;
             }
             break;
      // the initial state
      case 119:
             //wait for signal to start the execution
             //of this process
             wait_initController();
             timer = 0;
             GEN_EVENT(itsADC, readtemp());
             CLOCK_DELAY(3);
             state = 106;
             break;
      }
}
```

The implementation of the functions like `mapping()` and `estimate()` is provided by users; they can only use the synthesizable subset of SystemC defined by Synopsys. Note that although SystemC interfaces and channels are not synthesizable by CoCentric Synopsys, we can still generate behavioral level SystemC code from the models that have interfaces and channels like the one in this example. In this case, the model's channels are declared as `sc_module`s at SystemC level. The interfaces in the UML models are not generated, instead each channel/module that implements an interface has a thread that receives triggers for method calls from other modules, locally calls the methods and returns values by sending signals to the caller modules.

Fig. 3-15 shows the simulation speed — in terms of transactions per second — of the μ-PCR example on the same platform as the one used in the previous example. By a transaction we mean the period of time in which the controller senses the current voltage, computes and outputs to the plant.

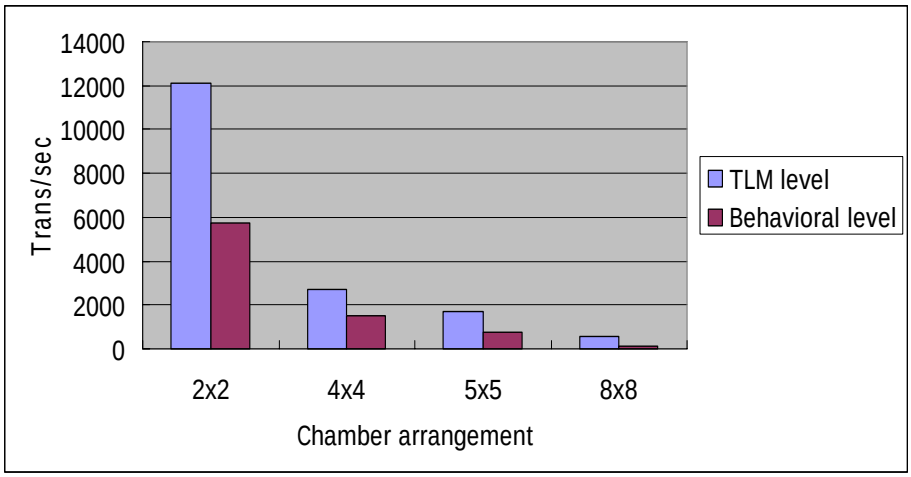

Figure 3-15: Simulation speed of the μ-PCR example

Our simulation results show, as expected, that simulation speed at the TLM level is higher than that at the behavior level. The most important reason is that at TLM level, communication is done through function calls only and the timing information is captured as units of time, not clock edges. While at behavioral level, communication is done through signals and the behavior is aligned with clock edges.

The experiments also give evidence that the code we generate scales fairly well in terms of performance.

3.4.3 A digital down converter

For our third example, we implemented a *digital down converter* (DDC) for the *global system for mobile communications* (GSM) - a wireless communication protocol. Digital radio receivers often have fast analog to digital converters delivering vast amounts of data. However, in many cases, the signal of interest represents a small proportion of that bandwidth. A DDC is a filter that extracts the signal of interest from the incoming data stream. Our implementation closely follows the MATLAB example in Xilinx's system generator (see Fig. 3-16).

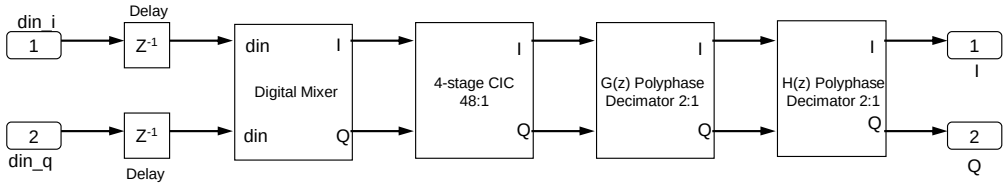

Figure 3-16: The block diagram of the digital down converter for GSM

The desired channel is translated to baseband using the digital mixer comprised of multipliers and a direct digital synthesizer (DDS). The sample rate of the signal is then adjusted by a multistage, multirate filter consisting of a cascade integrator-comb (CIC) filter and two polyphase finite impulse response (FIR) filters with a decimation factor of 2. The functions performed in the system are complex multiplication and multirate filtering. The overall down sampling rate of the converter is 192:1.

Each of the components is mapped to a module, and data is sent through the chain by events (see Fig. 3-17). The model has been translated into both TLM and behavioral levels. We could not find the source code for a similar DDC in UML or SystemC for comparison. Hence we could only compare the FIR module of our design with an FIR example provided by Synopsys. The only modification we did to the

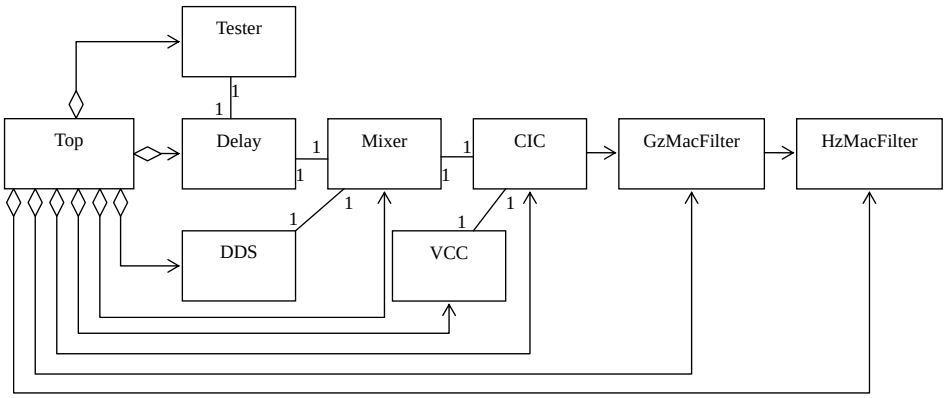

Figure 3-17: The class diagram of the digital down converter for GSM

Synopsys code was to ensure that the coefficients and the bit widths of the ports are the same as those of our FIR model. The codes were compiled into gate level netlist using Synopsys `tc6a_cbacore` library, which targets cell based array architectures [116]. The same timing constraints were used on the synthesis runs of both. The following table shows the comparisons of the final synthesized hardware. From the result we can see that our generated code uses about 33.25% more resources than the hand coded version. This is a preliminary result and further study is necessary in order to reduce this difference.

	FIR from Synopsys(S)	FIR from DDC(D)	Ratio((D-S)/S)
Number of ports	260	261	0.39%
Number of nets	18393	27942	51.92%
Number of cells	18010	27547	55.15%
Number of references	93	99	6.45%
Combinational area	30181.2	50583.7	67.60%
Noncombinational area	34560.0	36844.2	6.61%
Net interconnect area	244806.2	325033.1	32.77%
Total cell area	64741.1	87430.3	35.05%
Total area	309547.6	412461.1	33.25%

Table: Area statistics for FIR component implemented on cell based array architecture

3.5 Summary

Our main contributions in this chapter are:

- A method to synthesize SystemC code from arbitrary nested behavioral state machines.
- The ability of our framework to model both platforms and applications at a high level of abstraction.
- In addition, there exists a good deal of potential to further refine the generated SystemC code to hardware implementation, although more research, tools and techniques are necessary.

Although our translator generates code from Rhapsody internal representation, our translator is not restricted to the Rhapsody tool because we decoupled the part

that depends on the structure of the Rhapsody internal representation to other parts of the translator. Hence if there is any change in the internal representation, we only need to modify the file defining the structure of the XML file.

Chapter 4

UML-based design for time-triggered systems

Today, many new innovative functions in cars are enabled and driven by software, such as energy management and the current step into hybrid solutions [27]. More than 2000 individual functions are realized or controlled by software in premium cars. However, software engineering in cars is still in the preliminary state, although software appeared in cars 30 years ago [27].

In the recent years, several functionalities in cars have been implemented on time-triggered architectures (TTAs). In these systems, all actions are carried out at predefined points of time. Thanks to this time predictability, TTAs has become strong candidate platforms for safety-critical real-time applications. Their application domains not only include automotive but also aerospace, and railway applications.

Designing applications to run on TTAs is a challenging task. System-level design is an attractive approach for the design of time-triggered applications. The application of system level design in the domain of TTAs involves the choice of suitable models for time-triggered software tasks and platforms, effective intermediate representation and last but not least, fast simulation techniques. In this chapter we discuss our UML-based framework for the design of time-triggered applications to be executed

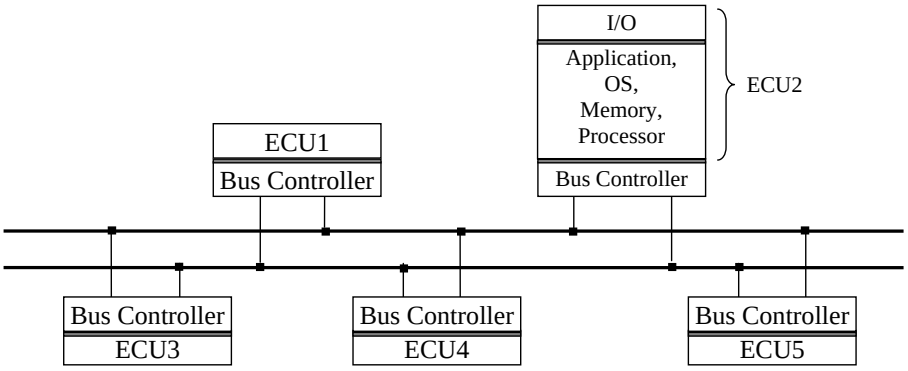

Figure 4-1: A time-triggered architecture

on TTAs.

4.1 Time-triggered architectures

A typical time-triggered architecture is constituted by one or more clusters. Each cluster consists of nodes, usually called engine control units (ECUs) communicating with one another via a time-triggered communication protocol. Two different physical interconnection topologies within a cluster are bus and star. In a TTA bus, the connection consists of replicated buses as shown in Fig. 4-1. ECUs can also be connected in a star configuration [77]. There are several time-triggered protocols such as FlexRay [55], TTP [78] and TT-CAN [52]. Among them, FlexRay has backing from many major automotive companies. We think that it is becoming the de-facto standard for automotive communication systems soon.

All the ECUs in a time-triggered cluster must have the same notion of time. Thus, the clocks on the ECUs must be synchronized. In addition, all the ECUs must have a consistent view on the "health" of every ECU. This is important for the efficiency and correctness of the applications. The clock synchronization service is usually implemented at the communication protocol layer. The membership service may be implemented at either the protocol or the application layer.

An ECU consists of a processor, a memory management unit and a communication interface to which a bus controller is attached. Often, an ECU will also have sensors and actuators associated with it. The bus controller mediates between the ECU and the communication channels and implements the key communication-related functionalities of the ECU. These functionalities include the scheduling of message transmission, assembling/deassembling messages, coding/decoding, performing physical access to the busses and clock synchronization.

In developing distributed applications that have hard real-time constraints, the time-triggered paradigm advocates a number of design principles [50] to be followed:

- **Temporal firewall:** This design principle requires sender tasks to push data onto the time-triggered communication platform and receiver tasks to pull data from this platform. A sender does not send any control signal directly to a receiver.

- **Global time:** All the local clocks in a TTA cluster must be synchronized sufficiently often to establish the global time of the cluster. The granularity of the global time g must be greater than the precision achieved via the clock synchronization mechanism.

- **Composability:** This principle covers several aspects. First, it requires that nodes can be designed independently of each other assuming that the architecture and service have been specified precisely. Secondly, independently developed components can be integrated with minimal integration effort.

Applications targeted to run on TTAs must comply with the timing requirements imposed by the underlying time-triggered communication protocol. Further, the applications developer must be aware of architectural support that may be available to support the time-triggered communication fabric. Finally, basic design principles such as temporal firewalls (no control signals shall flow across communication interfaces) may need to be enforced in the process of developing applications. Thus one

needs a design framework specifically geared towards TTAs. Such a tool must satisfy the following requirements:

- Designers can add more functionality to the existing architecture easily by
  - re-scheduling the communication among the ECUs.
  - re-scheduling the tasks to be executed inside an ECU.
- Designers can integrate components from different suppliers quickly, since it is very common for automotive, aerospace and railway device producers to buy and use sub-components from other suppliers.
- Components can be replicated at different nodes in a distributed system to support fault tolerance.
- Pre-designed components can be re-used to improve design productivity.
- The system being designed can be verified through simulation or formal verification to make sure that it functions correctly in both the value and time domains.

4.2 Our contributions

At the top layer we propose a menu of UML-based notations using which applications can be rigorously specified and key parameters of the underlying TTA and communication protocol exposed.

There are a variety of TTAs and associated protocols [106]. Here we have based our ideas on the FlexRay standard. Due to strong backing from the automotive industry [55], it is likely to emerge as the industry standard for a time-triggered in-vehicle communication system. Until recently, work on FlexRay has focused on defining and validating the standard and developing appropriate communication hardware. With this the stage close to completion, one can begin to develop actual applications. In

this context, a UML-based backbone such as the one we advocate here can provide necessary levels of abstraction and standardization to support reuse.

We expose the key aspects of the FlexRay at the UML level by introducing appropriate stereotypes, tagged values and more importantly, using an activity diagram to specify the static schedule defined by the protocol. Through a disciplined use of ports and interfaces we also enforce the temporal firewall principle required by time-triggered protocols [50]. This principle demands that only data values but no control signals should flow across communiction/component interfaces of TTA architectures. We also provide preliminary evidence suggesting the composability quality enjoyed by the TTA approach can be reflected at the UML-level. We do so by adding a cruise control application to an initial design that consists of a brake-by-wire application.

Finally, we demonstrate a translation to SystemC for initial validation. A key advantage of our translator is that it automatically synthesizes – using information gathered at the UML layer – a *simulation driver*. This driver mediates between the kernel simulator of SystemC and the generated SystemC code in order to achieve fast simulations.

It is worth pointing out that we do not address the issue of how/whether the generated SystemC code can be used in the software development process. The worst case scenario is that the SystemC code is used only for prototyping and initial validation. Even in this case, it can serve as a valuable design specification for starting the code generation process.

Though we have carried out our work in the specific setting of FlexRay, our approach can be adapted to handle other time-triggered protocols since the design of the application is separated from the communication platform layer by ports and interfaces. In fact, the design of the applications depends only on the fact that there is a communication layer that makes sure that messages are transferred to the bus in a time-triggered manner.

4.3 Related work

UML has been proposed to be used in the design process of automotive software applications in different settings. For instance, it has been used for system specification [101, 84] but without mechanisms for validation and synthesis. On the other hand, AnyLogic [53] supports specifications via UML-RT [111] based modeling, generates Java code and executes animated models. Our approach differs from this in that we use standard UML notations with a few extensions. Secondly, we support simulations by automatically generating SystemC code which provides a standard executable intermediate representation. Next we note that AML (Automotive Modeling Language) [26, 25] is also a related UML-based modeling framework that supports multiple abstraction levels. The key difference is that we have a path to an executable intermediate representation that allows functional validation. In addition, our code generation process takes advantages of the special features of TTAs so as to achieve good simulation speeds.

A number of toolsets that support time-triggered protocols are currently available. Typical examples are: TTTech's [11] toolset for the TTP protocol, TTAutomotive's [119], Decomsys' [40] and dSPACE's [115] toolsets for FlexRay. In general, these toolsets deal with the design of TTAs at a more concrete and platform-specific level. These tools support the modeling and simulation of applications in Matlab/Simulink. Code running on target platform is generated. An important future challenge will be to bridge the gap between this lower level and the UML-level abstraction we propose.

Model-driven development environments based on SCADE and built on top of TTPPlan and TTPBuild have been proposed [46, 34]. The idea is to model time-triggered software tasks using the GUI of the SCADE tool. The SCADE Code Generator will then generate code that can be run as time-triggered OS tasks. We feel that UML 2.0 notations are an attractive alternative since they are more generic, offer a variety of diagram types to deal with both the structural and behavioral aspects of applications. Further, UML, unlike SCADE, is not strongly tied to the synchronous

programming paradigm.

STEP-X [88] uses UML notations and targets general automotive applications. It does not cater to the specific needs of TTA-based applications. A UML profile for TTAs is defined in [85] with a proposed mapping from designs developed using this profile to the tools TTPPlan and TTPBuild. Our work shows that the standard notations of UML 2.0 – with a mild dose of stereotypes and tagged values – suffice for high levels of abstraction.

More generally, Giotto [71, 58] offers a software layer for specifying and composing time-triggered tasks. This level of abstraction uses logical time rather than physical time. In contrast, at the UML-level, we use physical time as dictated by the static schedule of the FlexRay protocol. Consequently, in our framework, it will be much easier to relate timing behaviors at the implementation and specification layers. Finally, the AUTOSAR [3] consortium has been working actively on proposing an overall software architecture standard for automotive applications. Our modeling framework is more specific in that it is targeted towards time-triggered platforms.

4.4 UML-level modeling

A key issue we must address is how to expose the platform's architecture and services at UML level. In our framework we focus on the choice of diagrams, their intended role and the required usage pattern so that rigorous specifications that adhere to the above principles can be developed. Moreover, the key features of the architecture and the communication protocol are blended into the specifications. This is done in order to facilitate the automatic generation of SystemC code that can be efficiently simulated. As mentioned earlier, we will focus here on a specific embodiment of the TTA principles, namely, the FlexRay standard [54].

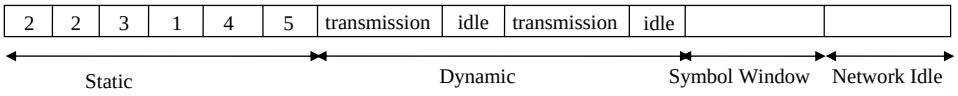

Figure 4-2: FlexRay basics

4.4.1 The FlexRay communication platform

FlexRay [55] is a communication protocol for real-time distributed systems. It is implemented on time-triggered architectures in which several ECUs are connected to one or two communication channels as shown in Fig. 4-1.

The ECUs transmit data to each other mainly in a time-multiplexed fashion as dictated by the FlexRay protocol. The protocol executes in recurrent periodic cycles called *communication cycles*. The protocol is implemented using a four level timing hierarchy. Each cycle at the top level consists of a static segment, an optional dynamic segment, an optional symbol window and a network idle time (see Fig. 4-2).

- Data generated by tasks running on the ECUs are sent and received in the *static segment* in a time-triggered fashion. More precisely, write access to the bus by the ECUs is scheduled according to a fixed *time division multiple access* (TDMA) scheme during the static phase. Consequently, an ECU is granted exclusive write access to the bus at exactly specified time intervals called the *static communication slots*. All slots have the same time duration and exactly one frame is to be transmitted per slot. Further, the order of allocation of slots to ECUs in the static phase is identical for all cycles.

- The *dynamic segment* is in some sense an event-triggered phase where, using a priority scheme, the ECUs can be scheduled to transmit messages of varying time duration measured in terms of *mini slots* in a time-deterministic fashion (see [54]).

- The *symbol window* is used to transmit special messages such as “cluster-wake-

up".

- The *network idle time* phase is used to calculate and apply clock correction terms in order to achieve clock synchronization.

At the lower levels of the FlexRay timing hierarchy, static and dynamic slots consist of a fixed number of *macro ticks*. Each macro tick consists of a fixed number of *micro ticks* generated by a local clock.

Exposing communication features at UML level

For the purpose of application development, macro and micro ticks are not exposed at the UML level. We also do not consider the symbol window and network idle time segments since they do not directly influence the functionalities of the application tasks. In effect, the applications are modeled with the assumption that clock synchronization is assured by the underlying communication platform.

The features of the protocol that we expose at the UML level include:

- The length of each segment. Although we do not consider the activities happening at the communication layer during the symbol window and network idle time segments, their length exposed at UML level can be larger than 0.
- The number of slots in the static and dynamic segments and their lengths.
- The owner of each static slot (which is also the order in which the ECUs' messages are scheduled in the static segment).
- The ECUs that may send or receive in the dynamic segment (in order to optimize simulation speed at SystemC level).

The above information is captured in a UML activity diagram associated with a usecase (the bus communication usecase in Fig. 4-3) depicting how the communication platform is used by the software applications. Such an activity diagram is shown in Fig. 4-4. Activity modeling emphasizes the sequence and conditions for coordinating

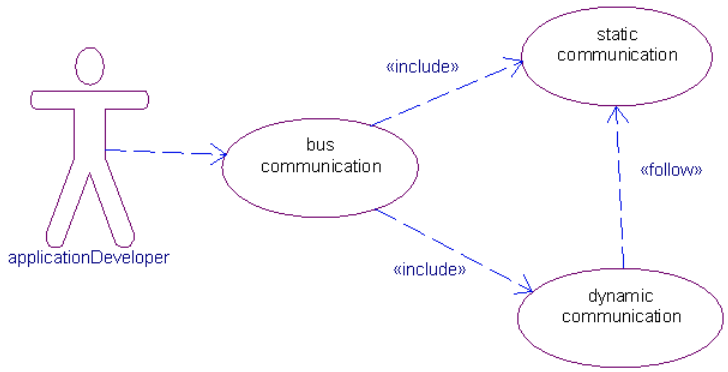

Figure 4-3: The usecase diagram describing the services provided by the communication platform

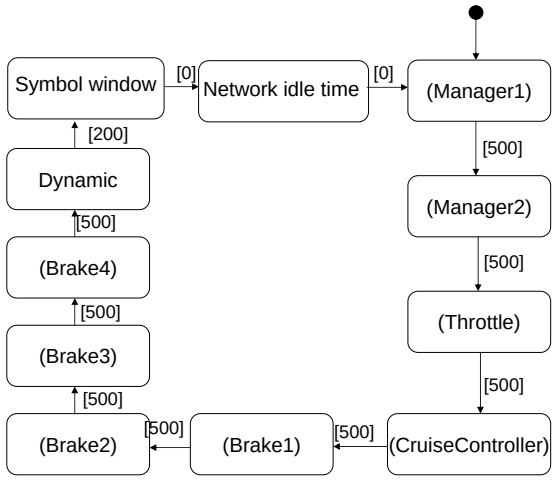

Figure 4-4: The activity diagram describing the communication cycle

behaviors of different objects. In Fig. 4-4, a token which determines the activity to be executed starts at the initial activity. Thus, `Manager1` occupies the bus for 500 micro-seconds before `Manager2` is granted access to the bus. Similarly, `Manager2` is given a 500 micro-seconds slot on the bus, etc. In this example, the dynamic, symbol window and network idle time segments of the FlexRay protocol have been assigned 200, 0, and 0 micro-seconds, respectively. The activity node `Dynamic` has a tagged value for the length of each mini-slot inside the dynamic segment and another tagged value which is a boolean expression on `ids` of nodes, describing all the nodes that may send or receive during a dynamic segment. All the above information is required at

the application layer to ensure that the tasks are designed to communicate on time. (Due to separation of concerns, we do not consider here how this is achieved). This information is also used by our translator to automatically configure and initialize the communication controller of each node at the SystemC level. We will say more about the communication controllers later in this section.

4.4.2 Modeling technique

Structure modeling In order to model the software task(s) at each ECU, the structure of the entire time-triggered cluster must be made available. This is because in order to describe tasks and their relationships to the cluster's communication schedule, the application developers need to know on which node a task is located and which nodes the task communicates with are located.

We use *class diagrams* and (*composite*) *structure diagrams* to capture the architecture of the computing and communication infrastructure of the system. As mentioned in section 2.3, a class is an abstraction for those components of a system that have the same behavior and class diagrams are used in the standard way to describe the relationship among the classes. Structure diagrams are used to capture the nested structure of components.

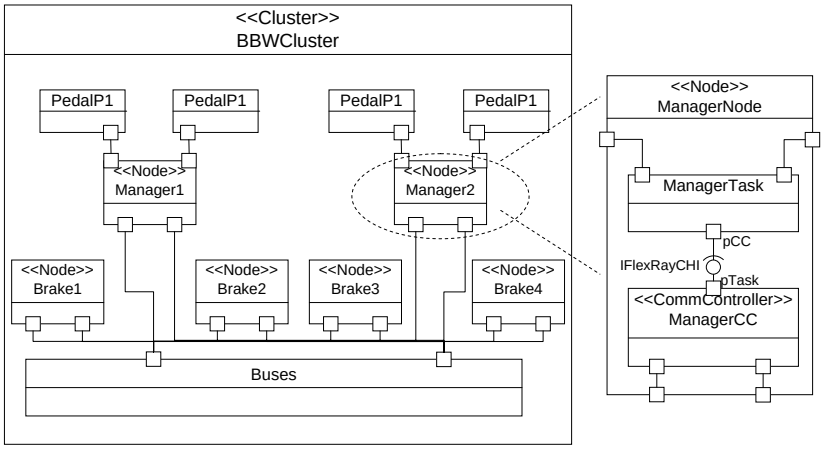

Figure 4-5: The composite structure diagram of a BBW cluster

Fig. 4-5 is a structure diagram depicting the layout and interconnections of a cluster consisting of six ECUs for a Brake-By-Wire (BBW) application. Each object in a structured class is instantiated from other classes. These objects may be connected through association links. The links connecting them show the possible ways they may interact with each other.

An important feature of structured diagrams are the ports with *provided* and *required* interfaces. This allows for a natural description for the provision and use of services by the components of a system. In Fig. 4-5 the socket notation refers to the required interface `IFlexRayCHI` of port `pCC` while the circle notation refers to the provided interface `IFlexRayCHI` of port `pTask` in the link between object `ManagerTask` and object `ManagerCC`. The `IFlexRayCHI` interface consists of functions for the task `ManagerTask` to call when it wants to send/receive data from the communication controller `ManagerCC`. The required and provided interfaces of two ports connecting to each other must match. In addition, only data-flow can take place across these interfaces and the interfaces include functions to push data to the communication controller to send and pull data from the communication controller to receive. These basic restrictions are imposed in order to satisfy the temporal firewall design principle of TTAs.

Structure diagrams allow hierarchical structures to be captured. For example, one of the classes (instantiated twice in the left half of Fig. 4-5), namely the `ManagerNode`, has an internal structure shown in the right half of Fig. 4-5, which is actually the structure diagram of the `ManagerNode`.

Behavior modeling In the present context, application tasks are triggered at specific time instances. Hence, the transitions of our behavioral state machines will be time-outs that denote the amount of time an object has stayed in the source state during which it would have executed the code associated with that state. The time here is the global time which is assumed to be common knowledge to all the nodes.

Guided by these considerations, the behavior of a task assigned to a node will be modeled as a behavioral state machine which at the top level will consist of a *single loop* of states and transitions. According to our observation, the sum of all time-out expressions in this loop must be a multiple of the FlexRay cycle. Thanks to the information on the communication protocol provided by the activity diagram, the designer is able to make sure that the application sends data to the lower communication layer before the node's allocated slot and reads incoming data at the correct slots.

The time-out expressions associated with the transitions can be constants or arithmetic expressions whose variables must be initiated in the object's constructor. Fig. 4-6(a) shows an example of the top level of a behavioral state machine. The notations at the top right hand side of the states indicate that the states have actions to be performed on entry. This action can be any C++ (or SystemC) statements which include function calls through ports to transmit or receive data from the communication platform. This state machine models the fact that when a brake actuator object enters the state `start`, it will execute the state's action on entry. It will stay in this state for 4,000 micro-seconds after which it will take a transition and enter the `dynamic1` state in which it may want to communicate with other nodes in an event-triggered manner.

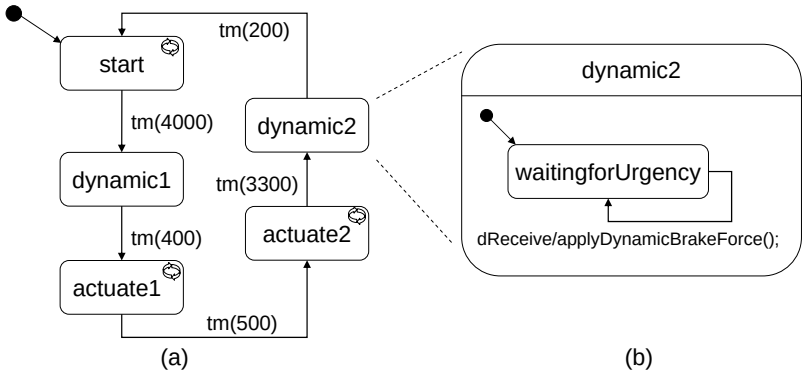

Figure 4-6: The behavioral state machine of the brake actuator

Compared with our approach of modeling time-triggered behavior, another ap-

proach [112] which models the clock module explicitly and this module has to generate clock ticks events to other components in the systems. In addition, there is an interface for other components to query about the time. We think that this fine grain timing modeling is not suitable for modeling at high level of abstraction and simulation at this level of granularity is extremely slow and not good for system level modeling. In fact, we have modeled the BBW application running on FlexRay at this level of granularity and found that it took huge effort to model and debug the system and the simulation speed was unacceptably slow.

Time-triggered transitions in behavioral state machines alone are not enough due to the event-triggered nature of the behaviors allowed in the dynamic segment. To cope with this, we allow the top-level states corresponding to the dynamic segment to have internal states that will capture the event-triggered reactions restricted to the dynamic segment. Fig. 4-6(b) shows the internal sub-states of state `dynamic2`. When the system is in the dynamic segment, the brake actuator objects wait for an event coming from the communication platform that signals that the car needs to brake immediately. This event can be sent and received only in the dynamic segment and that too only if emergency braking is necessary. If such an event is received, the function `applyDynamicBrakeForce()` is called. Strictly speaking, this is a violation of the temporal firewall design principle as control signals (events) are passed across communication/component interfaces. However, FlexRay violates this principle only within the dynamic segment. Further, even within this segment, the event-triggered signals are generated according to a fixed static priority scheme and users must fix the signals' lengths. Hence temporal determinacy is largely preserved and temporal non-determinacy is confined strictly to the dynamic segments.

In order to eliminate any additional violations of the temporal firewall principle, non-urgent requests or conditions from the environment are not modeled as events in our framework. Instead, they are sent from an object's environment to the object through ports and they are handled in a time-triggered manner, i.e., they take effect

only at times specified by the users. Whether these requests or conditions are buffered or overwritten is defined by users in the functions implementing the interfaces of the corresponding ports.

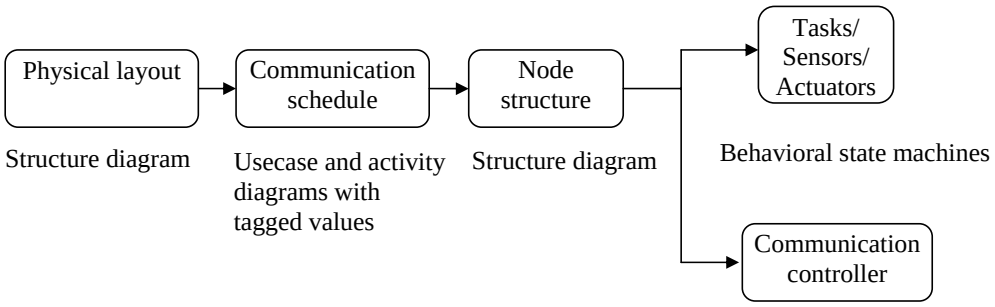

Figure 4-7: The UML-based design flow for TTAs

The Design Framework Fig. 4-7 summarizes our proposed design flow of a time-triggered application using UML 2.0 notations.

For convenience, we add some simple notational extensions – as is allowed by the UML 2.0 standard – through stereotypes and tagged values. The stereotypes of clusters, nodes, and communication controllers are applied to classes as well as to instances of these classes. In addition, tagged values in the activity diagram capture timing details of the physical communication platform. These extensions not only aid the modeling effort, they also contribute to the process of generating SystemC code. They can also be exploited during the implementation stage.

The communication schedule is determined before each task on the nodes are modeled, so the nodes can be developed independently. Our design framework supports composability in the sense that a new application can be smoothly added to an existing model. When a designer wants to extend a cluster with a new application, the following steps must be performed. First we remark that if the new application is to be run on an existing node, then this will basically boil down to a fresh local schedule for that node followed by suitable modifications to the (hierarchical) behavioral state machine describing the execution of tasks on this node. Hence below, we outline the

steps needed to add a new node on which the new application will run.

- modify the structure diagram of the cluster class to include a new ECU;
- modify the activity diagram which describes the bus communication schedule;
- model the new node as well as the tasks running on this node for the new application using structure diagrams and a behavioral state machine;
- if necessary, modify the behavioral state machines of the tasks of the existing applications if their schedules are affected by the new changes. This modification is restricted only to the time parameter of the time-out transitions. However, timing constraints must be checked by the designer to make sure that there is enough time for a task to complete the actions that it is supposed to execute when in a particular state. This can be done through worst case execution time analysis. However, we do not focus on this in the thesis.

The above modifications are well localized and users can easily identify the places in the model which have to be changed.

Finally, a SystemC model of the communication controllers (CCs) is needed to simulate the system model translated from the UML layer. Users can choose to supply this SystemC model and compose it with the application's generated SystemC code. Alternatively, they can model the communication controllers at UML level using structure diagrams and behavioral state machines (as in Fig. 4-8) and then use our tool to generate SystemC code for the communication controller automatically. The later approach is faster and easier. In fact we have taken this approach to derive an abstract model of the FlexRay communication controllers at the UML level. This model can be included in a library of UML-level communication controllers for application developers who want to develop various applications running on time-triggered protocols. The library can include the communication controllers for different protocols such as FlexRay, TTP [78] and TT-CAN [52]. Each compo-

nent in the library should be validated before being put into the library. Fig. 4-8 summarize the alternatives presented here.

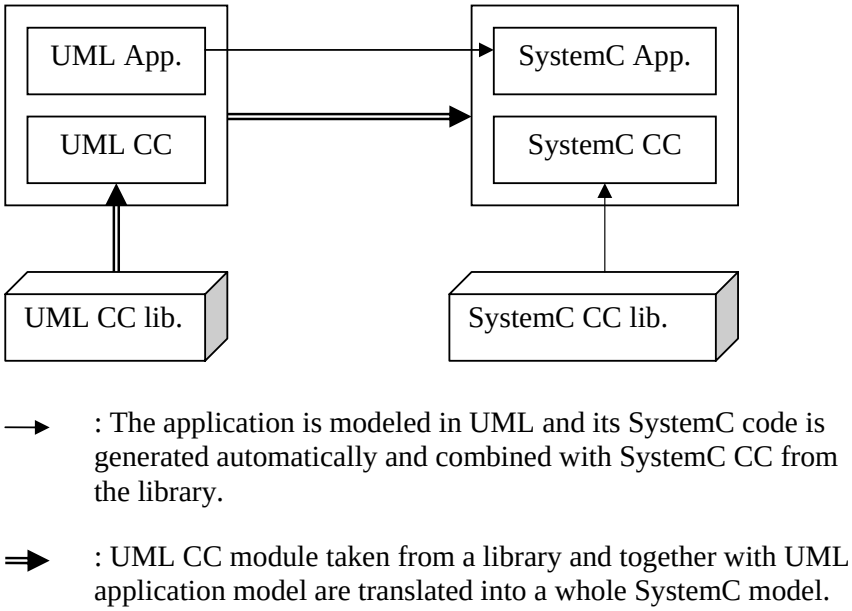

Figure 4-8: The communication controller library

4.5 SystemC code generation

We have constructed a translator that automatically converts the UML model of a system into executable SystemC code. The translation process is not routine and we outline here the main steps as well as the implementation choices we have made in order to obtain fast simulation.

A UML class is translated into a SystemC module. The objects inside a structure diagram of a class will become sub-modules. Initialization codes that create modules and connect them together are generated automatically from the structure diagrams of the structured classes. The initialization codes are then inserted into the respective structured classes' constructors.

Communication through ports defined by interfaces in the UML model is sup-

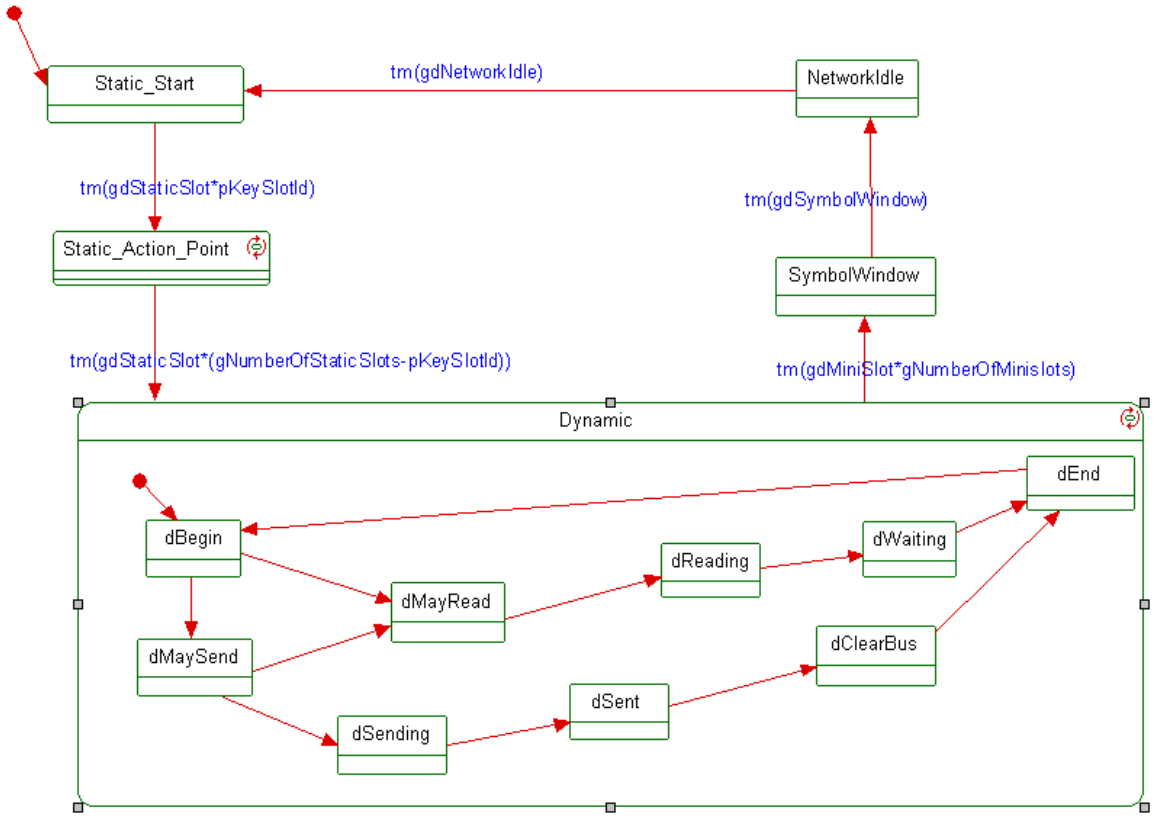

Figure 4-9: The state machine of the message transmitting component in the FlexRay communication controller

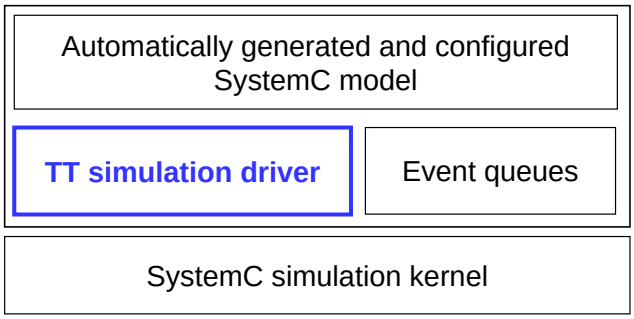

Figure 4-10: The simulation layers for time-triggered applications

ported naturally in SystemC by the concepts of `sc_port` for ports with required interfaces, and `sc_export` for ports with provided interfaces.

The number of ECUs per cluster in a TTA can be large. Our translator automatically configures the SystemC communication controllers by extracting the relevant information from the activity diagram we highlighted in the previous section (see Fig. 4-4). This process utilizes our UML stereotypes to identify and pass arguments to the nodes, and subsequently to the communication controllers.

The key to fast simulation is how the code for execution of behavioral state machines is executed. The OSCI SystemC simulation kernel is a generic discrete event simulator. It does not take advantage of the crucial property of TTAs, namely, the communication schedule is pre-defined. Thus, if we generate SystemC in the usual way by mapping the behavioral state machine of each class to a SystemC thread, we will get poor simulation speeds. Instead, our translator consolidates a schedule table for a whole cluster. A module called *simulation driver* will execute the model according to this schedule table. Since each time-triggered composite state in the state machines has the form of a cycle, the schedule table's length is finite and can be determined by our translator. In effect, we will have just one SystemC thread residing in the simulation driver to execute all time-triggered actions. To repeat, the simulation driver is generated automatically by our translator. Fig. 4-11 shows the block diagram of the generated SystemC code from the model in Fig. 4-5 in which

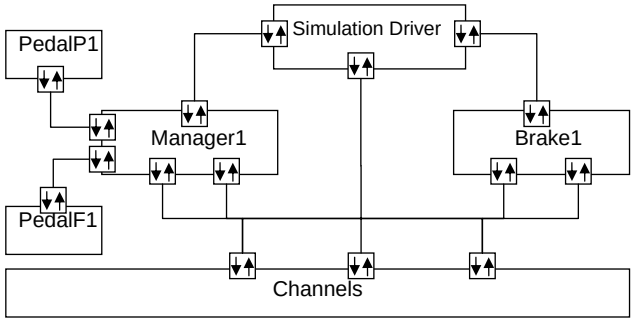

Figure 4-11: The block diagram for the SystemC generated code

we have eliminated `Manager2` and the three Brakes to improve the clarity of the figure. The simulation driver is connected to each node (and the task on each node) to drive the simulation. The driver is not connected to the two pedal sensors since they are passive modules (there are no state machines associated with them); They just provide functions that the `Manager`s can call.

Following is a segment of the code for the simulation driver:

```
while(true){
    schedule_0();
    wait(200, SC_MS);
    schedule_200();
    wait(300, SC_MS);
    schedule_500();
    . . .
}
```

Each function `schedule_x()` includes calls through ports which execute the actions supposed to be run at time `x`.

```
void BBWCluster_SimulationController::schedule_200(){
    pSC_manager_Manager1_car->Activate(11);
    pSC_manager_Manager2_car->Activate(11);
```

```
    pSC_manager_Manager1_car->Activate(10);
    pSC_theCruiseController_MyCruiseController_car->Activate(47);
}
```

The parameter of the Activate() function represents the state from which the object being called should leave, take action of the transition to leave this state and move to another state. Following is a sample of code for BrakeManager class which has 2 objects called above (`manager_Manager1_car` and `manager_Manager2_car`)

```
void BrakeManager::Activate(int state){
    switch(state){
        case 9: // BrakeManager_TOPSTATE_initial
            brakeStatus[0] = 1;
            brakeStatus[1] = 1;
            brakeStatus[2] = 1;
            brakeStatus[3] = 1;
            Sensing();
            OUT_PORT(pCC)->transmitChannelA(pedalPosition);
            break;
        case 10: // S1
            otherPedalPosition = OUT_PORT(pCC)->receiveChannelA();
            break;
        case 11: // state_14
            . . .
    }
}
```

Although all the time-triggered actions are executed by only one thread in the simulation driver, additional SystemC threads are unavoidable due to the states corresponding to dynamic segments. Interestingly, we can switch off these threads for

the ECUs that don't send or receive during dynamic segment. This information is available via the tagged values of the `Dynamic` activity in the activity diagram.

Our translator can insert codes to print out traces, including state transitions, event notifications and their occurring time.

Implementation

The implementation for the translation from UML to SystemC is as the same as described in section 3.3. The differences are in the pre-processing procedure and Velocity templates. The pre-processor does the followings:

- From the activity diagram, determine the parameters needed to be passed to all the nodes in order to configure theirs communication controllers;
- Compose the whole model's schedule table based on all the state machines. This includes determining the values of all the variables mentioned in the time-out statement of the transitions. These variables must be initiated before the simulation starts, meaning in the constructor of the modules or passed through the constructor's parameter from the father module;
- Add a simulation driver to the model;
- Connect the simulation driver to its controlled modules.

The XML parsing and pre-processing procedures described above can detect some modeling errors such as:

- There are not enough information (tagged values) in the activity diagram.
- Some state machine doesn't have a cycle form, or its cycle's length in term of time is not a multiple of the length of a FlexRay cycle.
- Some variable mentioned in a time-out statement in some state machine doesn't have any value initialized (which means the time at which an action of this object is triggered is not determined).

4.6 Experimental results

We have experimented our approach with a number of metrics in mind. The most important one is simulation speeds as a function of the number of FlexRay cycles. We also compare our simulation speeds with the simpler thread-based approach in which the simulation driver is not synthesized and instead one SystemC thread per behavioral state machine is created as implemented for event-triggered applications (in section 3.3). We also observe the amount of time it took to add new ECUs of a new application to an existing cluster and the number of lines of code generated by our translator as a (very) rough estimate of the effort saved to create an executable SystemC model.

4.6.1 A Brake-by-Wire (BBW) application

We have created at the UML level an automotive brake-by-wire (BBW) application running on FlexRay. The communication components were taken from our framework's library (see Fig. 4-8). The BBW application was developed based on the material in [70] and [28].

Fig. 4-5 shows the structure diagram for a cluster. There are six nodes in the system, one for each wheel (brake node) and two manager nodes. Each brake node controls a brake. The manager nodes obtain the force and position applied to the brake pedal from sensors, calculate the brake force that each brake node should apply and send it via the bus to the brake nodes. In turn, the brake nodes send their current status to the managers also via the bus. All these communications are done in the static segment. The dynamic segment is not used here. The FlexRay communication subsystem at each node places data on the bus at the scheduled slots and reads data from the bus when the application needs to.

1,333 lines of SystemC code were generated. At the UML level, we could not gain access to the code corresponding to the algorithms for computing the brake force etc.

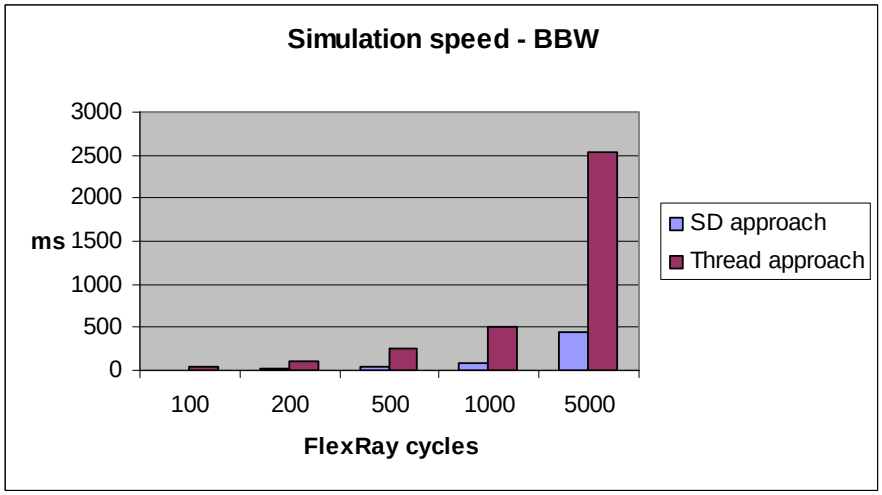

Figure 4-12: Simulation speed of the BBW application

Instead, we inserted our own simplified code to mimic the functionalities. (This is the also the case for the adaptive cruise controller to be described later.) We simulated the generated SystemC code for varying numbers of FlexRay communication cycles on a PC with a 3 GHz Pentium 4 CPU and 1 Gbytes of RAM. The simulation times, in terms of milliseconds, are shown in Fig. 4-12. The figure shows the simulation times for the code generated using the thread-based approach. As can be seen and expected, there is a significant gain in simulation speeds when we synthesize a simulation driver which can leverage on the time-triggered nature of the static segments.

We also determined that simulation times obtained via the simulation driver approach are almost the same as the times obtained by hand-creating a SystemC model for the application. Since this is an ad hoc approach, we have not shown this comparison here.

4.6.2 An adaptive cruise control (ACC) application

To check the extent to which composability is supported, we added an adaptive cruise control application to the BBW cluster.

In this application, once a driver presses the `Set` button, the cruise controller will

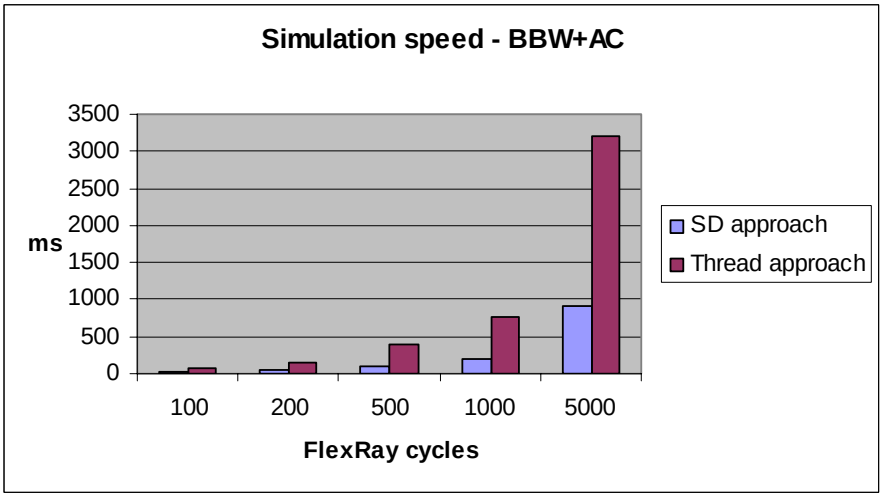

Figure 4-13: Simulation speed of the BBW and ACC applications

maintain the car's speed at a setting provided by the driver. It will also maintain a safe distance from the vehicle ahead by adjusting the throttle using the car's current position, speed and its distance from the vehicle ahead. The cruise control will be disengaged whenever the driver steps on the brake pedal or presses the `Off` button. In addition, there is a `Resume` button that allows the car to resume cruise control if it has been disengaged. The `Set`, `Off` and `Resume` buttons (i.e. the corresponding sensors) reside on the same node as the cruise controller.

In our experiment, to implement this new application, two more nodes were added to the BBW cluster; one for the cruise controller and the other for the throttle. This resulted in two more static slots in the static segment of the communication round. One slot is used by the throttle to send the current position of the throttle and the second slot is used by the cruise controller to send a new position that the throttle is instructed to attain. The ACC uses the dynamic segment to handle the emergency situation when there is a vehicle in front of the car at an unsafe distance.

It took the author around three hours to have the cruise controller application added to the existing BBW cluster, generate the SystemC code for the extended model and debug the new model to get the new application to work properly. The

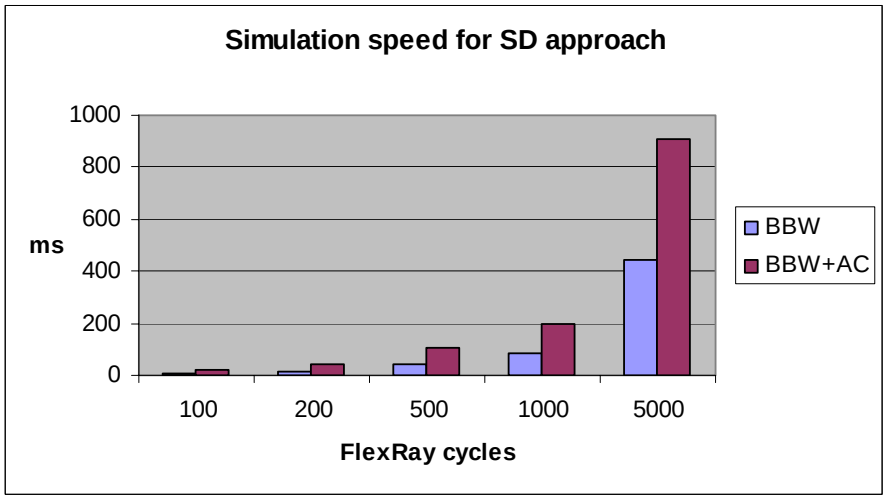

Figure 4-14: Simulation speed of the simulation driver approach

new application could be added quickly because our modeling method, as detailed in the previous section, makes it easy to identify the places where changes have to be made. Further, the SystemC configuration of communication controller is generated automatically. This saves a significant amount of time especially in a system with many nodes. The generated SystemC code is 4,314 lines in total.

Fig. 4-13 shows the simulation speeds of the combined application in milliseconds for varying numbers of communication cycles. The dynamic segment is used in this application. Hence the `OR` states corresponding to dynamic segment are mapped onto SystemC threads. So not only are we simulating more nodes compared to the BBW cluster running alone, there are 13 additional SystemC threads due to the use of the dynamic segment (as opposed to only 1 thread in the first case). Thus there is a noticeable increase in the simulation times. Admittedly, the new threads are not computationally intensive but as is often the case with SystemC simulations, the speed penalty incurred is due to the context switching that is required in the presence of multiple threads (see in Fig. 4-14).

There are also some slight differences between the simulation speeds of our simulation driver approach and the hand-written code. The simulation driver approach

however still gains over the thread-based approach.

Apart from composability and simulation speed aspects, this experimentation showed the re-usability in our framework. Components designed in the previous experimentation (BBW) were re-used conveniently in this experimentation.

4.7 Summary

We have described the design of time-triggered applications by constructing a UML-based design framework. We expose the relevant features of the underlying architecture and time-triggered protocol at the UML-level through a suitable choice UML diagram types and fixing their roles. In particular, structure diagrams are used to display the underlying TTA architecture, restricted behavioral state machines to capture the control flow of application tasks guided by the communication cycles of FlexRay, and an annotated activity diagram to display the main features of the static communication schedule. In addition, our framework enables fast prototyping of time-triggered applications and early design validation. We allow applications to be developed at a more abstract level before full implementation.

To support preliminary functional validation, we have constructed a translator by which SystemC code can be automatically generated from UML designs. In this aspect, our contributions include the novel use of a simulation driver which uses a single thread to drive the simulation at the SystemC level to significantly improve simulation speeds and its automatic synthesis by our translator. We believe this technique will be applicable in other settings as well; in particular, when there is a system-level static schedule involved.

Our framework also supports key design principles of TTAs, such as temporal firewalls through the restrictions imposed on the usage of ports with proper interfaces and composability by localizing the changes to be made at the UML-level in order to incorporate a new application.

We have experimented with two different configurations (static segment only, and static followed by dynamic segment)using two standard applications. Due to the XML-based intermediate representation, the current framework can be easily connected to other tools for formal verification [105, 107] and low level design tools such as schedulability analysis [99] and synthesis [98]. Comparison/combination with analytical frameworks such as one described in [67] is also a potential future work. In addition, worst case execution time (WCET) analysis is necessary for schedulability analysis and timing annotation in our framework.

Today, premium cars feature not less than 70 ECUs connected by more than five different bus systems [27]. These systems may be running on different protocols, namely time-triggered protocols such as FlexRay, TTP and TT-CAN and event-triggered protocols such as CAN [24], LIN [5] and MOST [6]. Thus, our future work includes exploring other time-triggered protocols such as TTP [78] and TT-CAN [52] and include their models in the library of communication platforms for application developers. Exposing some services provided by the communication platforms such as the startup and membership service provided by TT-CAN and TTP is also desirable.

Here, we have focused on systems consisting of a single cluster. However it will not be difficult to extend our framework to handle multiple clusters connected through gateways. The clusters may be based on different communication protocols which include both event-triggered and time-triggered ones. Moreover, it will be important to investigate issues such as the case when more than one task are allocated to an ECU; and modeling of time-triggered operating systems which comply to the standard for time-triggered operating systems OSEKtime [65].

In a larger context, one needs backward association mechanisms through which faulty runs (including application's communication's error) obtained at the SystemC level can be traced back to the ill-behaved parts of the UML-level model. The construction of such a mechanism will enable the creation of test benches at the UML level and their verification in SystemC. We will target this issue in the next chapter.

Chapter 5

Design validation

As described in the previous chapters, our proposed framework can model reactive and time-triggered embedded systems at UML level, generate SystemC code automatically and perform simulations for validation. However, because the systems being modeled can be complicated especially in terms of the interactions among the components, it is difficult for designers to discover failures during simulation. This chapter sketches how our design framework can be augmented to facilitate designers to test their design and to increase the amount of confidence in the correctness of the designed systems. In particular, we show how to model test cases at UML level and generate the test driver automatically for SystemC simulation. In addition, the means for establishing model association is done by which the simulation trace can be reflected back to the model level. The validation framework proposed here is meant for both reactive and time-triggered systems. However, our initial effort is oriented towards time-triggered applications because of the strong demand for high level validation tools for this kind of applications. Further, we expect that the test cases to be generated and modeled manually.

We shall first review the related works. Next we will show how usage and expected behavioral scenarios can be modeled in UML. Then the details of how SystemC Test Driver is generated for simulation and model association will be presented. Finally,

the case studies will show the usage of our framework to validate the functionality, robustness and performance of a system.

5.1 Background

A *failure* [120] is an undesired behavior. Failures are typically observed during the execution of the system being tested. A *fault* is the *cause* of the failure. Once we have observed a failure, we can try to find the fault that caused it and correct that fault. So *testing* is the activity of executing a system in order to detect faults.

The tests may be categorized by the kind of information we use to design them. In *Black-box testing* we do not use the information about a system's internal structure to create the test. In other words, the system under test is treated as a black-box. On the other hand, in *white-box testing* the implementation code is used as the basis for designing tests.

Model-based testing is the automation of black-box test design, in which tests are generated automatically from models that describe the behavior of the system [69]. A model-based testing tool uses various test generation algorithms and strategies to generate tests from a behavioral model of the system-under-test (SUT) [120].

In our framework, the model is created at the UML level and the simulation is carried out at the SystemC level. Model association is the process by which the simulated SystemC trace is translated back to the UML level and displayed in terms of the model.

The traces displayed at modeling level can help designers to validate both functionality and performance. More importantly, they will help designers identify bugs in the models and thus, fix them early in the design process. With the increasing complexity of embedded systems, tools that support the above activities would be of much help in the high level design process and they will make our design framework more substantial.

The LEIRIOS LTG/UML [120] is a model-based testing tool in which use case diagrams are used to capture possible usage of the SUT. The abstract model of the system consists of class diagrams and behavioral state machines. Details unrelated to the functionalities under test are ignored. For example, some classes in the model for design are replaced by simple enumerations of test values. The LEIRIOS LTG/UML tool generates a test suite in terms of test scripts.

There are also a number of similar tools [20] which take the UML diagrams as input to generate test suite. CowSuit also uses use cases, sequence diagrams for describing possible usage of the system. The use cases and sequence diagrams are weighted such that the tool can generate tests differently for different functionalities based on their levels of importance.

Apart from CowSuite, [122] uses activity diagram to model an operation to be tested, from which test cases are generated.

Although there are a number of tools generating test suite automatically, they do not support the execution of the generated test cases and the means for using the test results. In our opinion, designers need more support in terms of test suite execution and the displaying of the test results at the model level. In particular, one must be able to highlight the differences between the simulation trace and the expected behavior at the model level.

5.2 UML pattern for validation purpose

As mentioned above, it is necessary to capture test cases and display simulation traces at the model level to facilitate the design validation. We have chosen UML sequence diagrams (section 2.3) for this purpose. They show the interaction among the components in a way that easy for designers to comprehend the message exchanges and the ordering of messages as well as the data being exchanged. Communication and sequence diagrams are semantically equivalent. On the other hand, it is more

difficult to visualize message ordering in communication diagrams.

We separate the sequence diagrams for the usage scenarios and the ones for the expected scenarios. A usage scenario shows how the system's environment or user will interact with the system. It shows how the system's user sends requests or data to the system's components. An expected scenario shows how the internal components of a system are expected to behave after receiving the requests or data from the user. It can also show the output or response of the components to the environment. The reason for this separation is that the usage and expected scenarios serve different purposes in our framework. The usage scenario is used to facilitate the execution of the test. On the other hand, the expected scenario is used to check the behavior of the designed system. Another reason is that often the objects (components) involved in these two types of diagrams are different and if we combine them together, there would be many life lines and the diagram would be very difficult to read.

5.2.1 UML modeling for usage scenarios

Our UML model of the system under design includes the components at the boundary of the system such as sensors from which inputs are to be received. The system under test can receive input data to be processed by the system or requests from the environment. It can also take note the time at which the data/request is sent to the system.

The components at the boundary of the system provide input either passively or actively to other internal components. They do so passively when the internal component(s) retrieves input from the boundary components. In this case, the internal components decide at which points of time they retrieve the input, hence the environment only supplies data to the internal component. This case usually occurs in time-triggered applications when data are processed in a time-triggered manner. They do so actively when the environment itself decides at which point of time to send data to or to trigger the functions of the internal component(s). In this case,

the point at which this happens is important to the system and may be taken as data to be processed as well.

We recall that in chapter 4, an automotive system consisting of brake-by-wire and adaptive cruise controller applications was used as a case study. An example of usage scenarios for such a system would show how the driver of the car will drive or interact with that system, e.g. the force that the driver applies on the brake which is sensed by the brake's sensor or when she/he sets/stops the automatic cruise by pressing the corresponding buttons. We call the objects such as the brake sensors and the buttons *boundary objects*. They interact with the environment and relay the inputs to other internal objects.

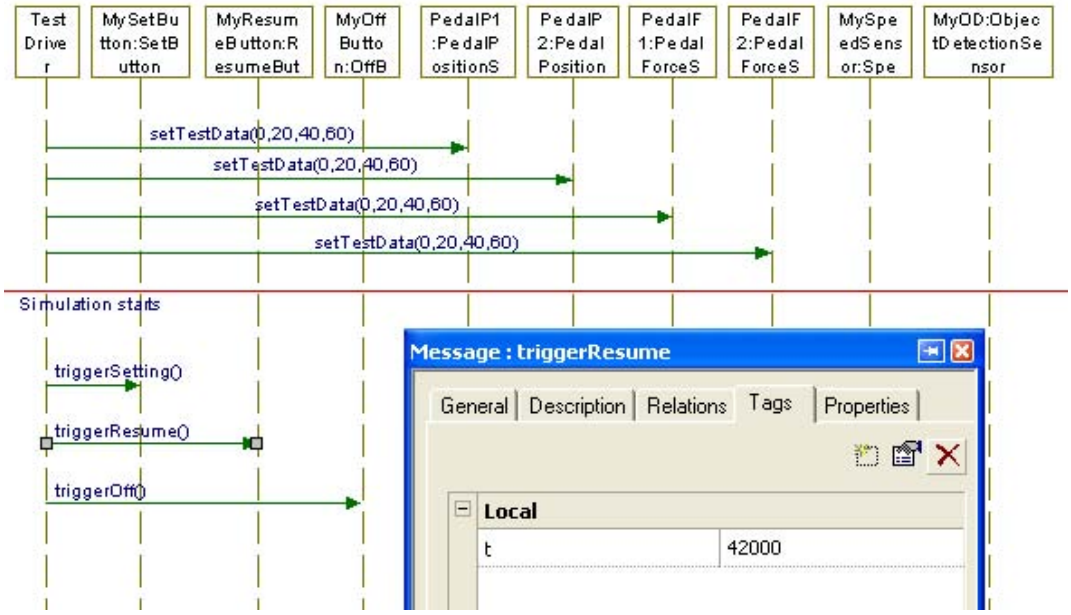

Figure 5-1: A BBW usage scenario in Rhapsody

Fig. 5-1 shows the usage scenario for the BBW application, which is modeled as a sequence diagram in the Rhapsody software [100]. The vertical lines (called life lines in UML) represent the objects (components) in the system, while the horizontal arrows represent the messages being sent from the object at the tail of the arrow to the other end. There is a special life line called Test Driver in the sequence diagram,

which represents the car driver in this case study. In general, the Test Driver is the environment in which the system is operating. It can be some outside components or the system's users. The Test Driver does not exist in the system's UML model discussed in the previous chapters. It only appears in the sequence diagram for usage scenarios. The other life lines in the sequence diagram represent the boundary objects.

There are two types of messages from the Test Driver. The first type corresponds to the case when the boundary objects provide data to other internal components passively. It consists of messages that set the input data for the system. In our example, it consists of the values that the brake or cruise controller will read from the sensors. The data is retrieved from the boundary objects and processed by the controllers in a time-triggered manner. Hence they can be set in advance, before the simulation starts. For example, the first message *setTestData(0,20,40,60)* is a message or function call from the Test Driver to *PedalP1* sensor.The sensor provides this function for the Test Driver to call in order to set its sensor data. The second type of messages represent the triggers or requests from the environment (the car driver in this case) to the system. The point of time at which they occur are important, hence these messages have a tag for each, which tells the point of time the trigger/command occurs. For example, the message *triggerSetting* represents the action of the driver pushing the Set button to set the car to automatic cruise mode.

The above usage scenario modeling is expected to be done manually by designers. At present, our framework will take it as an input to generate a SystemC Test Driver module.

5.2.2 UML modeling for expected scenarios

The sequence diagrams for expected scenarios specify how the components in the system should behave or interact with each other or with the environment when the corresponding test (described in the sequence diagram for usage scenario) is carried out. They are similar to those for usage scenarios, except that the diagrams for

expected behaviors do not have the messages which appear before simulation starts. Fig. 5-2 shows such a diagram. The specified behavior will be compared with the trace generated while doing simulation in order to help designers identify design faults.

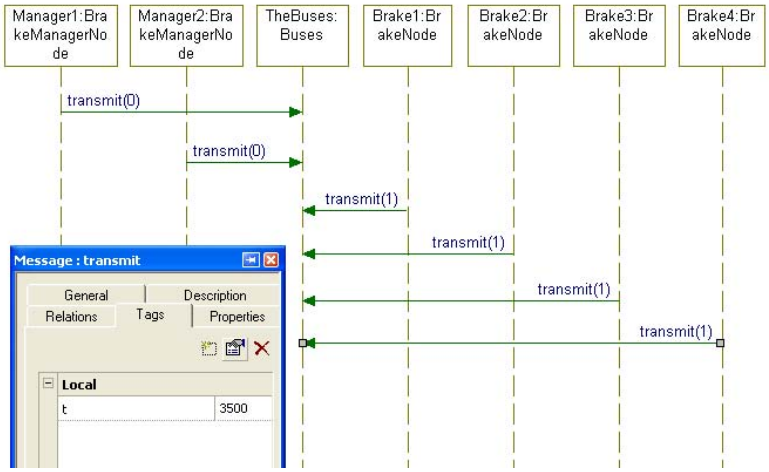

Figure 5-2: An expected BBW scenario in Rhapsody

5.3 SystemC Test Driver generation

As mentioned above, the Test Driver represents a system's environment. Hence, the Test Driver is not a part of the UML model of the system. At UML level, it only presents in the sequence diagram(s) displaying the usage scenario(s). However, it must exist as a component in the SystemC simulation so that it can drive the boundary objects according to the specified usage scenario(s).

In our framework the SystemC module for the Test Driver is generated and connected to the SystemC modules of the boundary objects. The code generation process is similar to that for the model, namely the Rhapsody internal representation for each sequence diagram is converted into an XML file which is then parsed by the translator to get information about the behavior of the Test Driver and its connections with other objects. These connections, as mentioned above are not specified in any structure or class diagrams. Instead they are realized in SystemC by adding ports and

links into the abstract syntax tree. The Test Driver will become a SystemC module which sets the input data before the simulation starts and then triggers the connected modules by calling their functions through the newly created ports according to the specified sequence diagram.

The advantage of the Test Driver approach is that designers can change their test easily by creating a new sequence diagram and generating a new SystemC Test Driver, without having to generate code from the system's model again.

5.4 Model association

While simulating a complex design, the SystemC run involves many objects executing concurrently. Debugging such a big model is difficult at the SystemC level. Moreover, designers would like to analyze simulation results so that they can answer some questions about the design at the model level. For example, in the context of time-triggered applications, typical questions are:

- Does the chosen schedule meet the required deadline?
- Does the number of static slots in static segment and minislots in dynamic segment suffice?
- Is an event-triggered message never sent because it has low priority or the dynamic segment is too short?
- Does the system produce correct results or display an expected behavior within a specified period of time?

The following actions/events occurring during a simulation run of a system are important in answering the above questions. Firstly, the messages passed among components in a system or from the system to its environment and their order. Secondly, the point of time at which the messages occur. Finally, the data being sent along with the messages.

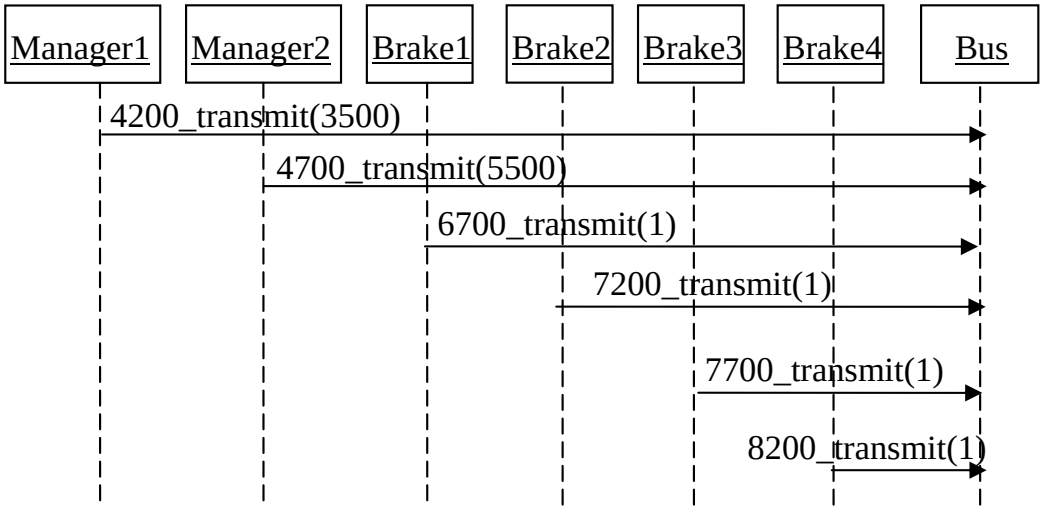

Figure 5-3: A trace sequence diagram

Our translator inserts code to record the above information. In addition, the designers can insert their own code to print out the information they want. This information is then reflected back at UML level as sequence diagrams. This way of displaying a SystemC run back at UML level can help designers validate and debug their design.

The sequence diagram resulting from the simulation trace is filtered so that only the objects specified in the sequence diagram of the expected scenario are depicted. Fig. 5-3 shows an example of the trace sequence diagram. It is displayed by the tool UMLGraph [12]. The information about the messages are displayed explicitly as the messages' labels.

Our framework includes a program that reads the SystemC simulation trace and builds an abstract sequence diagram. Subsequently, a UMLGraph textual description of the filter diagram is generated so that an image of the sequence diagram is built by UMLGraph.

Sequence diagram comparator Although the trace sequence diagram is filtered as mentioned above, the sequence diagram may still be large for designers to check

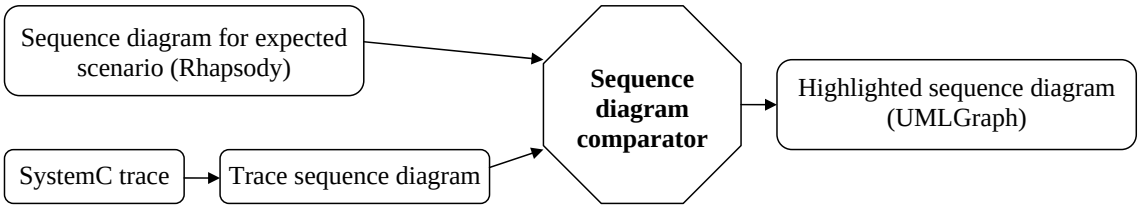

Figure 5-4: The sequence diagram comparator

and compare with the requirements. Our framework includes a sequence diagram comparator. The trace sequence diagram is compared with the sequence diagram for expected scenario so that the differences are highlighted. Fig. 5-6 shows an example of the highlighted sequence diagram displayed by UMLGraph [12]. In order for designers to easily analyze the results, the test scenario specified by the designer in Rhapsody is also converted to UMLGraph format. The highlighted diagram can give the designers some hints to go back to the model and modify the fault(s) if there is any failure.

Fig. 5-5 summarizes our validation approach.

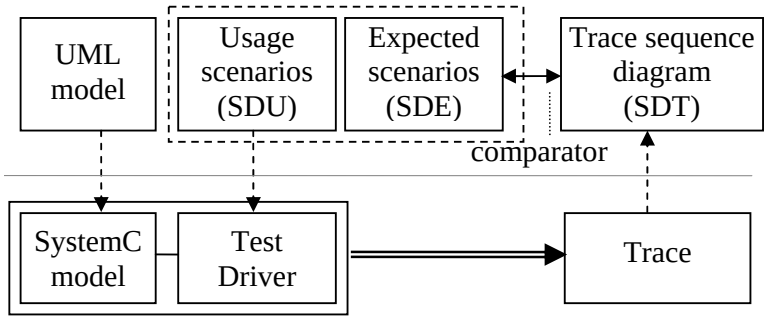

Figure 5-5: The validation framework

5.5 Experimental results

5.5.1 Brake-by-Wire (BBW)

The BBW model was presented in section 3.4. In this application, the BBW managers compute the brake force to be applied at each wheel and send this data to the bus.

Suppose there is a problem in the computation of one of the managers, our tool will highlight the first occurrence of the error (message *4700_transmit(5500)_???* in Fig. 5-6). In this case, the datum being sent at time 4700 is not as expected. Since usually the data being sent on the bus are computed just before the node's assigned slot, the designer can go back to the UML model and check the computation at that point of time.

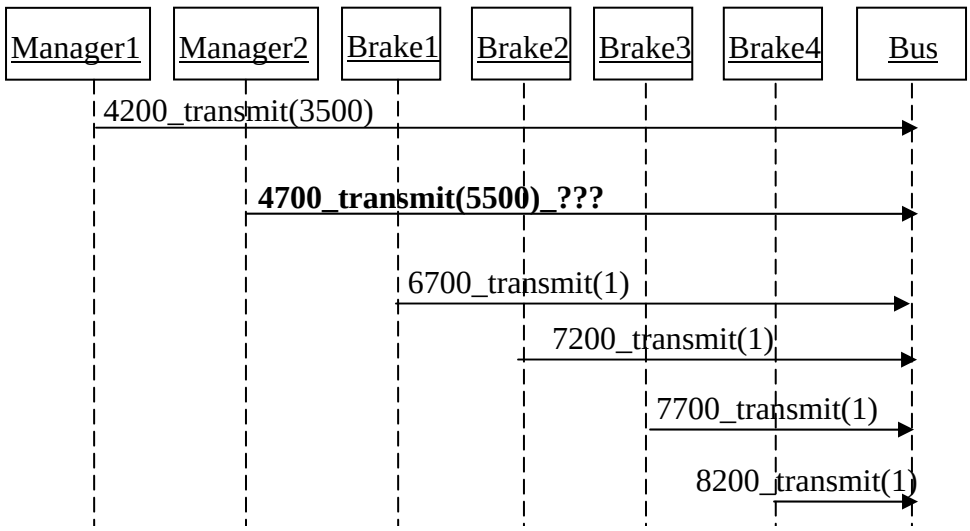

Figure 5-6: The highlighted trace for BBW application in case there is some computational error

5.5.2 Soft state protocol

This case illustrates the use of our method to detect errors such as deadlock, thus realizing the missing features in their communication protocol. Soft state protocols are used in packet-switching networks to manage the nodes's information called states which are refreshed by periodical messages. Otherwise these state will expire. Here we can consider states to be variables that we want to have replicated and up-to-date version maintained on other nodes in the network. We used a typical state management protocol [56] as a case study.

This protocol involves three entities: the Initiator, the Forwarder and the Tar-

get. The Initiator installs, refreshes and removes the state on the Target by sending messages over a lossy channel through the Forwarder, which may be a router in a packet-switching network where the Initiator and the Target reside. This protocol needed some adjustment when implemented on a time-triggered platform. Due to the bus architecture, the Forwarders are not necessary. Instead, the Initiator is allocated a slot for it to send the request (either setup, refreshing or removing) and each of the Targets is assigned one slot for it to send the acknowledgement back. Without loss of generality, we modeled the soft state protocol with one Initiator and one Target. We modeled the protocol on the FlexRay communication platform provided as a library in our framework. The library has significantly eased and accelerated the task of modeling the protocol.

A typical execution of the protocol includes 3 phases:

- Setup phase: when the Initiator receives an ASetup request with state information data from its environment, it installs the state locally and sends a *trigger* message to the bus. The Target issues a *notify* message back to the Initiator through the bus to indicate that the setup has been done successfully. If the Initiator does not receive the *notify* message after a pre-defined period of time, it will transmit the *trigger* message again.

- State maintenance phase: after a *refreshing* period of time, the Initiator periodically sends a *refresh* message. If no *Refresh* is received by the Target after a state expiration period of time, the state will be removed.

- Teardown phase: when the Initiator receives an ATeardown request from its environment, it sends a *remove* message toward the Target to remove the state. The Target upon receiving this message will remove its state and send a *notify* back to the Initiator. If no *notify* is received within a given period of time, the Initiator will retransmit the *remove* message .

Possible problems in a state management protocol include failures to install or

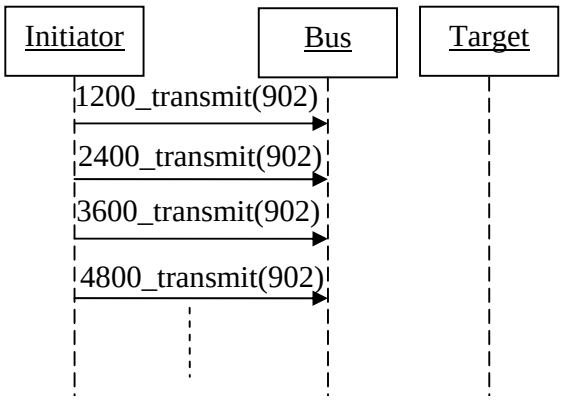

Figure 5-7: The trace for the case of no re-trial counter

remove a state correctly. Moreover, the defined periods for retransmission, refreshing and state expiration must be taken into account.

In order to check how our framework can help in detecting design faults, we purposely eliminated the messages from the Initiator to the Target, simulating the case when the Initiator's outgoing link or the Target's incoming link is broken. The Initiator does not receive the desired *notify* message after sending out a *trigger* message. Hence it sends the *trigger* message again. At first when the maxRetransmission variable which determined the maximum number of times the Initiator could try to send the *trigger* message was not defined, the Initiator kept sending the *trigger* message indefinitely (as shown in Fig. 5-7). Meanwhile the Target was still waiting for a message from the Forwarder. This should prompt the designer to add a retrial counter to limit the number of retransmission for the *trigger* message. The same should be applied for the *remove* message in the teardown phase.

5.5.3 Membership service

We modeled a membership service running on FlexRay [21]. The membership service is to ensure that nodes in a cluster have a consistent view of the status of the nodes in a system. All the nodes will start with their local opinion that all the nodes are OK. The steps of the membership service in each FlexRay communication cycle consists of

each node sends its status to the bus in its assigned slot during the static segment. All the nodes observe the status information sent (OK or silence). During the dynamic segment, if a node detects that the observed status of any node is different from its current local opinion about that node, it will broadcast its observation to all other nodes. The dynamic segment is configured to be long enough for all the nodes to broadcast. During the network idle time segment, each node will run the decision function locally to decide its view of the system. The decision function used in this protocol is "strict majority", meaning at least 50% of the nodes must agree on the status (alive or dead) of a node. If the final decision is different than the local opinion of a node or does not contain the node as a member, the node will halt. In other words, a node is removed from the membership set if its local opinion is different from the final decision.

A requirement for the membership protocol is that it must be able to detect and exclude any faulty node within two FlexRay cycles. There are three possible failures. A node failure occurs when the node is unable to send out its status information during the static segment or its local opinion during the dynamic segment. An outgoing link failure (OLF) occurs when the outgoing link of a node fails. In this case it may be able to receive messages but it is unable to send messages to the network. An incoming link failure (ILF) on the contrary of the above.

On a node, the above failures may occur before a static segment (the assigned slot of the node in concern) and after the dynamic segment of the previous cycle; or after the static segment (the assigned slot of the node in concern) and before the dynamic segment. Different possible failures, faulty nodes and occurring times result in many possible scenarios in the test suite. Automatic test suite generators such as the one proposed in [59] are specially useful in this case.

We have done experimentation with different faulty situations for 4 nodes.

- A node fails before sending its status during the static segment. However it can still receive messages and send its own local opinion during the dynamic

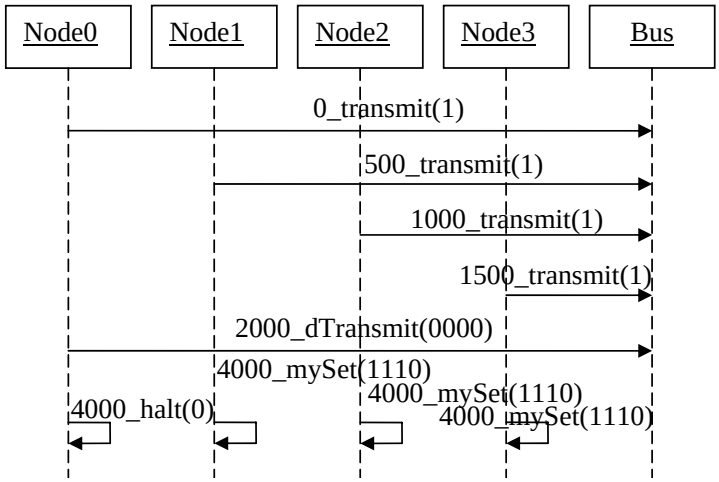

Figure 5-8: The trace for the incoming link failure case

segment. The simulation trace in Fig. 5-9 shows that all other members were able to come to a consensus within one FlexRay cycle.

- A node's incoming link fails before all nodes sending their status. Hence the faulty node did not receive any status messages, so in the dynamic segment it sent out its local opinion which is different than all others. Hence it is halted in the current cycle (Fig. 5-8).

- A node's outgoing link fails before it sends it status during the static segment. Like above, the consensus was able to be reached within one FlexRay cycle.

- One node fails and another node's outgoing link fails. The consensus was reached within one cycle.

These failures are simulated by having the Test Driver setting the faulty nodes' status to 0 for the first three cases and modifying the values of the messages received by the faulty node in the last case.

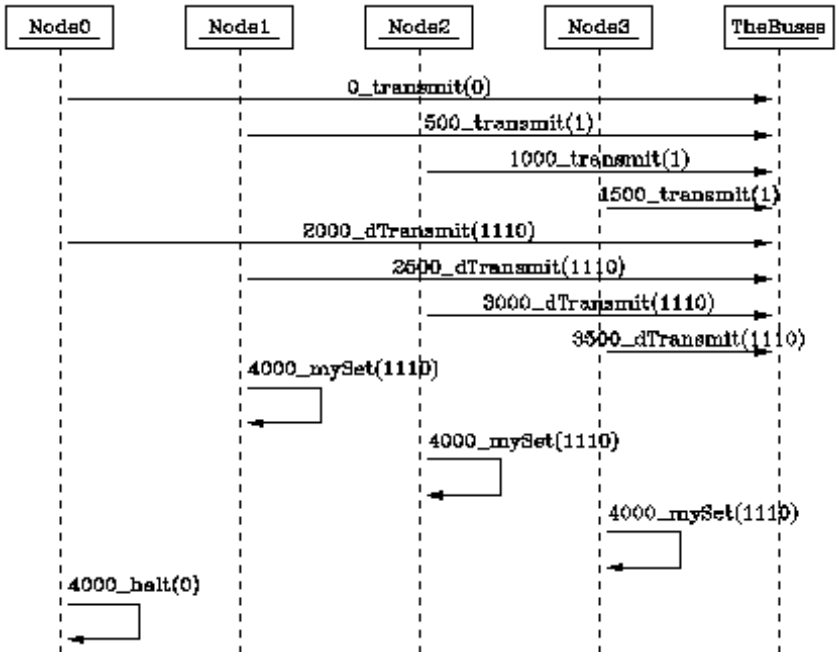

Figure 5-9: The trace for the node failure case

5.6 Summary

We have presented here how testing is supported in our framework. We use UML sequence diagrams for the modeling of usage and expected scenarios. From the UML sequence diagram for the usage scenario our framework produces at SystemC level a Test Driver, which drives the simulation as specified in order to carry out the test. This automatic generation saves designers much effort and time because the generator does all the connection between the Test Driver and the components that have interaction with the system's environment and fills in the triggering code of the Test Driver. In addition, the SystemC simulation results and traces are reflected back also as UML sequence diagrams to facilitate designers to find faults in their design if there is a failure. The resulting sequence diagram and the specified expected scenario's sequence diagram are compared and the differences are shown by a comparator.

Through the case studies we have illustrated the applicability of our framework to validate different aspects of a design. In particular, the functionality of an application

can be validated to make sure that its output is correct. In addition, the robustness and performance of an application under different conditions such as network failures can be tested.

Future work Although our framework does not include an automatic test suite generator, integration with ones like [59] will be of much help for designers because there are many possible failures which may occur at different points of time and on different nodes.

For complicated interactions between a system and its environment and among the components in a system, the ability to model these interactions is desirable. It may be worthwhile to exploit the decomposition of sequence diagrams, namely vertically decomposing of lifelines and horizontally decomposing a sequence diagram into interaction fragments such as sequencing, choice and iteration.

If there is any mismatch (failure) between the requirements and the simulation traces, designers need support to identify the design fault(s). One important point here is that when there is a mismatch, the problem may either be in the design to be tested or the requirement itself. In the later case, there may be some conflicts in the requirement. Thus, support to detect inconsistencies in the requirement is necessary.

In summary, what we offer here is an initial step to model association and model-based testing. Although we have oriented this initial effort towards time-triggered applications, we believe that the framework proposed here can also be applied for event-triggered systems.

Chapter 6

Conclusion

In this thesis, we have constructed two crucial components of a system level design approach, namely the high level modeling and its intermediate representation. The chosen modeling and intermediate representation languages satisfy the requirement for system level languages, namely supporting high levels of abstraction, and separation of computation and communication.

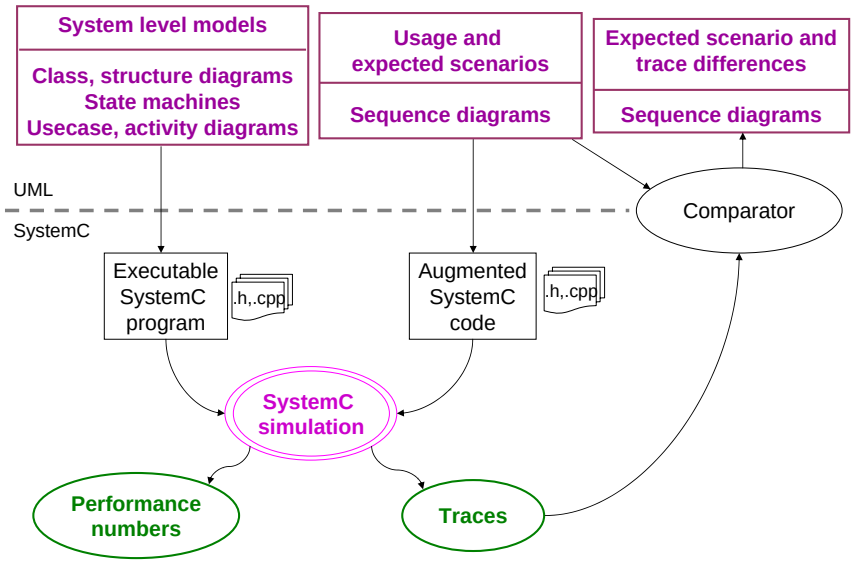

Figure 6-1: Summary of the UML and SystemC-based design framework

Fig. 6-1 summarizes our framework. First, we have proposed a modeling language

for initial specification of requirements, high level test cases, structure and behavior of a system. We have chosen a subset of UML together with some extension mechanisms. UML has a wide range of notations and diagrams for modeling different aspects of a system. Further, UML is a standard which is popular in software engineering community and is likely - with suitable augmentations - to get accepted by the embedded system community as a notational standard. The selection of the UML subset to be used in our framework, from among the numerous UML notations and diagrams, was based on their ability to capture the above high level aspects of a system and to generate code for validation and further refinement. As we have shown, class diagrams, structured classes, behavioral state machines, usecases, activity diagrams and sequence diagrams of UML together constitute a powerful conceptual and notational base for developing system level designs.

We have also proposed an executable intermediate representation in which validation through simulation can be carried out before further refinement and implementation. In our framework, SystemC is the intermediate representation. It supports modeling of complex systems through hierarchy, concurrency, separation of computation and communication. In addition, SystemC supports multiple levels of abstraction. Thus a system model after being validated at high levels of abstraction can be refined to lower levels, towards implementation. Recently, SystemC has become an IEEE electronic design language standard. This paves the way for SystemC to be used more widely in the embedded system design community. This could also help SystemC to get better support from commercial design tool providers, especially for hardware synthesis.

We have automated the model transformation process whereby the high level model is converted to the executable representation for validation through simulation. UML and SystemC, the key elements of the framework, are good basis for a model-driven system level design process. The key to realizing this potential is to *automatically* generate *executable* code from specifications developed using the cho-

sen diagram types so that one can carry out simulation, performance estimation and verification. We feel that the UML-SystemC bridge that we are advocating here can, with continued effort, help achieve this purpose.

The linkage between the UML layer and SystemC layer we have constructed serves a dual purpose. On the one hand, we use it for transforming applications described at the UML layer to SystemC code for initial simulation. On the other hand, our translation mechanism also enables us to *pull up* the platform description mechanisms to the UML layer. In this latter usage, we could consider both the executable platform description and the application models to be available at the UML layer where one can hope to do formal verification. Further one can also begin to tackle a more abstract version of the problem of mapping an application to a platform. Using our translator, a designer can then translate these two descriptions down to the SystemC level for more detailed simulation and move towards a hardware/software implementation.

Finally, we have also initiated the process of model association. Once a system model has been used to generate code to do simulation, one would like to be able to test and debug the design at modeling level. Thus, model association is crucial. But it is an area that has not been explored well in the context of system level design. In our framework, simulation results constructed at SystemC level are reflected back at UML modeling level to help designers fix design errors. In addition, usage scenarios of a system are also be captured at UML level so that a SystemC Test Driver is generated automatically in order to drive the simulation. This automatic generation saves designers much effort because the generator establishes all the connections between the Test Driver and the internal components that interact with the system's environment and fills in the triggering code of the Test Driver.

A key aspect of our framework is that it is able to handle both conventional reactive (event-triggered) and time-triggered platforms and applications. The system's structure is modeled similarly for both types of systems. In particular, class diagrams and structured diagrams are used for the complex nested structure. For reactive sys-

tems, complicated behaviors can be specified in terms of hierarchical state machines. SystemC communication channels are synthesized from the UML events. We are able to generate SystemC code from arbitrary nested structures of UML behavioral state machines.

For time-triggered applications, the behavioral state machines are of a restricted type. The transitions at the top level are triggered by time while the states corresponding to the event-triggered (dynamic) phase can contain event-triggered transitions. To take advantage of the static schedule of time-triggered applications, a SystemC simulation driver is constructed by the UML-SystemC translator to speed up simulation. Furthermore, the service provided by the time-triggered communication platform is lifted up to UML level, where the communication schedule is specified. This enables the system simulation. And this also provides enough information about the communication platform for the application developers.

We have exercised our framework using a number of event-triggered and time-triggered applications and platforms. What we have done here can also serve as design patterns for event-triggered and time-triggered system level modeling. For reactive systems, we have done three case studies with a benchmark example of a transaction level model, a case study to check the simulation performance for different sizes of systems and another case study to test out the behavioral synthesis path (with a very old and now defunct tool). The results show that our approach offers acceptable simulation speed considering we are catering for hierarchical state machines. In addition, the simulation scales well for different sizes of systems. The behavioral synthesis example indicates that integration with further refinement to implementation is possible with our framework, although synthesis is not the main focus in this work.

We have also targeted time-triggered applications. We satisfy the requirements for designing time-triggered applications, namely temporal firewall, global time and composability. We have experimented in our framework with two automotive appli-

cations, a brake-by-wire and an adaptive cruise controller system. Our results show that for time-triggered applications, simulation speed can be substantially optimized since the schedule is known in advance. Indeed our results show that even for reactive systems if we have some knowledge about the schedule of a system (which is usually the case because designers often need to come up with a good scheduling policy first), we can optimize the simulation speed. Apart from that, other simulation speed optimization techniques that we have applied based on our survey and experiences have also helped us speed up a good deal for both reactive and time-triggered systems. Our proposed design framework can be used in the early design process of automotive software. This could be especially valuable since the design of embedded automotive software is still in its infancy.

Our initial steps in model association and model-based testing show encouraging results. The usage scenario specified at UML level and its corresponding SystemC Test Driver (generated from the UML diagram) facilitate the testing process. Our simple case studies show that the simulation traces displayed at UML level help designers identify functional bugs in a system. In addition, errors such as deadlock in a communication protocol can also be revealed and to some extend, the traces can guide the designers to fix the bugs. Moreover, the behavior of a system under certain invalid conditions such as network failures can also be observed.

6.1 Future work

Currently we use UML behavioral state machines to capture the behavior of components in a system. We think that it will be important to support heterogeneous modeling which involves different models of computation. In particular, some version of dataflow graphs should be supported. UML activity diagrams can be used to capture dataflow graphs. Moreover, the modeling of systems which consist of both event-triggered and time-triggered sub-systems needs to be explored as this is the

trend in many types of applications, especially in the automotive domain.

Another important issue is how the SystemC program generated from a UML model of a system can be used in the later steps in a design process. This involves hardware synthesis and generating software to execute on the target platform. There is an emerging family of tools that support the design for both event-triggered and time-triggered systems at low levels of abstraction. Integration of our tool with such tools to support a seamless development process is desirable.

Another important set of tools to be integrated are the ones for formal verification: There are two possibilities, namely verification of UML models and SystemC program. Some projects [82, 109] on verification of UML models take in the similar UML subset. There also have been works on verification of SystemC model [79, 66, 73]. On the other hand, timing analysis frameworks such as [67] are also helpful in a system level development process. With additional effort, it should be possible to integrate our design framework with these verification tools.

As discussed, our framework helps in the modeling and validating the functionality and architecture model of a system. This can fit nicely into the Y-chart and platform based approaches of system level design. However, how to do the mapping between the functionality and the architecture has not been fully investigated in this work.

Our support for model association can be extended further by allowing designers to specify more complicated scenarios. This can be done based on the ability of UML 2.0 sequence diagrams to express interaction fragments such as sequencing, choice and iteration. Furthermore, model-based testing is also desirable. This includes generating test cases from the UML specification of a system. On this front, integrating with the works such as [59] is worth exploring.

In summary, our proposed framework is a good basis for system level design to tackle the complexity problem of embedded systems. The framework is based on the UML and SystemC languages. The automatic translation from UML to SystemC enables design validation through simulation. This framework can be applied for

both reactive and time-triggered systems. And its implementation is amenable for integration with other tools to support a full development process.

Bibliography

[1] Synopsys CoCentric SystemC compiler behavioral user and modeling guide, 2001.

[2] Systemc synthesizable subset version 1.1.18. OSCI Drafts Under Public Review, 2006.

[3] Automotive open system architecture - AUTOSAR. http://www.autosar.org, 2008.

[4] JDOM. http://www.jdom.org/, 2008.

[5] Lin-consortium website. http://www.lin-subbus.org/, 2008.

[6] MOST cooperation. http://www.mostcooperation.com/home/index.html, 2008.

[7] Ptolemy project. http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm/, 2008.

[8] Rational rose technical developer. http://www-306.ibm.com/software/awdtools/developer/technical/, 2008.

[9] SysML. http://www.sysml.org/, 2008.

[10] SystemC home page. www.systemc.org, 2008.

[11] TTTech Computertechnik AG. http://www.tttech.com, 2008.

[12] UMLGraph 5.0. http://www.umlgraph.org/, 2008.

[13] Velocity home page. jakarta.apache.org/velocity/, 2008.

[14] The Xpilot system. http://cadlab.cs.ucla.edu/soc, 2008.

[15] J. Ali and J. Tanaka. Converting statecharts into Java code. In *the Fourth World Conference on Integrated Design and Process Technology (IDPT)*, 1999.

[16] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. pages 436–450. 2007.

[17] P. Andersson and M. Hst. UML and SystemC - a comparison and mapping rules for automatic code generation. In *Forum on specification and Design Languages conference (FDL)*, 2007.

[18] T. Arons, J. Hooman, H. Kugler, A. Pnueli, and M. Zwaag. Deductive verification of UML models in TLPVS. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings*, volume 3273 of *LNCS*, pages 335–349. Springer, 2004.

[19] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Paserone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 2003.

[20] F. Basanieri, A. Bertolino, and E. Marchetti. The cow_suite approach to planning and deriving test suites in UML projects. In *Fifth International Conference on the Unified Modeling Language - the Language and its applications*, 2002.

[21] M. Bergstrm and J. Hgberg. Implementation of membership algorithms in GAST-cluster with FlexRay. Master's thesis, Computer Science and Engineering, Chalmers University of Technology, Gteborg, Sweden, 2007.

[22] Inc. Bluespec. ESL synthesis from SystemC overview. http://www.bluespec.com/products/documents/BluespecSystemCOverview.pdf, 2006.

[23] S. Bocchio, E. Riccobene, A. Rosti, and Scandurra P. A SoC design flow based on UML 2.0 and SystemC. In *International DAC Workshop - UML for SoC Design (UML-SoC)*, 2006.

[24] BOSCH. CAN resource page. http://www.semiconductors.bosch.de/en/20/can/index.asp, 2006.

[25] P. Braun, Michael von der Beeck, U. Freund, and M. Rappl. Architecture centric modeling of automotive control software. World Congress of Automotive Engineers, SAE Transactions Paper, 2003.

[26] P. Braun, Michael von der Beeck, M. Rappl, and C. Schrder. Automotive software development: A model-based approach. Congress of Automotive Engineers, SAE Transactions Paper, 2002.

[27] M. Broy. *Handbook of Real-Time and Embedded Systems*, chapter Embedded systems and software technology in the automotive domain. Chapman & Hall/CRC, 2008.

[28] M. Bruce. Distributed brake-by-wire based on TTP/C. Master's thesis, Department of Automatic Control, Lund Institute of Technology, June 2002.

[29] F. Bruschi. A SystemC based design flow starting from UML models. The 9th European SystemC Users Group Meeting, 2004.

[30] J. Cabot, R. Claris, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *Workshop on Model Driven Engineering, Verification, and Validation: Integrating Verification and Validation in MDE*, 2008.

[31] Cadence. Incisive enterprise simulator datasheet.

[32] L. Cai and D.D. Gajski. Transaction level modeling: An overview. In *1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign & System Synthesis*, 2003.

[33] M. Caldari, M. Conti, M. Coppola, S. Curaba, L. Pieralisi, and C. Turchetti. Transaction-level models for AMBA bus architecture using SystemC 2.0. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20026, Washington, DC, USA, 2003. IEEE Computer Society.

[34] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *ACM-SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.

[35] Celoxica. Agility compiler for SystemC. www.te.rl.ac.uk/europractice/vendors/agility_compiler.pdf, 2008.

[36] H. Chang, L. Cooke, M. Hunt, G. Martin, A.J. McNelly, and L. Todd. *Surviving the SOC revolution: a guide to platform-based design*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[37] CoWare. ConvergenSC model library.

[38] W. Damm and B. Jonsson. Eliminating queues from RT UML model representations. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 375–394. Springer-Verlag, 2002.

[39] W. Damm and B. Westphal. Live and let die: LSC based verification of UML models. *Sci. Comput. Program.*, 55(1-3):117–159, 2005.

[40] Decomsys website. http://www.decomsys.com/, 2007.

[41] D. Densmore. Formal refinement verification in Metropolis. Technical Report UCB/ERL M04/10, UC Berkeley, May 2004.

[42] D. Densmore. Metropolis architecture refinement styles and methodology. Technical Report UCB ERL M04/36, University of California, Berkeley, September 2004.

[43] D. Densmore, S. Rekhi, and A. Sangiovanni-Vincentelli. Microarchitecture development via Metropolis successive platform refinement. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10346, Washington, DC, USA, 2004. IEEE Computer Society.

[44] N Dhanwada, R.A. Bergamaschi, W.W. Dungan1, I. Nair, P. Gramann, W.E. Dougherty1, and I.C. Lin. Transaction-level modeling for architectural and power analysis of PowerPC and CoreConnect-based systems. *Design Automation for Embedded Systems*, 2005.

[45] J.L. Diaz-Herrera. *Handbook of Software Engineering & Knowledge Engineering*, volume 2 Emerging Technologies, chapter A survey of system level design notations for embedded systems, pages 727–756. World Scientific, 2002.

[46] B. Dion, T. Le Sergent, B. Martin, and H. Griebel. Model-based development for time-triggered architectures. In *Digital Avionics Systems Conference (DASC)*, 2004.

[47] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta. Balboa: A component-based design environment for system models. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, December 2003.

[48] F. Doucet, V. Sinha, and R.K. Gupta. Microelectronic system-on-chip modeling using objects and their relationships. In *1st Online Symposium for Electrical Engineers*, 2000.

[49] A. Rosti S. Bocchio E. Riccobene, P. Scandurra. An HW/SW co-design environment based on UML and SystemC. In *Forum on specification and Design Languages (FDL)*, 2005.

[50] Wilfried Elmenreich, Gnther Bauer, and Hermann Kopetz. The time-triggered paradigm. In *Proceedings of the Workshop on Time-Triggered and Real-Time Communication*, Manno, Switzerland, Dec. 2003.

[51] C. Erbas, S.C. Erbas, and A.D. Pimentel. A multiobjective optimization model for exploring multiprocessor mappings of process networks. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 182–187, New York, NY, USA, 2003. ACM.

[52] Th. Fhrer, B. Mller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther, and Robert Bosch GmbH. Time triggered communication on CAN. In *7th International CAN Conference*, 2000.

[53] A. Filippov and A. Borshchev. Daimler-Chrysler modeling contest: Modeling S-class car seat control with AnyLogic. Object-Oriented Modeling of Embedded Real-Time Systems OMER-2 workshop, 2001.

[54] FlexRay Consortium. FlexRay communications system, protocol specification, version 2.1, revision A. 2005.

[55] FlexRay home page. http://www.flexray.com, 2008.

[56] X. Fu and D. Hogrefe. Modeling soft state protocols with SDL. In *NETWORKING*, pages 289–302, 2005.

[57] D.D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, 2000.

[58] A. Ghosal, T.A. Henzinger, D. Iercan, C.M. Kirsch, and A. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Sixth Annual Conference on Embedded Software (EMSOFT)*, 2006.

[59] A. Goel and A. Roychoudhury. Synthesis and traceability of scenario-based executable models. In *Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2006.

[60] R. Goering. Next-generation Verilog rises to higher abstraction levels. EE Times, 2002.

[61] E. Grimpe and F. Oppenheimer. Extending the SystemC synthesis subset by object-oriented features. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 25–30, New York, NY, USA, 2003. ACM.

[62] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC.* Kluwer Academic Publishers, 2002.

[63] D. Grosse and R. Drechsler. Formal verification of LTL formulas for SystemC designs. In *International Symposium on Circuits and Systems*, 2003.

[64] T. Grotker. *System Design with SystemC.* Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[65] The OSEK group. OSEK/VDX: Time triggered operating system version 1.0. http://portal.osek-vdx.org, 2008.

[66] A. Habibi and S. Tahar. Design for verification of SystemC transaction level models. In *DATE '05: Proceedings of the conference on Design, Automation and*

Test in Europe, pages 560–565, Washington, DC, USA, 2005. IEEE Computer Society.

[67] A. Hagiescu, U.D. Bordoloi, S. Chakraborty, P. Sampath, P.V. Ganesan, and S. Ramesh. Performance analysis of FlexRay-based ECU networks. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 284–289, New York, NY, USA, 2007. ACM.

[68] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, June 1987.

[69] A. Hartman, M. Katara, and S. Olvovsky. *Hardware and Software, Verification and Testing*, chapter Choosing a Test Modeling Language: A Survey, pages 204–218. Springer Berlin / Heidelberg, 2007.

[70] B. Hedenetz and R. Belschner. Brake-by-wire without mechanical backup by using a TTP-communication network. *SAE TRANSACTIONS*, 107(6), 1999.

[71] T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, and W. Pree. From control models to real-time code using Giotto. IEEE Control Systems Magazine 23(1), 2003.

[72] A. J. C. van Gemund. Performance prediction of parallel processing systems: the PAMELA methodology. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 318–327, New York, NY, USA, 1993. ACM.

[73] D. Karlsson, P. Eles, and Z. Peng. Formal verification of systemc designs using a petri-net based representation. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1228–1233, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[74] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonolization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[75] B. Kienhuis, E. Deprettere, P. van der Wolf, and K. Vissers. A methodology to design programmable embedded systems: The Y-chart approach. In Ed F. Deprettere, Jurgen Teich, and Stamatis Vassiliadis, editors, *Embedded Processor Design Challenges, LNCS 2268*, pages 18–37, 2002.

[76] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *11th International Conference of Applications-specific Systems, Architectures and Processors (ASAP'97)*, pages 338–349, 1997.

[77] H. Kopetz and G. Bauer. The time-triggered architecture. In *IEEE Special Issue on Modeling and Design of Embedded Software*, volume 91, pages 112–126, January 2003.

[78] H. Kopetz and T. Thurner. TTP - a new approach to solving the interoperability problem of independently developed ECUs. SAE International Congress andExposition, Detroit, MI, USA. SAE Technical Paper 981107, Feb. 1998.

[79] D. Kroening and N. Sharygina. Formal verification of SystemC by automatic hardware/software partitioning. In *MEMOCODE '05: Proceedings of the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pages 101–110, Washington, DC, USA, 2005. IEEE Computer Society.

[80] L. Lavagno, G. Martin, and B. Selic. *UML for Real: Design Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003.

[81] P. Lieverse, T. Stefanov, P. Wolf, and E. Deprettere. System level design with Spade: an M-JPEG case study. In *ICCAD '01: Proceedings of the 2001*

IEEE/ACM international conference on Computer-aided design, pages 31–38, Piscataway, NJ, USA, 2001. IEEE Press.

[82] J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. Technical report, 1999.

[83] Y.C. Lin, C.C. Yang, M.Y. Hwang, and Y.T. Chang. Simulation and experimental verification of micro polymerase chain reaction chip. In *International Conference on Modeling and Simulation of Microsystems MSM*, 2000.

[84] I. Majzik, G. Pintér, and P.T. Kovács. UML based design of time triggered systems. In *ISORC*, pages 60–63, 2004.

[85] I. Majzik, G. Pintér, and P.T. Kovács. UML based Visual Design of Embedded Systems. In *Proc. The* 7th *IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS-2004)*, pages 115–120, Stará Lesná, Slovakia, Apr. 18–21 2004.

[86] G. Martin. SystemC and the future of design languages: Opportunities for users and research. In *The 16th Symposium on Integrated Circuits and Systems Design. IEEE Press*, 2003.

[87] G. Martin, L. Lavagno, and J. Louis-Guerin. Embedded UML: A merger of real-time UML and co-design. In *The 9th International Symposium on Hardware/Software Co-Design*, 2001.

[88] M. Mutz, M. Huhn, and C. Krompke. Model based system development in automotive. *SAE technical paper series 2003-01-1017*, 2003.

[89] K.D. Nguyen, I. Cutcutache, S. Sinnadurai, S. Liu, C. Basol, A. Curic, B.T. Tok, L. Xu, F.H.E. Tay, and T. Mitra. A SystemC-based fast simulator for biomonitoring applications on wireless BAN. In *Workshop on Software and Systems for Medical Devices and Services (SMDS)*, 2007.

[90] K.D. Nguyen, I. Cutcutache, S. Sinnadurai, S. Liu, C. Basol, E.J. Sim, L.T.X. Phan, T.B. Tok, L. Xu, F.E.H. Tay, T. Mitra, and W.F. Wong. Fast and accurate simulation of biomonitoring applications on a wireless body area network. Proceedings of the 5th International Workshop on Wearable and Implantable Body Sensor Networks (BSN), 2008.

[91] K.D. Nguyen, Z. Sun, P.S. Thiagarajan, and W.F. Wong. Model-driven SoC design via executable UML to SystemC. In *25th IEEE International Real-Time Systems Symposium (RTSS)*, 2004.

[92] K.D. Nguyen, Z. Sun, P.S. Thiagarajan, and W.F. Wong. *UML for SoC Design*, chapter Model-Driven SoC Design: The UML-SystemC Bridge. Kluwer/Springer, 2005.

[93] K.D. Nguyen, P.S. Thiagarajan, and W.F. Wong. A UML-based design framework for time-triggered applications. In *RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 39–48, Washington, DC, USA, 2007. IEEE Computer Society.

[94] Object Management Group. Unified Modeling Language: Superstructure, 2005.

[95] STOC Home Page. www.specc.org, 2008.

[96] P. Pedreiras and L. Almeida. Combining event-triggered and time-triggered traffic in FTT-CAN: analysis of the asynchronous messaging system. In *IEEE International Workshop on Factory Communication Systems*, 2000.

[97] A.D. Pimentel. The Artemis workbench for system-level performance evaluation of embedded systems. *Int. Journal of Embedded Systems*, 1, 2005.

[98] P. Pop, P. Eles, and Z. Peng. Schedulability-driven communication synthesis for time triggered embedded systems. *Real-Time Syst.*, 26(3):297–325, 2004.

[99] T. Pop, P. Eles, and Z. Peng. Schedulability analysis for distributed heterogeneous time/event triggered real-time systems. *ecrts*, 00:257, 2003.

[100] Rhapsody home page. http://modeling.telelogic.com/, 2008.

[101] M. Rhodin, L. Ljungberg, and U. Eklund. A method for model based automotive software development. In *12th Euromicro Conference on Real-Time Systems*, 2002.

[102] E. Riccobene, A. Rosti, and Scandurra P. Improving SoC design flow by means of MDA and UML profiles. In *3rd UML Workshop in Software Model Engineering (WiSME@UML)*, 2004.

[103] E. Riccobene, A. Rosti, and P. Scandurra. Improving SoC design flow by means of MDA and UML profiles. 3rd UML Workshop in Software Model Engineering (WiSME'2004), 2004.

[104] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A SoC design methodology involving a UML 2.0 profile for SystemC. In *Design, Automation and Test*, 2005.

[105] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, 1999.

[106] J. Rushby. Bus architectures for safety-critical embedded systems. In Tom Henzinger and Christoph Kirsch, editors, *the First Workshop on Embedded Software, EMSOFT*, volume 2211 of Lecture Notes in Computer Science, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.

[107] J. Rushby. An overview of formal verification for the time-triggered architecture. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 83–106, London, UK, 2002. Springer-Verlag.

[108] A. Sangiovanni-Vincentelli. Quo Vadis, SLD? reasoning about the trends and challenges of system level design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[109] I. Schinz, T. Toben, C. Mrugalla, and B. Westphal. The rhapsody UML verification environment. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 174–183. IEEE Computer Society, 2004.

[110] M. Schulz-Key, C.and Winterholer, T. Schweizer, T. Kuhn, and W. Rosenstiel. Object-oriented modeling and synthesis of systemc specifications. In *The Asia South Pacific Design Automation Conference (ASP-DAC)*, 2004.

[111] B. Selic. Using UML for modeling complex real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[112] B. Selic. Turning clockwise: using UML in the real-time domain. *Commun. ACM*, 42(10):46–54, 1999.

[113] V. Sinha, F. Doucet, C. Siska, and R. Gupta. YAML: A Tool for Hardware Design Visualization and Capture. In *13th International Symposium on System Synthesis*, 2000.

[114] C. Spahn. Integration of event and time-triggered systems for X-by-wire applications. Diploma Thesis, Department of Computer Science, Technische Universitat Darmstadt, 2006.

[115] J. Stroop, R. Stolpe, and R. Otterbach. Designing and testing FlexRay systems. *Automotive Electronics II*, 2004.

[116] Synopsys. Designware building block IP user guide, 2007.

[117] SystemCrafter. http://www.systemcrafter.com/, 2008.

[118] Forte Design Systems. Cynthesizer. http://www.forteds.com/products/tlmsynthesis.asp, 2008.

[119] TTAutomotive Software GmbH. Time-triggered architecture and FlexRay. 2005.

[120] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[121] Y. Vanderperren, G. Sonck, P. Oostende, M. Pauwels, W. Dehaene, and T. Moore. A design methodology for the development of a complex system-on-chip using UML and executable system models. In *Forum on Specification and Design Languages (FDL'02)*, 2002.

[122] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng. Generating test cases from UML activity diagram based on gray-box method. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 284–291, Washington, DC, USA, 2004. IEEE Computer Society.

[123] A. Wasowski. On efficient program synthesis from statecharts. In *the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 163 – 170, 2003.

[124] Q. Zhu, A. Matsuda, S. Kuwamura, T. Nakata, and M. Shoji. An Object-Oriented design process for System-on-Chip using UML. In *15th international symposium on System Synthesis, (ISSS02)*, pages 249–254, October 2002.

[125] Q. Zhu, A. Matsuda, and M. Shoji. An object-oriented design process for system-on-chip using UML. In *The 15th International Symposium on System Synthesis*, 2002.

[126] R. Zurawski. *Embedded Systems Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 2004.