UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LISANE BRISOLARA DE BRISOLARA

Strategies for Embedded Software Development Based on High-level Models

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Prof. Dr. Ricardo Augusto da Luz Reis
Advisor

Prof. Dr. Luigi Carro
Co-advisor

Porto Alegre, August 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Brisolara, Lisane Brisolara de

Strategies for Embedded Software Development Based on High-level Models / Lisane Brisolara de Brisolara – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.

129f.:il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Advisor: Ricardo Augusto da Luz Reis; Co-Advisor: Luigi Carro.

1. Sistema embarcado. 2. Software embarcado 3. Desenvolvimento de software. I. Reis, Ricardo A. da Luz. II. Carro, Luigi. III. Título.

ACKNOWLEDGMENTS

Primeiramente, gostaria de agradecer as pessoas que estiveram diretamente envolvidas no desenvolvimento do meu trabalho de tese. Meu orientador, prof. Ricardo Reis, com quem trabalhei desde o mestrado e que sempre está disposto a conversar, seja sobre trabalho, seja sobre questões pessoais. Ao prof. Luigi Carro, que dividiu as tarefas de orientação, muitas vezes tomando para si esta responsabilidade e cujas críticas e opiniões foram muito importantes para o desenvolvimento deste trabalho. Quero estender este agradecimento ao prof. Flávio Wagner que também teve uma importante contribuição e me orientou em várias ocasiões. Eu gostaria de agradecer do mesmo modo alguns colegas com quem trabalhei durante os anos de tese: Leandro Becker, Júlio Mattos, Márcio Oliveira, Francisco Assis, Ricardo Redin e Emilena Specht.

Não poderia deixar de agradecer a minha família, meus pais, Maria Luci e Arlindo Brisolara, e minhas irmãs Elisa e Cibele, pois os seus apoio e carinho foram sempre muito importantes em todas as minhas conquistas. Não poderia deixar de fazer um agradecimento especial ao meu namorado, Vinícius Pazutti Correia, por todo o companheirismo, compreensão e amor que me deram forças para vencer este grande desafio, além da paciência para ler meus textos e assistir minhas apresentações, corrigindo-me e colaborando com críticas construtivas. Gostaria de agradecer também a Rejane e Jorge Correia, que me “adotaram” e me deram todo o apoio que podiam, principalmente neste último ano de tese. Eu gostaria de também estender este agradecimento a outros membros da família, tios e primos, pois a convivência com eles, mesmo que esporádica, contribuiu para meu equilíbrio pessoal.

Aos apadrinhados, Milena e Felipe, obrigado por simplesmente poder contar com vocês. Aos demais amigos, obrigada pelos bons momentos compartilhados. Aos colegas de laboratório, agradeço os inúmeros intervalos com bate-papos, risadas e café e toda ajuda que me deram ao longo do doutorado. Também gostaria de agradecer a todos do grupo do vôlei pelos bate-bola agradáveis que me permitiram espairecer nos momentos mais estressantes do doutorado.

É importante também agradecer ao CNPq pela bolsa de doutorado, que me possibilitou dedicação exclusiva a minha formação acadêmica. À CAPES, agradeço a bolsa de estágio no exterior, que me permitiu colaborar com um excelente laboratório de pesquisa na França e ter uma experiência muito relevante para minha formação e para a qualidade deste trabalho. Eu gostaria de agradecer também a Ahmed Jerraya por ter me dado a oportunidade de trabalhar no seu grupo de pesquisa no Laboratório TIMA. Deste estágio, os frutos não foram apenas artigos e conhecimentos trocados, mas também as amizades. Gostaria de agradecer a meus colegas no TIMA, Katalin Popovici, Sang-il Han, Xavier Guerin, Patrice Gerin, Hao Shen, Kai Huang e Lei Li pela colaboração no trabalho e pelo tempo divertido que passamos juntos. Thank you all.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AG	Automatic Generated
CAAM	Combined Application Architecture Model
CASE	Computer Aided Software Engineering
COTS	Commercial-off-the-shelf
DSE	Design Space Exploration
EMF	Eclipse Modeling Framework
ESL	Electronic System-Level
FB	Functional Block
FSM	Finite State Machine
HdS	hardware dependent software
HW	Hardware
JET	Java Emitter Templates
JML	Java Modeling Language
KPN	Kahn Process Network
LSC	Live Sequence Charts
MA	Message Aggregation
MARTE	Modeling and Analysis of Real-time and Embedded Systems
MDD	Model Driven Development
MDR	Netbeans Metadata Repository
MoC	Model of computation
MPSoC	Multiprocessor System-on-chip
MSC	Message Sequence Charts
OAL	Object Action Language
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Modeling Technique
OO	Object-Oriented

OS	Operating System
PBD	Platform-Based Design
QoS	Quality of Service
RTW	Real-Time Workshop
SDF	Synchronous Data Flow
SMW	System Modeling Workbench
SoC	System-on-chip
SysML	Systems Modeling Language
SW	Software
UML	Unified Modeling Language
UML-SPT	UML Profile for Schedulability, Performance, and Time
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language for Transformation
WD	Written by Designers

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The use of techniques starting from higher abstraction levels is required to cope with the complexity that is found in the new generations of embedded systems, being crucial to the design success. A large reduction of design effort when using models in the development can be achieved when there is a possibility to automatically generate code from them. Using these techniques, the designer specifies the system model using some abstraction and code in a programming language is generated from that. However, available tools for modeling and code generation are domain-specific and embedded software usually shows heterogeneous behavior, which pushes the need for supporting software automation under different models of computation.

In this thesis, strategies for embedded software development based on high-level models using UML and Simulink were analyzed. We observed that the embedded software generation approaches based on UML and Simulink have limitations, and hence this thesis proposes strategies to improve the automation provided on those approaches, for example, proposing a Simulink-based multithread code generation.

UML is a well used language in the software engineering domain, and we consider that it has several advantages. However, UML is event-based and not suitable to model dataflow systems. On the other side, Simulink is widely used by control and hardware engineers and supports dataflow, and time-continuous models. Moreover, tools are available to generate code from a Simulink model. However, Simulink models represent lower abstraction level compared to UML ones. This comparison shows that UML and Simulink have pros and cons, which motivates the integration of both languages in a single design process.

As the main contribution, we propose in this thesis an integrated approach to embedded software design, which starts from a high-level specification using UML diagrams. Both dataflow and control-flow models can be generated from that. In this way, an UML model can be used as front-end for different code generation approaches, including UML-based one and the proposed Simulink-based multithread code generation.

Keywords: embedded software, embedded systems, software development.

Estratégias para Desenvolvimento de Software Embarcado Baseadas em Modelos de Alto Nível

RESUMO

Técnicas que partem de modelos de alto nível de abstração são requeridas para lidar com a complexidade encontrada nas novas gerações de sistemas embarcados, sendo cruciais para o sucesso do projeto. Uma grande redução do esforço pode ser obtida com o uso de modelos quando código em uma linguagem de programação pode ser gerado automaticamente a partir desses. Porém, ferramentas disponíveis para modelagem e geração de código normalmente são dependentes de domínio e o software embarcado normalmente possui comportamento heterogêneo, requerendo suporte a múltiplos modelos de computação.

Nesta tese, estratégias para desenvolvimento de software embarcado baseado em modelos de alto nível usando UML e Simulink são analisadas. A partir desta análise, observaram-se as principais limitações das abordagens para geração de código baseadas em UML e Simulink. Esta tese, então, propõe estratégias para melhorar a automação provida por estas ferramentas, como por exemplo, propondo uma abordagem para geração de código *multithread* a partir de modelos Simulink.

A comparação feita entre UML e Simulink mostra que, embora UML seja a linguagem mais usada no domínio de engenharia de *software*, UML é baseada em eventos e não é adequada para modelar sistemas *dataflow*. Por outro lado, Simulink é largamente usado por engenheiros de hardware e de controle, além de suportar *dataflow* e geração de código. Porém, Simulink provê abstrações de mais baixo nível, quando comparado a UML. Conclui-se que tanto UML como Simulink possuem prós e contras, o que motiva a integração de ambas linguagens em um único fluxo de projeto.

Neste contexto, esta tese propõe também uma abordagem integradora para desenvolvimento de *software* embarcado que inicia com uma especificação de alto nível descrita usando diagramas UML, a partir da qual modelos *dataflow* e *control-flow* podem ser gerados. Desta maneira, o modelo UML pode ser usado como *front-end* para diferentes abordagens de geração de código, incluindo UML e a proposta geração de código *multithread* a partir de modelos Simulink.

Palavras-chave: *software* embarcado, sistemas embarcados, desenvolvimento de *software*.

1 INTRODUCTION

‘Embedded everywhere’ is an expression that is getting materialized with the new generation of computer systems. This is a reality in sectors such as automotive, aeronautics, telecommunications, consumer electronics, and medical devices. Such embedded computational systems are often implemented as heterogeneous systems-on-a-chip (SoCs), which are usually composed of dedicated hardware modules, programmable processors, memories, interface controllers, and software components.

Usually, embedded systems have hard constraints regarding performance, memory, power consumption, dimensions, and weight, among other aspects. In addition, such systems are increasingly required to operate in real-time, bringing the necessity to ensure that system results are not only correctly computed, but also delivered at the precise times. At the same time, the life cycle of embedded products becomes increasingly tighter. In this scenario, productivity and quality are simultaneously required in embedded systems design in order to deliver competitive products. That makes the design of such embedded systems an ever-growing challenge, demanding new strategies and tools to improve the design productivity.

Platform-Based Design (PBD) is a successful approach that implements a meet-in-the-middle strategy to maximize the reuse of pre-designed components and, consequently, improve the design productivity (VERKEST, 2000) (SANGIOVANNI-VINCENTELLI, 2004). In platform-based design, design derivatives are mainly configured by software.

Burch (2001) indicated that the interest on software-based implementation has risen due to the increase in computational power allowing one to move more functionality to software. It is also an effect of the rising costs for hardware development that motivates the reuse of pre-defined platforms. Product differentiation is then achieved by the software. For those reasons, the software development is where most of the design time is spent, and is the largest cost factor in embedded system design (GRAFF, 2003). This scenario motivates the investigation of strategies to accelerate the embedded software development by process automation.

In the software engineering area, Computer Aided Software Engineering (CASE) tools are widely used to automate the development process. Since conventional software is usually suited for a single domain, most of the software automation approaches focus on the management of huge domain-specific systems. However, embedded software usually shows heterogeneous behavior, which applies to systems whose respective models require different models of computation (MoCs) (EDWARDS, 1997), like stream processing (dataflow), control flow, and continuous time. For example, the specification of a mobile phone requires not only digital signal processing for the telecommunication domain, which is a time-discrete MoC, but also sequential logic programs to describe several available applications (e.g. contacts and alarm clock). This pushes the need for supporting software automation under different models of computation, a task not completely supported by any current software automation tool. In addition, embedded software usually has memory and power restrictions, which makes the use of traditional CASE tools not feasible, but this is outside the scope of this thesis.

Simultaneously to PBD, the use of higher abstraction levels has been adopted in order to deal with the complexity growth of systems and to increase the design productivity. Selic (2003) and Gomaa (2000) argue that the use of techniques starting from higher abstraction levels is the only viable way of coping with the complexity that is found in the new generations of embedded systems, being crucial to the design success.

The use of higher abstraction levels hides details of implementation in the programming language, facilitating the system specification that will be on the model level, instead of code level. Using this approach, models of embedded systems should evolve from high level views into actual implementations, ensuring a relatively smooth and potentially much more reliable process as compared to traditional forms of software engineering. The translation of the high-level model into an executable description should be automatic, but depending on the modeling notation, it may need different degrees of designer interaction. The high-level modeling language should be able to express both the application requirements and the functional specification. Also, it should provide facilities to allow model validation, as well as features that can be used to guide implementation. Regarding to modeling approaches, many models have been proposed for embedded software specification, but no consensus is reached to any particular model that is good for all applications.

Two widely used and distinct approaches highlight, one that is functional-based like provided by Simulink and another one that is object-oriented like provided by UML-based tools. Traditionally, the functional block (FB) modeling approach has been used by the signal processing and control engineering communities for the development of embedded systems. This approach has been widely accepted in industrial designs, driven by an extensive set of design tools, as for instance Matlab/Simulink (MATHWORK, 2003). On the other hand, the UML modeling language is considered the *de facto* modeling notation for any object-oriented (OO) system and has gained in popularity also for real-time embedded systems specification and design. Efforts that describe the use of UML in different phases of an embedded system design are shown in (LAVAGNO, 2003). In the context of this work, both modeling approaches are evaluated regarding modeling, code generation and design exploration capabilities. These analysis results are found in (BRISOLARA, 2004; BRISOLARA, 2005b).

From this analysis, we observed that the embedded software generation approaches based on UML and Simulink have limitations, and so we propose strategies to improve the automation provided on that approaches. As UML tools usually required more code to be specified by the designer using some action language to specify method bodies, a way to abstract the behavior specification is proposed in order to reduce the amount of code that must be written by designer. On the other hand, observing the limitations of Simulink-based tools regarding to code generation for MPSoC architectures, a Simulink-based multithread code generation strategy is proposed in the context of this work (BRISOLARA, 2007a).

The comparison between UML and Simulink shows that both modeling approaches present pros and cons, which motivates researchers to find a way to simultaneously exploit the benefits of both modeling languages in a single design process. Moreover, recent efforts show that both UML and Simulink are considered attractive for Electronic System Level (ESL) design (DENSMORE, 2006). Boldt (2007) proposes the integration of Simulink models into UML models in the Rhapsody tool (TELELOGIC, 2007). In addition, SysML (OMG, 2006) was proposed as an extension to UML for systems engineering applications, with a higher degree of integration with the FB paradigm. However, the first SysML language specification was so close to UML that it is difficult to clearly define its improvements. Additionally, the available SysML modeling tools have not been evaluated yet for compliance or modeling capabilities.

As most automation tools are domain-specific, Reichmann (2004) proposes a way to integrate models developed with different tools (UML, Simulink and Statemate). Using this approach, a heterogeneous system is partitioned into sub-module. Each of them can be modeled using the more appropriated tool, and domain-specific code generators are used to generate code for it.

In this thesis, we propose a way to integrate UML and Simulink in a single design flow, allowing one to start with an UML model and decide which is the most appropriated tool to generate code for the system module (BRISOLARA, 2007b). Differently of the Boldt and Reichmann approaches, our approach uses a single language for initial specification, i.e. UML, and proposes the automatic mapping from UML to Simulink. Besides the mapping, our tool performs three kinds of optimizations: inference of communication channels, thread grouping and loop detection. Inference of communication channels and thread grouping are used to build a model from that a multithread code targeted to a multiprocessor architecture can be generated. In addition, loop detection is provided to insert temporal barriers in a dataflow model, avoiding deadlock. The proposed integration allows designers to work at a higher abstraction level, avoiding the necessity of building or modifying Simulink models directly, which means abstracting about low-level details like signals and ports.

1.1 Thesis contributions

Firstly, this thesis contributes with an analysis and comparison between two widely used modeling approaches, UML and Simulink, using a case study that is a heterogeneous system with dataflow and control-flow modules (BRISOLARA, 2005b).

Strategies for embedded software generation from high-level models described in Simulink or using UML-based tools are proposed here and are also contributions of this thesis. In the UML-based strategy, the gap between model and code is bridged through of the use of a higher abstraction language. This strategy was finished because the definition of another language is a hard work and usually designers are not open to learn a new language. Moreover, during the thesis period, UML2 was defined, solving some problems in UML-based code generation. On the other hand, as Simulink is a commercial tool widely used and that already provide code generation capabilities, we also propose an approach to generate multithread code target to multiprocessor architectures from a Simulink model. The Simulink-based strategy has an optimization step that reduces the communication overhead during the code generation (BRISOLARA, 2007a).

Finally, this thesis proposes a way to integrate UML and Simulink in a single design flow (BRISOLARA, 2007b). In this approach, UML is used as the initial specification and Simulink can be generated from UML. In this way, an UML model can be used as front-end for different code generation approaches, including UML-based one and our Simulink-based multithread code generation.

1.2 Thesis organization

The remaining of this thesis is divided as follows: Chapter 2 gives an overview of the state-of-the-art of modeling approaches, languages and tools used in the embedded system domain. That chapter also presents a comparison between UML and Simulink modeling approaches through a case study. Chapter 3 addresses the UML-based strategies for software generation and presents a proposal to solve the limitation of the existing code generation approaches. Chapter 4 presents a multithread code generation approach able to generate multithread code target to a multiprocessor architecture from a Simulink model. Chapter 5 explains the integration of UML and Simulink in a single design flow proposed in this thesis as a way for supporting software automation under different models of computation. Finally, chapter 6 concludes this text with final remarks and future works.

2 HIGH-LEVEL MODELS AND ASSOCIATED TOOLS

Current research on embedded systems design emphasizes that the use of techniques starting from higher abstraction levels is crucial to the design success. Some authors like Douglass (1998), Gomaa (2000), and Selic (2003) argue that this approach is the only viable way of coping with the complexity that is found in the new generations of embedded systems. Using this approach, models of embedded systems should evolve from high level views into actual implementations, ensuring a relatively smooth and potentially much more reliable process as compared to traditional forms of engineering.

The combination of abstraction and automation has inspired a set of modeling technologies, and corresponding development methods, collectively referred to as model-driven development (MDD) (SELIC, 2006). This chapter presents the state-of-art on strategies for embedded software development based on high-level models. In addition, a case study is used to compare two widely used strategies, UML-based and Simulink-based, regarding to the main capabilities required in designing embedded software.

2.1 Embedded software from high-level models

Effective design of embedded computer systems requires the capture of the system specification using high-level models in a model-centered approach in order to cope with the increasing complexity. This high-level model should reflect the nature of the application domain and the used high-level modeling language should be able to express both the application requirements and the functional specification.

Once a specification is captured, the design process should progress towards implementation via well-defined stages (SANGIOVANNI-VICENTELLI, 2001). Tools are required to automate the model refinement and guide implementation. To obtain an embedded software implementation from the system specification, some tools provide code generation capabilities (e.g. Simulink/Stateflow, ASCET). Before the implementation, it is interesting to validate the specification, by simulation or formal verification.

With the widespread use of platform-based design, most embedded applications are developed by simply mapping the application onto the target platform. In this way, the mapping allows one to configure and refine the system until the implementation, and constraint-driven mapping can conduct the design space exploration. To support this, analysis tools are required to evaluate intermediate results with respect to the design constraints, avoiding solutions that are not good enough. To do that, simulation and estimation tools are required.

Sangiovanni-Vicentelli (2006) presents a classification for Electronic System-Level (ESL) tools and languages, focusing on the platform based design. With base in this work and in our background, we defined the Table 2.1, where existing tools dedicated to embedded systems design are listed, including academic and industrial ones. We focus our analysis on the tools more closed to the embedded software development, which is the main focus of this thesis. The tools are analyzed according to the design step where it can be used, its features and the model of computation (MoC) and/or the abstraction supported for them. Besides specification capabilities, code generation and design space exploration features are also important for automating the embedded system design, and thus, tools that focus on these aspects are also listed in this table.

Table 2.1: Tools for embedded system design

Provider	**Tools**	**Focus**	**Abstraction**
National Instruments	Labview	Control application development	LabView prog. languages
Mathworks	Simulink, Stateflow, RTW	Modeling, algorithm design, and SW development. Emphasis on control and dataflow embedded systems	Timed dataflow (discrete- and continuous-time), FSMs
Esterel Technologies	SCADE, Esterel Studio	Code gen for safely-critical applications such as avionics and automotive	Synchronous
ETAS	Ascet	Modeling, algorithm design, code gen, and SW development, with emphasis on the automotive market	Ascet models
Univ. of California, Berkeley	Ptolemy II	Modeling, simulation, design of concurrent, real-time, embedded systems	All MoCs
Royal Institute of Technology Stockholm	SML-Sys, ForSyDe*	SMS-Sys: Formal multi-MoC framework based on formal semantics and functional paradigms; ForSyDe: Capture system functionality based on a synchronous model	Multi- MoC (SML functional lang.), Synchronous*
I-Logic	Rhapsody, Statemate	Real-time embedded system applications	UML-Based
Seoul National Univ.	Peace	Codesign-environment for rapid development of heterogeneous digital systems	Ptolemy- based
dSpace	Target-link, RTI-MP	Optimized code gen for single-CPU and for multi-processor systems	Simulink models
Univ. of California, Berkeley	Metropolis	Operational functionality, arch. capture, mapping, refinement, and verification	All MoC (meta-model language)
Vanderbilt Univ.	Milan, GME, Desert	Support for domain-specific languages, and design space exploration (DSE)	UML-based and XML-based
Delft Univ. of Technology	Artemis, Sesame, Spade	Methods and tools to model and design SoC-based systems, DSE	KPN and UML
Tampere Univ. of Tech. and Nokia	Koski	DSE, code generation	UML state diagrams and KPN

2.1.1 Specification

Several tools for design capturing the high-level specification can be found in the industry and in the academy, varying the supported model of computation (MoC) and the used languages. Usually, these tools support the model simulation and code generation. Commercial packages such as LabView (NATIONAL INSTRUMENTS, 2006), Simulink (MATHWORKS, 2003a), ASCET-SD (HONEKAMP, 1999), and SCADE (ESTEREL TECHNOLOGIES, 2007) allow modeling and development of embedded control systems based on functional-block specifications. Commonly in these environments, the designer composes a system through the instantiation of pre-existing components available in a library.

Labview uses dataflow programming through a graphical interface to allow a designer to model and simulate control system using real-world stimuli. This tool provides a great number of functions for signal processing, analysis and advanced mathematics. ASCET-SD, from ETAS, supports modeling, simulation, and rapid prototyping of automotive embedded software modules and, in addition, it provides optimized code generation for various microcontroller targets. SCADE provides modeling of dataflow and state machines and code generation for safety-critical applications, such as avionics and automotive. SCADE checks model completeness and determinism, including cycle detection in nodes. The tools from Mathworks allows one to model a system through functional block diagrams using Simulink and/or through finite state machines (FSM) described using Stateflow (MATHWORKS, 2003b). Simulink representation language handles discrete dataflow and continuous time and FSM by the integration with Stateflow tool. Many embedded application have been successfully developed using these tools. However, these tools are domain-specific and only support fixed MoC.

In this context, academic research projects, like Ptolemy (2004) and SML-Sys (MATHAIKUTTY, 2006), have addressed the heterogeneity of embedded systems, proposing multi-MoC modeling frameworks that support the simulation of heterogeneous systems. PtolemyII (BHATTACHARYYA, 2007), the version presently under development in the Ptolemy project, includes a growing suite of domains, each of which realizes a MoC. It also includes a component library. The system model in Ptolemy can be described by instantiation of pre-existing components through a graphic interface or components defined in Java by the user. The main advantage of this project is that it is open-source and supported MoC and components can be extended.

Another example of multi-MoC, SML-Sys uses formal semantics and is based on functional paradigm, being readily susceptible to formal analysis. Furthermore, executable models in SML-Sys can be translated to VHDL/Verilog descriptions using ForSyDe (SANDER, 2004). Different from Ptolemy, knowledge on functional languages is required to use the SML-SyS framework.

On the other hand, object-oriented approaches, like UML-based, have gained popularity for embedded systems design. UML supports several diagrams that can be used to specify different graphical views of the system. Recently, several proposals of use of UML for embedded systems can be found in (LAVAGNO, 2003) and (MARTIN, 2005), which were motivated by the huge popularity of this language to specify computation systems, using object-oriented approaches. In this context, UML tools such as Artisan Studio (ARTISAN SOFTWARE, 2007), Rhapsody (TELELOGIC, 2007) (GERY, 2002), and MagicDraw (NO MAGIC, 2007) have also been considered for embedded software specification.

Many modeling approaches and tools have been proposed for embedded software specification, but there is no model that is more appropriated or good for all applications.

2.1.2 Code generation

Simulink and Ptolemy are examples of embedded software code generation tools, which generate code from functional blocks models. Regarding the code generation functionality, Simulink with Real-Time Workshop (MATHWORKS, 2004), from Mathworks, is probably the most widely used environment in the industry. The Real-Time Workshop takes a Simulink model as the input and generates C code as output. The Real-Time Workshop Embedded Coder, which is an extension for RTW, generates C code from Simulink and Stateflow models, enabling the code generation form data and event-based models. TargetLink (DSPACE, 2005), from dSPACE, is another commercial tool with focus on the generation of efficient code from Simulink/Stateflow models.

Ptolemy supports the modeling and simulation of heterogeneous models, but it has limited implementation capabilities for models other than dataflow (BUCK, 2000). At present, Ptolemy II proposes two different code generation approaches (ZHOU, 2007). In the first one, the code generator called Copernicus generates Java code (.class) from non-hierarchical Synchronous Data Flow (SDF) models, using a component-specialization framework built on top of a Java compiler. The second approach is a template based code generation system, in which a component called “codegen helper” is used to generate code for a Ptolemy II functional block (actor) in a target language. Currently, this template based code generator produces C code for synchronous dataflow (SDF), finite-state machines (FSM) and heterochronous dataflow models (HDF). The later is an extension of SDF that permits dynamic changes of production and consumption rates without sacrificing static scheduling. This code generator consists of actor templates (called helpers) that contain C code stubs that are stitched together. However, presently only a subset of actors has helpers already described. Although it is an interesting approach, a large amount of work is yet required to implement templates (helpers) for other widely used components and templates for different target languages before having a powerful code generation environment.

Several UML-based tools have code generation capabilities, but some tools generate only code skeletons for class diagrams, while others generate also behavioral code from state diagrams. MagicDraw (NO MAGIC, 2007) is an example of tool that support only generation of code skeleton from the static structure. On the other hand, Artisan Studio, Rhapsody, UniMod and BridgePoint UML Suite (MENTOR GRAPHICS, 2005) are examples of tools that support generation of complete code from UML models. UniMod defines a methodology for designing object-oriented event-driven applications, focusing on execution and code generation from UML state diagrams. Rhapsody (GERY, 2002) allows creating UML models for an application and then generates C, C++ or Java code for the application. These tools support complete code generation, but only based on UML state diagrams, so they are more appropriate for event-based systems. Recently, Telelogic launched the new version of Rhapsody that provides the code generation from flowcharts (activity diagrams) used to specify complex algorithms (TELELOGIC, 2007).

Besides code generation, several UML-based tools provide reverse engineering capabilities, for example, the MagicDraw tool supports reverse engineering from Java and C++ code (e.g. Java or C++ code to class diagram, Java code to sequence diagrams, etc.) and facilities to maintain the coherency between code and model.

With the increasing interest on multiprocessor platforms for embedded systems, researchers have addressed the code generation for multiprocessor platforms. Real-Time Interface for Multiprocessor Systems (DSPACE, 2005), from dSPACE, generates software code from a specific Simulink model for multiprocessor systems. However, the software code generated by RTI-MP is targeted to a specific architecture consisting of several COTS processor boards and the main purpose is high-speed simulation of control-intensive applications. Since multiprocessor platforms are becoming more popular, flexible and powerful code generation approaches are desired to aid designers in the difficult task of programming these platforms. This is discussed again in chapter 4, where a new code generation approach is proposed.

2.1.3 HW/SW co-design and design space exploration

ForSyDe (SANDER, 2004) starts at a higher abstraction level, with a synchronous formal specification model, and synthesizes VHDL and C, generated for the HW and SW implementation, respectively. The synthesis process is divided into two phases. In the first phase, the specification model is refined into a more detailed implementation model by design transformations. The second phase is the mapping from the implementation model onto a given architecture and comprises activities like partitioning, allocation of resources and code generation. The system specification used in the ForSyDe environment is made in Haskell. This language does not provide the high abstraction desired by the designers, besides requiring them knowledge over yet another language.

Metropolis (BALARIN, 2001) is HW/SW co-design framework, which integrates modeling, simulation, synthesis and verification tools. In order to support multiple MoC, a MetaModel language is used in Metropolis. However, the generality of the MetaModel language creates difficulties for its usage by users and tool developers. Only manual design space exploration is supported in Metropolis.

Milan (MOHANTY, 2002) is a hierarchical design space framework based on Generic Modeling Environment (GME) (LEDECZI, 2001), which is a framework for creating domain-specific modeling languages. For design space exploration, Milan uses DESERT (NEEMA, 2003) that is considered a semi-automated tool, because once the design space has been specified, it performs optimization and automatically indicates the optimal design.

Other examples of design space exploration environments, SPADE (LIEVERSE, 2001), Artemis/Sesame (PIMENTEL, 2001) (PIMENTEL, 2006) and Koski (KANGAS, 2006) abstract the application model using Kahn Process Network, KPN, (KAHN, 1977), and that application model is mapped to the architecture model during the design space exploration. SPADE (LIEVERSE, 2001) is a system-level performance analysis methodology and tool which uses trace-driven simulation for exploration purposes. Based on SPADE, Pimentel proposed Artemis (2001) and Sesame (2006). Artemis is a methodology for heterogeneous embedded systems modeling, while

Sesame is an environment targeted to provide modeling and simulation methods and tools for design space exploration of heterogeneous embedded systems. Koski (KANGAS, 2006) is an UML-based MPSoC design flow which provides an automated path from UML design entry to FPGA prototyping, including the functional verification and the automated architecture exploration. However, all these approaches still require the designer to manually specify the behavior for each process in the KPN.

2.2 Analysis of the state-of-the-art

Most of the academic and commercial solutions for software automation focus on the management of huge domain-specific systems, focusing in a single-domain such as databases SQL, web-based systems, or XML-based data sources and in a particular language. That is because conventional software is usually suited for a single domain. However, most complex embedded systems have a heterogeneous behavior and multiple MoC are required to describe such behaviors. Moreover, tools that automate general and conventional software development are not aware of code optimizations, a crucial step for embedded systems because of their tight restrictions.

As shown in section 2.1, the embedded system research area is very active. With the increasing complexity of embedded software and the interest in software-based embedded systems, several efforts have addressed the limitation on software development approaches and common difficulties found in designing embedded systems (e.g. heterogeneity, hard constraints, etc.). In this context, several tools have been proposed to automate the implementation from high-level models and the code optimization. For example, Telelogic Tau provides the Agile C that is a code generator dedicated to small footprint and high-performance applications. However, for embedded software design, usually power is an important issue and all the physical aspects (performance, memory and power) need to be evaluated to check if the solution meets the system requirements. Moreover, the existing tools have some limitations and frequently are domain-specific.

Despite of the huge investigation on strategies to accelerate the embedded software development, the existing tools are somewhat limited, and they do not cover the full spectrum of embedded system design. As result of the analysis of the state-of-art, it was found that none of the presented approaches targets the ultimate goal of providing appropriated abstraction (higher abstraction, multi-MoC) to increase software production and quality, with the necessary code generation and design space exploration capabilities. Nonetheless, this study shows that there are two high level modeling approaches in evidence that are functional block and object-oriented with UML.

Traditionally, the functional block (FB) modeling approach has been used by the signal processing, industrial automation, and control engineering communities for the development of embedded systems (JOHN, 2001). These models are widely accepted in industrial design, driven by an extensive set of design tools, as for instance, Matlab/Simulink (MATLAB, 2004). Features like modularity, abstraction level, and reusability contributed to the popularity of this modeling approach. On the other hand, object-oriented approaches with the Unified Modeling Language (UML) are widely used in software design. UML is considered by far the most-used modeling notation for software engineers. Recently, UML has gained in popularity as a tool for specification and design of embedded systems and SoCs. In (LAVAGNO, 2003) one can find several efforts that describe the use of UML during the different phases of an embedded system design process.

Observing that both UML and Simulink are considered attractive for Electronic System Level (ESL) design, these two widely used domain-specific modeling approaches are analyzed in more detail in section 2.3.

2.3 Comparison between UML-based and Simulink-based approaches

This section presents a comparative analysis of UML and Simulink modeling approaches, which is driven by a case study. The modeling capabilities of both approaches are evaluated, as well as capabilities of tools based on UML and Simulink are analyzed. The results of this analysis were published in (BRISOLARA, 2005b). Although the UML models used in this case study follows notations from UML 1.3, which was the language version available when this case study was published, we extend here this analysis considering also capabilities provided by UML2 and SysML, recently defined by OMG.

2.3.1 Functional block modeling and Simulink

In the functional block (FB) approach, applications are designed by connecting several FBs. This modeling language does not allow the designer to express system requirements. Therefore they start modeling already thinking of the solution for the problem under consideration. Our modeling starts with the functional decomposition, and the result is the definition of the modules that interact during the system execution. Each FB output must be connected with an appropriate input, coming from a FB or another model element. The communication among blocks occurs through the data exchange by the interfaces instead of message exchange used in object-oriented approaches. The behavior for each block is described using different languages oriented to functional blocks, like languages for programmable controllers (PLCs) and Matlab/Simulink (HEVERHAGEN, 2003).

Simulink, from Mathworks, is a block diagram commercial tool and language for the system modeling and simulation, which supports multiple models of computation (MoC) such as continuous time, discrete time, and event-oriented (by the integration with Stateflow). For example, using this tool, a block behavior can be described through transference functions, discrete equations, C or Matlab code, or state machines. A wide variety of tools (Stateflow, Real Time Workshop, etc.) and libraries with pre-defined blocks are integrated in the Simulink environment. The complexity of the blocks varies from simple adders or multipliers to complex filter algorithms. The functionalities of a block can be specified as a C or Matlab code or instantiating pre-defined components from the libraries. Simulink is suited for control engineering and digital signal processing applications.

2.3.2 Object-oriented modeling and UML

On the other hand, the object-oriented (OO) modeling paradigm has gained popularity over the last years among the general-purpose software design community. The object-oriented design and analysis uses concepts like design, polymorphism, and inheritance to model structural and behavioral system aspects. The use of high-level abstraction turns the design and implementation process easier, reducing design time.

As a result of a standardization process among different object-oriented design methodologies, the Object Management Group (OMG) promoted the creation of the Unified Modeling Language (UML) (OMG, 1999), which is currently in version 2.1.1 (OMG, 2007a). UML is considered the *de facto* modeling notation for any OO system.

With the production of SoC with large amount of memory, the use high-level languages and object-oriented approaches could be considered in embedded software design. With the interest by OO methodologies, the UML language gained also popularity in embedded system domain. Sgroi (2002) justifies this attention by the rich graphical notation and modeling power provided by this language that enables the capturing of structural and behavioral aspects in different abstraction levels. In addition, using OO concepts of UML, a definition of a class is made of its interface and its behavior. This distinction between definition and instances allows the development of libraries of reusable components. Another contribution from OO is the ability to define a component by inheriting features from another one, which again improves the reuse of components.

In addition, the UML has mechanism to extend the language by the definition of profiles for specific domains. The UML-SPT (OMG, 2003) and the QoS&FT (OMG, 2007b) are examples of profile proposed by OMG to model “Schedulability, Performance and Time” and “Quality of Service and Fault Tolerance”, respectively. However, these profiles cannot fully support the needs of the real time domain. OMG has therefore proposed the MARTE (OMG, 2005)(RIOUX, 2005), which includes the previous UML-SPT profile and affords generic concepts required to model real time aspects in both qualitative and quantitative terms and for schedulability or performance analysis on a model. It includes a set of modeling artifacts for embedded system specification, supporting asynchronous and synchronous computation models used in the RT domain. In addition, MARTE includes extensive models of standard platforms (POSIX, OSEK, etc.).

2.3.3 Case study: Comparison between UML and Simulink models

This section presents two different models developed to compare the object oriented modeling approach of UML to the FB modeling approach provided by Simulink. Our goal here is to analyze how suitable are these two approaches for the embedded system design. The results of this analysis were presented in (BRISOLARA, 2005b), where more detail about the used methodology can be found.

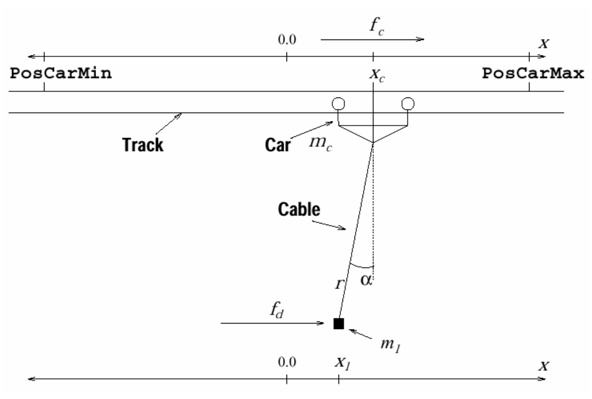


Figure 2.1: Crane system

The case study consists of a crane control system, proposed as a benchmark for system level modeling (MOSER, 1999). Once the user defines a position for the crane, the control system should activate the motor and move the crane to the desired point. Special care must be taken with speed and position limits while the crane is moving, to guarantee the safety of the transported load. Therefore, constant monitoring is needed to avoid unexpected situations. This system incorporates hard real time constraints. Figure 2.1 gives an overview of the system.

2.3.4 FB model

The Simulink environment was used to define the functional-block model of the Crane control system. Following the FB approach, the application was designed by connecting several functional blocks through of data links. In this approach, different hierarchical levels can be used in the model. As shown in Figure 2.2, the modeling resulted in four high level modules organized hierarchically, as follows: PlantActuators, Sensors, ControlAlgorithm, and JobControl. Each module has its intrinsic behavior and is further detailed along this section.

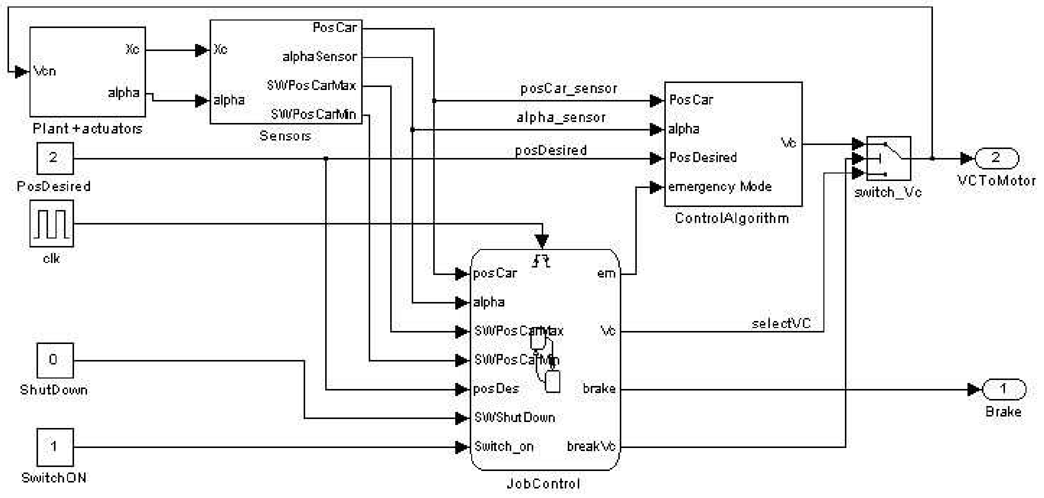


Figure 2.2: Crane model using Simulink

The crane system is composed of both data driven and event driven parts, as can be observed in Figure 2.2. The JobControl module is represented by a finite state machine (event based), while the other modules are data driven. Figure 2.3 shows a view of the JobControl module, which is composed by five states: Power_off, Init, PosDesiredTest, NormalMode and EmergencyStop.

The NormalMode is a composite state, containing two concurrent states, Diagnosis and Control, as can be observed in Figure 2.4. The Diagnosis module runs in parallel with the control algorithm. This module is responsible for monitoring the position and alpha sensors, indicating when some risk condition occurs. On the other hand, the control is responsible for detecting the braking condition for the control algorithm.

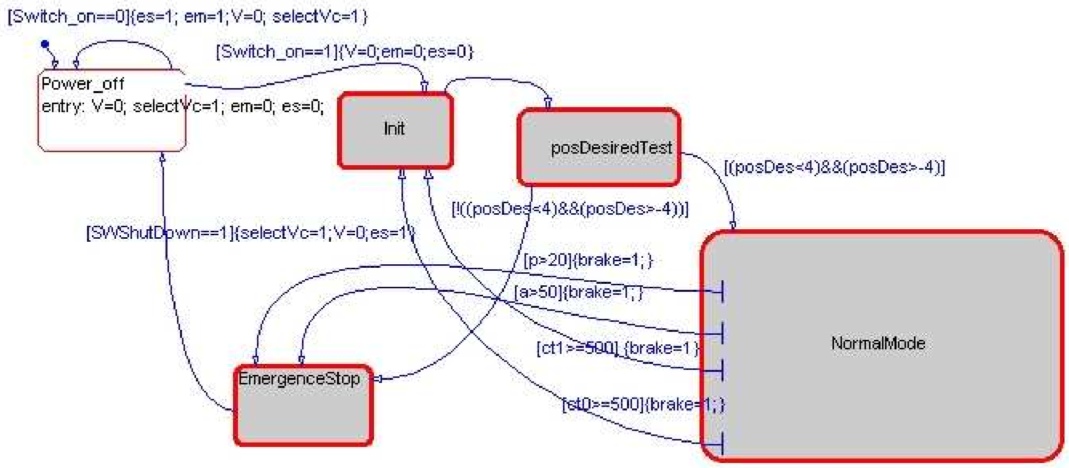

[Switch_on==0]{es=1; em=1; V=0; selectVc=1}
[Switch_on==1]{V=0;em=0;es=0}
Power_off
entry: V=0; selectVc=1; em=0; es=0;
Init
posDesiredTest
[(posDes<4)&&(posDes>-4)]
[!((posDes<4)&&(posDes>-4))]
[SWShutDown==1]{selectVc=1;V=0;es=1}
[p>20]{brake=1;}
[a>50]{brake=1;}
[ct1>=500] {brake=1}
[ct0>=500]{brake=1;}
NormalMode
EmergenceStop

Figure 2.3: Crane JobControl

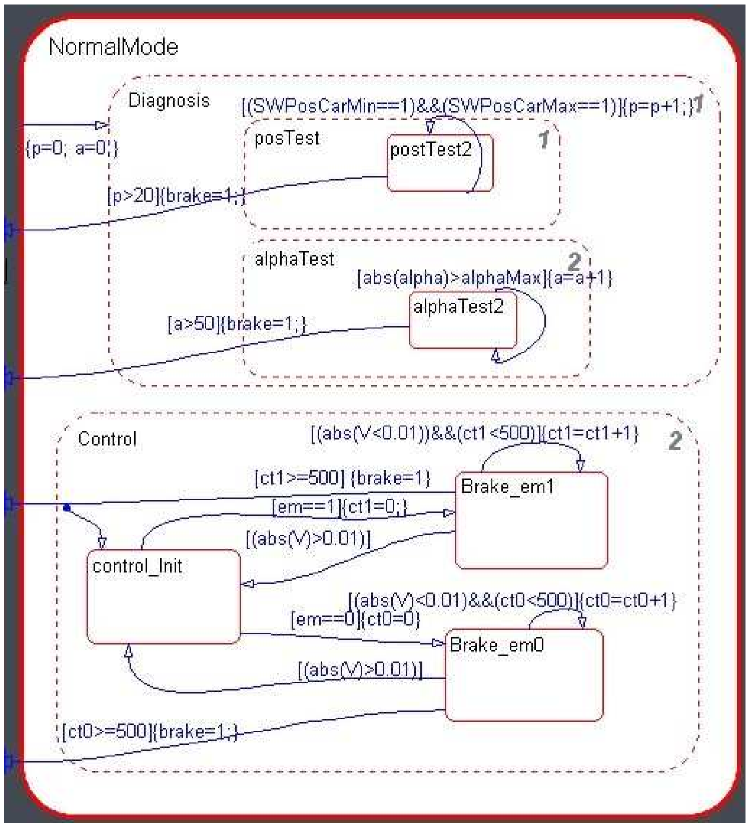


Figure 2.4: Crane JobControl – NormalMode

Figure 2.5 illustrates details of the ControlAlgorithm module, which is responsible for computing the control algorithm of the crane motor. This module receives the position of the car (posCar), the alpha angle of the cable (alpha), and the desired position of the load (PosDesired). The ControlAlgorithm computes a set of equations and determines the voltage (VC) that is applied to the crane motor. This FB contains two implicit MoCs, which are characterized as continuous time and discrete time, respectively. For example, it contains a discrete space state component used for differential equations resolution (top left), which is combined with those components that work in the time continuous domain. The control algorithm is periodic, with a period of 10 ms. Although this timing restriction could be represented in the model using a clock, this is not a suitable way of expressing timing requirements. For instance, no deadline can be stated, representing missing information required to perform schedulability analysis.

The Sensors module is responsible for reading the sensors and works with a fixed cycle time of 2 ms. Although this FB is not shown in this chapter, we observe that it has the same problems previously stated for the control algorithm regarding the representation of timing restrictions. Besides the position and angle sensors, there are two other sensors for indicating when the car is beyond the track limits (minimal and maximum car position).

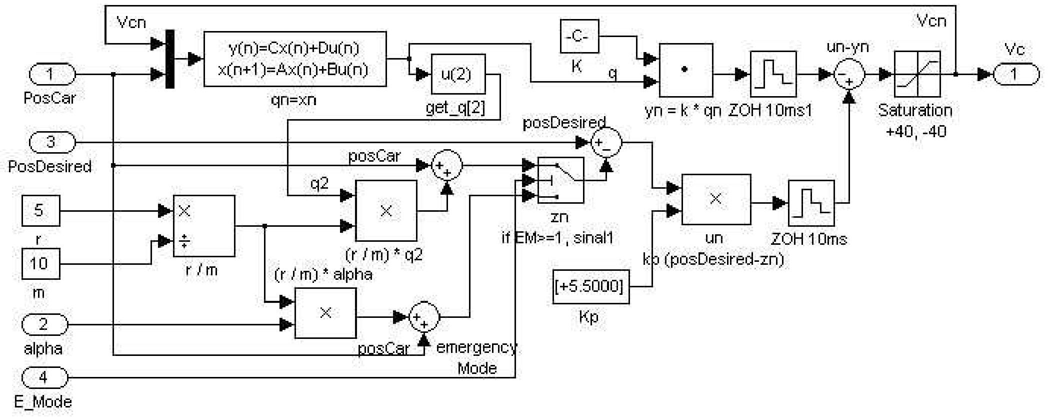


Figure 2.5: Control algorithm model in Simulink

Finally, the Plant module contains the specification of the physical plant (car and load) to be controlled. Although this module is not part of the system functional specification, it must be described in order to allow the simulation of the system behavior. For describing the continuous behavior of the plant, linear equations were represented by Simulink components such as integrators, adders, and gains. This highlights one important aspect of the FB approach, which is the possibility of reusing pre-defined FBs.

Once the modeling phase is completed, the simulation is performed to provide the validation of the FB model. Afterwards, the application code can be generated. Simulink allow the generation of C code for the corresponding FBs and the generated code can be executed in real time within the framework provided by the tool. However, reasonable effort must be performed to allow running this code in a target environment that is different from the development one.

2.3.5 UML model

Differently from the previous model, UML allows designers to represent the system's needs or functionalities before their implementation. This can be performed by means of the Use Case Diagram, where actors represent the external elements that interact with the system (I/O device or human user) and each use case represents a specific functionality that must be provided. The Use Case Diagram for the crane system is presented in Figure 2.6. Each use case also includes a textual specification to detail its related responsibility.

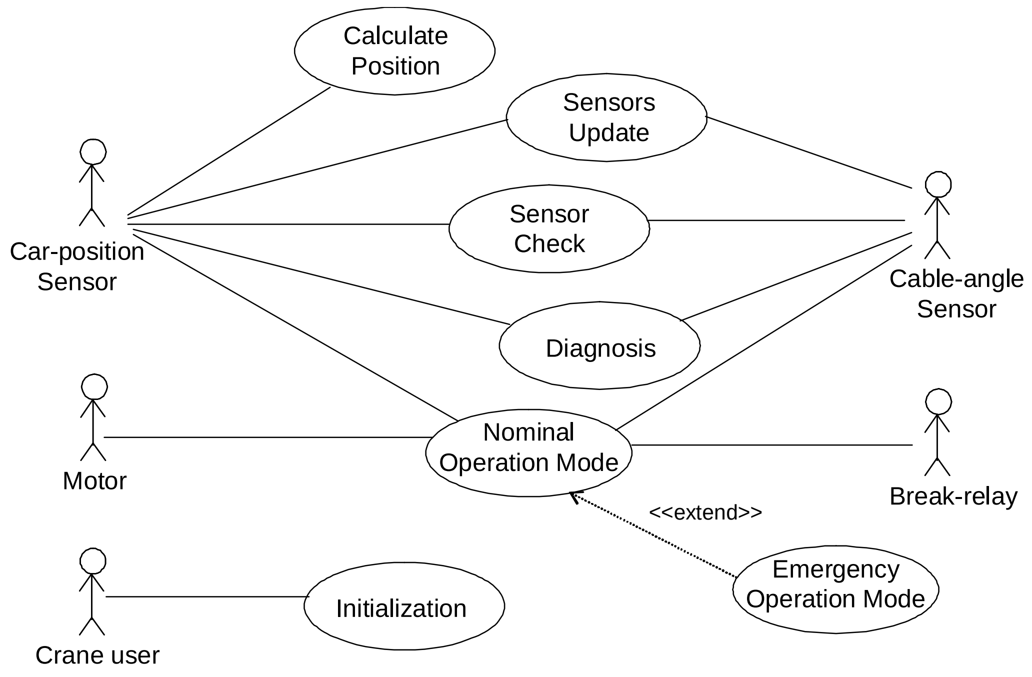


Figure 2.6: UML Use Case Diagram of the Crane system

For a better structuring of the model development, we followed the design phases proposed by Gomaa (2000) in the COMET/UML methodology. However, any other UML based design methodology that considers real time aspects could be used. Moreover, in this case study, UML1.3 was used, because UML2 was not yet available at the time.

To describe the interaction among objects that participate in each use case, they are further detailed using UML collaboration diagrams. This is part of the so called analysis modeling, which precedes the definition of requirements. Instead of collaboration diagrams, sequence diagrams also could be used. To highlight important characteristics of the modeled system (mainly timing restrictions), the UML profile for Schedulability, Performance, and Time (SPT) (OMG, 2003) is used. This profile is also usually referred to RT-UML, and is composed mostly by stereotypes and its related tags. Using this profile, a timer event for example is decorated with the stereotype <<SAtrigger>>. It includes information about its triggering frequency, as presented in the collaboration diagram from Figure 2.7 (see event num. 3 – *run()*). Such information is represented by the tag *RTat* of the stereotype that, in this case, means a periodic event with a 10 ms period.

Operations depicted in the diagram of Figure 2.7 represent the 'ControlAlgorithm' block from the FB model (see Figure 2.5) and, partially, the 'JobControl' one. Detailing the collaboration diagram, one can see three different operations sequences, denoted by the numbers 1, 2, and 3. Special attention is given to the third sequence, the control operation, which represents a periodic activity. Timing restrictions are denoted by the elements from the UML-SPT profile. Similarly to the FB model, the 'Controller' class also has an associated state diagram, which is presented in Figure 2.8. This is part of the system dynamic model, which represents the application behavior. One observed missing feature of UML is the lack of semantics to express the control algorithm itself, including its continuous-time characteristics.

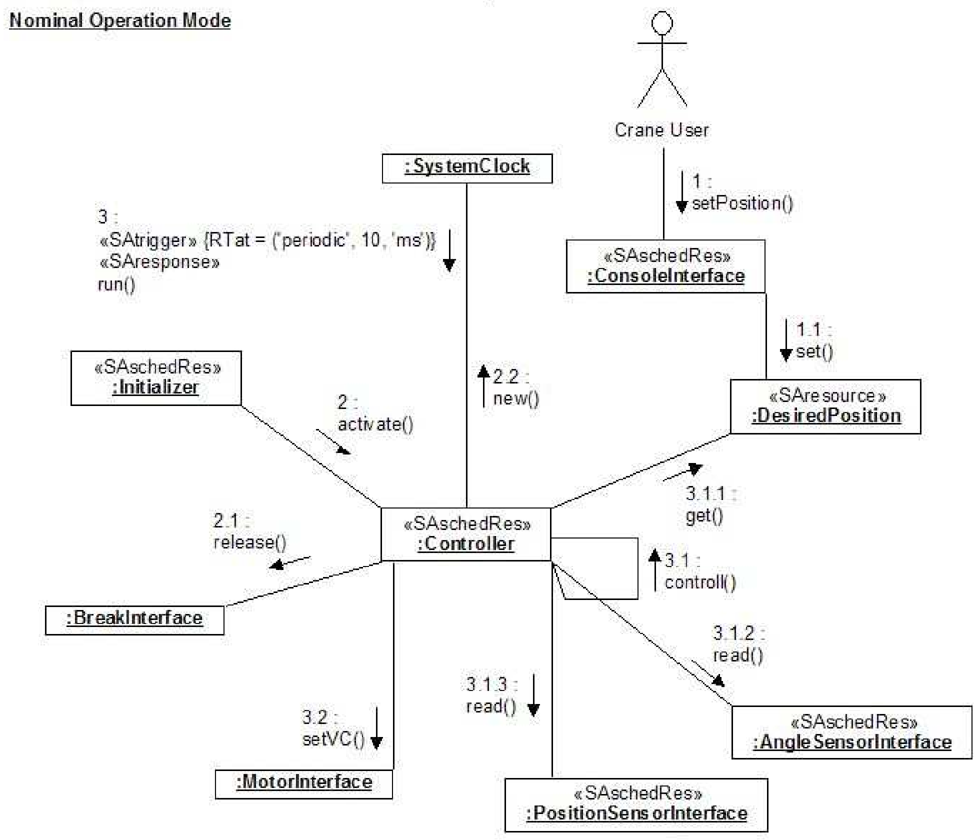


Figure 2.7: UML Collaboration Diagram of the Control Algorithm

The complete UML model of the Crane system includes 9 different collaboration diagrams. All classes from these diagrams constitute the system static structure, which is used as input for the next development step from the COMET methodology, that is known as Design Modeling. This phase is responsible for defining the architecture of the system, including the division of responsibility between client and server objects.

Since the Crane model makes use of decentralized control, it was necessary to classify objects as being passive or active. The former represents data repository elements, while the latter represents elements with their own thread of control that are capable of triggering an interaction sequence. The final result is represented by the class diagram depicted in Figure 2.9. Classes names are preceded by '::' to follow UML conventions. They can also contain a stereotype incoming from the UML-SPT profile (e.g. <<SAschedRes>>, which denotes a concurrent element in the system). The choice for the use of classes instead of capsules (part of UML 2.0) is due to the available runtime structure on which object communication is event based and does not use the port abstraction. This diagram is used as basis for the embedded system code generation.

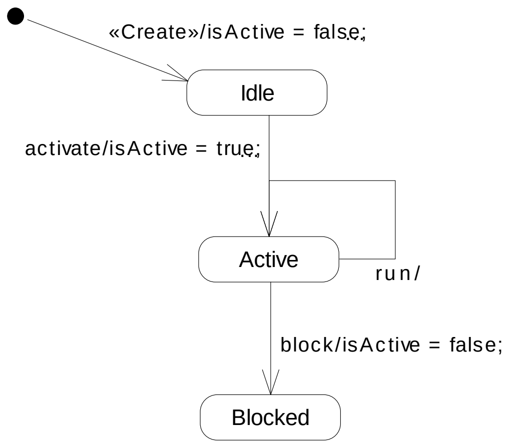


Figure 2.8: State Diagram of the Controller class

As the design tool used to build the UML model did not include a simulation module, the next step was the code generation for the system. Although other programming languages like C++ could also be used for code generation, the Java language was chosen as target in this study due to the current tool set used by our methodology (see BRISOLARA, 2005b). Details on the generated code will now be approached.

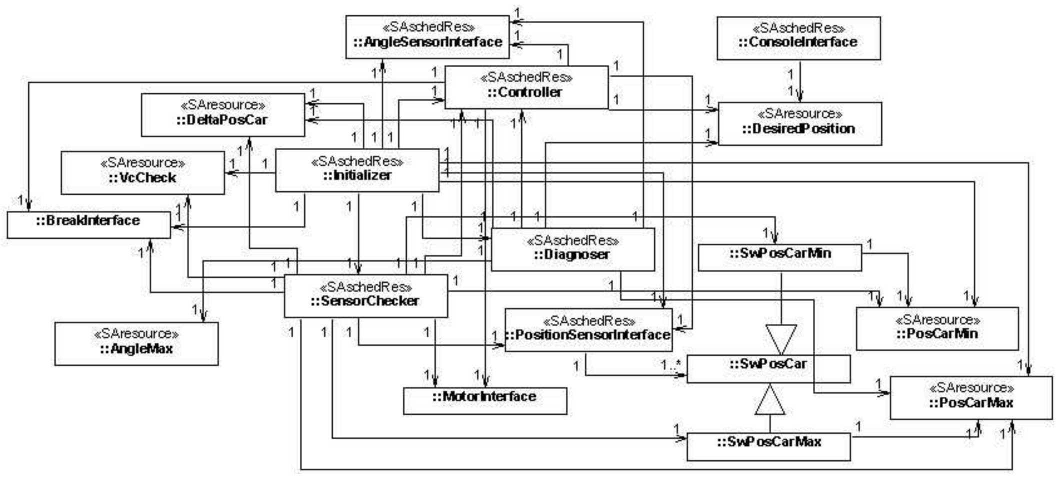

«SAschedRes» ::AngleSensorInterface
«SAschedRes» ::ConsoleInterface
«SAschedRes» ::Controller
«SAresource» ::DeltaPosCar
«SAresource» ::DesiredPosition
«SAresource» ::VcCheck
«SAschedRes» ::Initializer
::BreakInterface
«SAschedRes» ::Diagnoser
::SwPosCarMin
«SAschedRes» ::SensorChecker
«SAschedRes» ::PositionSensorInterface
«SAresource» ::PosCarMin
«SAresource» ::AngleMax
::SwPosCar
«SAresource» ::PosCarMax
::MotorInterface
::SwPosCarMax

Figure 2.9: UML Class Diagram of the Crane system

The Controller class, on which the associated stereotype denotes a concurrent real time task in the system, is selected to illustrate the generated code. This task is triggered periodically every 10 ms, with a deadline of 10 ms (see the collaboration diagram presented in Figure 2.7). To implement such features, the Controller class needs to inherit features from *RealtimeThread*, as shown in Figure 2.10. Moreover, it must define release parameters to implement the modeled timing constraints. Therefore, the class *PeriodicParameters* is used, and its instance is passed as parameter for the superclass constructor. A *RelativeTime* object is used to represent the 10 milliseconds from the task period and deadline. All these special classes are derived from an API proposed by Wehrmeister (2004).

```
import saito.sashimi.realtime.*;
public class Controller extends RealtimeThread {
   private static RelativeTime _10_ms =new RelativeTime(0,10,0);
   private static PeriodicParameters schedParams = new PeriodicParameters(
                        null,   // start time
                        null,   // end time
                        _10_ms, // period
                        null,   // cost
                        _10_ms);// deadline
    public Controller() {
      super("Controller", null, schedParams);
      // do other initializations
    }
    public void mainTask() {
      Crane.breakInterface.release();
      // periodic loop
      while(isRunning == true){
         this.controll();
         Crane.monitorInterface.setVC(m_vc);
         this.waitForNextPeriod();
       }
   }
   private int controll() { ... }
   public void exceptionTask() {
      // handle deadline missing
   }
};
```

Figure 2.10: Generated code for the Controller class

The Controller class provides two important methods: *mainTask()* and *exceptionTask()*. The former represents the task body, that is, the code executed when the task is activated. Since this task is periodic, there must be a loop which denotes the periodic execution. The loop execution frequency is controlled by calling the *waitForNextPeriod()* operation. This operation uses the task release parameters to interact with the scheduler and control the correct execution of the operation. The *exceptionTask()* operation represents the exception handling code that is triggered in case of a deadline miss, that is, if the *mainTask()* operation does not finish until the established deadline.

After the code generation process, the application was ported to the FemtoJava environment using the SASHIMI tool (ITO, 2001), which generates both a VHDL description for a dedicated Java processor and the respective program memory code (application code).

2.3.6 Evaluation criteria

In order to perform a comparison between the modeling approaches, several evaluation criteria have been identified. These criteria are based on the work conducted by Ardis (ARDIS et al., 1996), which performs a qualitative comparison among several design languages for reactive systems. Such work is extended here in the direction of searching for aspects that could be used to perform a quantitative evaluation of the designed models. Moreover, a new organization for the set of criteria is established. They are organized in groups that reflect the needs observed in the section 2.1, as can be observed in Table 2.2. The groups are further refined in subgroups to compose the evaluation criteria elements. In Table 2.3, each evaluation criterion is detailed, together with an explanation on how it is evaluated (in qualitative or quantitative terms).

Table 2.2: Evaluation criteria

Evaluation Criterion	Description
a) Requirements Specification	criteria to evaluate the capability to express and document user needs and system requirements.
b) Functional Specification	criteria to evaluate the model abstraction level and expressiveness, i.e. if it describes the problem domain elements and the system behavior/functionality in a natural and straightforward manner.
c) Validation or Simulation	criteria to evaluate if the specification can be validated before its implementation.
d) Implementability	criteria to evaluate if the specification can be easily refined or translated into an implementation that is compatible with the rest of the system.

Table 2.3: Evaluation criteria – subgroups

Criteria	Description	Evaluation	Expressed by
a1) Functional requirements	Capability of expressing and documenting the desired system functionality, together with the problem domain elements that interact with the system	Quantitative	the number (nbr) of modeling diagrams that can be used to implement the desired feature
a2) QoS requirements:	Capability of expressing the application QoS requirements and/or restrictions	Quantitative	The number (nbr) of QoS requirements that can be specified
b1) Applicability	Capability of representing system behavior or functionality by using different MoCs, according to systems nature	Quantitative	the nbr of supported MoCs
b2) Maintainability	Easiness to make modifications in the specification, e.g. addition of new elements and changes in the external elements like sensors	Qualitative	-
b3) Modularity and Hierarchy	Capability of dividing a large specification into independent modules, which could be again decomposed into even smaller parts	Qualitative	-
b4) Expressiveness	Capability of the modeling language primitives to describe the specification	Quantitative	b4.1) nbr of modeling primitives b4.2) nbr of different modeling primitives b4.3) nbr of handed lines of code
c1) Simulation	Capability of verifying if the specification can be used to validate the implementation	Qualitative	-
c2) Verifiability	Capability of demonstrating formally that the specification or generated program meets the requirements	Qualitative	-
d1) Code generation	Capability of generating an executable code from the model	Qualitative	-

2.3.7 Comparison results

This section presents an analysis and comparison of the UML and FB models according to the criteria discussed in the previous section. The results are summarized in Table 2.4. For evaluating the qualitative aspects, we have used the symbol “++” to indicate a particular strength of the approach, “+” to indicate that the model meets the criterion in a way that is adequate, but less than ideal, and “0” to indicate a clear weakness of the model.

Table 2.4: Comparison results

Evaluation criteria	FB	UML
a) Requirements Specification		
a1) Functional requirements	0	1
a2) QoS requirements	0	2
b) Functional Specification		
b1) Applicability	3	1
b2) Maintainability	+	++
b3) Modularity	++	++
b4.1) Number of used modeling primitive	111	184
b4.2) Number of different modeling primitive in use	5	5
b4.3) Number of line codes written by the designer	0	96
c) Validation / Simulation		
c1) Simulation	++	+
c2) Verification	0	0
d) Implementability		
d1) Code Generation	++	+

Source: BRISOLARA, 2005b, p. 33

This evaluation begins by analyzing the facilities for expressing the system functional requirements. UML offers the facilities provided by the use case diagram (1 point), where functional requirements are defined in terms of actors and use cases. On the other side, the FB approach does not support this kind of resource (0 points).

More recently, OMG proposes a new visual language called Systems Modeling Language (SysML) that reuses a subset of UML 2.0 and extends the language to satisfy the requirements of the UML for Systems Engineering (SE) domain. SysML provides two new notations to aid the requirements specification, which are Requirements diagrams and Parametric diagrams. Requirement diagrams can capture functional, performance and interface requirements, whereas with UML you are subject to the limitations of Use Case diagrams to define high-level functional requirements. Likewise, Parametric diagrams can be used to specify performance and reliability requirements during system analysis.

Regarding the support for QoS specification, one can see that the UML-SPT profile supports both timing and performance requirements specification (2 points), while in the FB approach there is no support for such issues (0 points). In the FB model, the timing requirements are implicit in the functional/behavior specification. Neither language gives support to the specification of power consumption and cost requirements.

Recently, the definitions from UML-SPT and the QoS&FT (UML profile for modeling Quality of Service and Fault Tolerance Characteristics and Mechanism) profiles have been used to define a new profile called MARTE (Modeling and Analysis of Real-time and Embedded Systems). MARTE provides a complete set of modeling elements to build specification and design models of embedded systems, and supports the various (asynchronous and synchronous) computation models used in the RT domain.

Analyzing the model applicability by means of the number of supported MoCs, it is possible to observe the advantages provided by the FB approach, as it supports three different MoCs (3 points): continuous-time (analog), discrete-time (digital), and event-based. Regarding UML, it supports only the event-based model (1 point). In spite of this, there are efforts described in literature that already address the lack of a dataflow model in UML (BICHLER, 2004)(CHEN, 2004).

Bichler (2004) proposes the D-UML, which integrates dataflow equations to the UML/Realtime modeling language. A comparison between UML, FB and the D-UML can be found in (BRISOLARA, 2005a). D-UML uses structure diagrams composed of UML2 capsules and flows connected by ports. In this approach, a statechart is developed for each capsule. Although D-UML allows model dataflow, this abstraction is in fact implemented using send/receive mechanisms, which are controlflow-like.

Using UML2 notations, activities diagrams can be used to define dataflow systems. More recently, in the SysML definition, the activity diagram is extended to support the traditional Systems Engineering functional block diagrams (dataflow) and continuous behaviors. However, activity diagrams are more closed to flowcharts than the dataflow proposed by FB models. Moreover, the commercial tools have just started to support these new features, so that it can not be used during this case study.

Regarding maintainability, the intrinsic OO properties from UML models, like the specialization/generalization facilities (inheritance), provide better maintainability if compared to the structured approach of FB models.

Considering modularity and hierarchy aspects, it is possible to conclude that the FB model leads to a slight better decomposition. This can be observed by comparing the Simulink high level model against the UML class diagram. The first one contains fewer elements, making the interpretation of the physical behavior easier. The UML class diagram used in our model maintains the whole system elements within the same abstraction level, which is somewhat not suitable, considering the desired hierarchical features. However, the addition of the composite structure diagram in UML 2.0 overcomes this problem, since it allows for decomposition in a natural and straightforward manner.

The next criteria concern model expressiveness: number of used modeling primitives vs. number of different modeling primitives in use. The FB model contains 111 modeling primitives, excepting the plant module, including Simulink components, connections, ports, states, and transitions. In the UML model, 184 primitives are used. Regarding different modeling primitives in use, the UML model is represented by means of classes, objects, associations, states, and transitions. Therefore, it is natural to observe an equivalent number of different modeling primitives if compared to the FB model, which includes blocks, ports, connections, states, and transitions. Nevertheless, using a design tool like Simulink, the designer can make use of different pre-defined components available in a component library.

Another relevant issue relates to the number of lines of code programmed by the designer in each model. It can be observed that in the UML model the designer has to manually write 96 lines of code, while in FB model the program code was completely generated by the tool. Several UML tools have code generation capabilities, but they generate only code skeletons for classes and, at most, code from the statecharts. The hand written code parts include mainly the methods' behaviors that cannot be captured from the model. On the other hand, by using the FB model and associated library, the designer is not required to code the program by him/herself, as observed in our case study. Lastly, our experimental results show that by using a component library within the UML model reduction on the number of hand written code from 96 to 66 lines can be achieved.

Regarding model validation and simulation, it is possible to observe that, in order to provide such features, suitable modeling and design tools are required. In the crane case study, only the FB model could be simulated, thanks to the Simulink tool that provides a simulation engine. The available version of the Real-time Studio tool, used for the construction of the UML model, does not support model simulation. However, considering the authors' experience with other UML-like modeling tools, they provide support at most for animation of statecharts (event based MoC). Consequently, one can state that for this task the FB model is more adequate, because the simulation environment supports all the three intrinsic MoCs.

Analyzing the verification features, neither UML nor FB approach have support of formal verification of complete models. In UML, some tools allow for model checking in specific diagrams, like Statecharts and Sequence Diagrams. Moreover, many tools support consistency checking between diagrams, for instance checking the connections between the components in a FB diagram or even guaranteeing that an operation called in an UML collaboration diagram exists in the related class. For this reason, both languages are considered weak in this aspect. Besides that, UML commercial tools check the syntax of actions in the statecharts. They also check if an operation called in a collaboration diagram was defined in the class. Therefore, Damm and Harel (2001) proposed the Live Sequence Charts (LSC) that are an extension of Message Sequence Charts (MSC) with rigorous semantic. The use of the LSCs allows consistency check between the generated scenarios and the sequence charts applying formal verification techniques.

Finally, considering the model implementability, one can see that from both models an architecture independent specification can be derived. Still, there are two aspects that lead to distinct capabilities: amount of code provided by designer and number of pre-defined components. In UML, the need for designer intervention is higher as can be observed in the crane case study, because some parts of specification cannot be expressed using UML diagrams (e.g. control algorithm). In the FB models, the whole code can be generated automatically, since it relies on the use of pre-defined libraries. However, the generated code requires several modifications/optimizations to be executed outside the framework provided by Simulink.

3 UML-BASED EMBEDDED SOFTWARE GENERATION

The Unified Modeling Language (UML) (OMG, 1999) is a standard notation for modeling and documentation of object-oriented software. The intention behind the definition of the language was to consolidate the various OO languages, methods, and notation in a single modeling language independent of vendor. UML was based on the OMT method of Rumbaugh (1991), the Booch method (BOOCH, 1981), and the OOSE (Jacobson, 1992). This language was defined to support specification, visualization, construction, and documentation of conventional computational systems.

The UML language is in constant evolution and OMG is responsible for maintaining and reviewing it. The organization can have the assistance from the members from both academy and industry. All members can propose new features and vote for new solutions for the UML language. Nowadays, UML is considered the de facto modeling language for software systems. Several tools based on UML are available for software modeling and code generation. The widely use of UML as a standard language also contributed for the definition of software development approach that shifts the focus from code to models, which is called model-driven development (SELIC, 2006). MDD aims to make models the primary resource in all aspects of software engineering and provide benefits of cost reduction and quality improvement.

UML1, the first version of the language, presented some limitations, mainly regarding to the low precision and lack of formal semantic. That left ambiguities in the model, allowing different interpretations and difficulting the implementation of tools for model capturing and code generation. To automate these steps, some vendors defined more precise semantics, but the problem was that these semantics varied from vendor to vendor. Recently, a major revision of UML was coordinate by OMG and the new version of the language (UML2) was defined, with enhanced semantic and more precision. The main objective of this revision is to eliminate the ambiguities, facilitating the design automation by tools.

The first minor revision of the original UML 2 specification has resulted in UML 2.1.1 (OMG, 2007a). Although this revision adds fixes to the abstract syntax to eliminate minor inconsistencies and ambiguities, existing UML-tools still have limited generate code capabilities. This is better discussed in the section 3.1, where the capabilities of the existing UML-based tools for embedded software generation are depicted. From that analysis, one can observe a gap between model and code, presented in section 3.2 using experiments, explained in section 3.3.

3.1 Existing approaches for code generation from UML models

To support SW automation based on UML models, the first step is the capture of the model. An UML model is an instance of a class model called UML meta-model. Models are stored using the XMI (OMG, 2002) model interchange standard, which is based on XML (*eXtensible Markup Language*) (GROSE, 2002). The problem is that different versions of the XMI are used for different vendors, making difficult the interchange between UML-based tools. Model repositories can be used to store an UML model represented using XMI, providing functions to create, add, remove and update a model, and thus, facilitating the tool implementation. Examples of available UML repositories are Eclipse EMF (ECLIPSE DEVELOPMENT TEAM, 2006), Netbeans Metadata Repository (MDR) (NETBEANS DEVELOPMENT TIME, 2005), and System Modeling Workbench (SMW)(PORRES, 2003).

After capturing the model, this must be transformed into code in the target language. This process typically uses a template engine to transform model into code, given the format specified by the template. Templates are a flexible approach to convert models to text. The most popular template engines are Velocity (APACHE SOFTWARE, 2005) and JET (Java Emitter Templates) (ECLIPSE DEVELOPMENT TEAM, 2005). Figure 3.1 presents an example of template in Velocity, which is an open-source project created to generate HTML code. Details about code generation using templates can be found in (BOAS, 2004). Model repositories, such as EMF and MDR, and templates can be used together in the implementation of a code generator.

Other methods to generate source code include the use of rules, writing programs that generate programs (code generators), and using transformations such as XSLT (TIDWELL, 2001). Indeed, XSLT is popular for XML transformations, but it is too verbose to be an effective language for model-driven code generation. More recently, as model-driven development approaches have gained interest, an alternative approach for code generation based on model transformations has been proposed. In this way, the model described in a higher abstraction is transformed into another one closer to the final implementation. More than one transformation steps could be applied to the initial model, including optimization steps. Finally, a simple conversion from model to text can be applied to produce code in the target language.

```
Public class $class.name{
#foreach ($att in $class.allAttributes)
#set ($javaType = $att($att.type.name))
// $att.name
private $javaType $att.Name; public $javaType
get$toUppercase($att.name) ( ) {
    return this.$att.name;
}
public void set$toUpperCase
        ($att.name) ($javaType $att.name) {
    this.$att.name = $att.name;
}
#end
#foreach ($att in $class.allOperations)
...
```

Figure 3.1: Velocity template example

According to Björklund (2004), templates are difficult to create and manage. They cannot be the only mechanism for code generation, mainly because code optimizations cannot be applied using templates. The author suggests that the most adequate way to generate code from a model is through model transformation. An example of the code generation from UML class diagrams to Java was presented in (BJÖRKLUND, 2005), and is illustrated in Figure 3.2. This figure shows three ways to generate Java code from a class diagram. A simpler one is just directly converting the model to text, as the model is a simple one, just a partial code could be generated with this approach. In an alternative way, the model could be transformed in another more detailed model, and then use this model to generate code. In addition, a complex mapping could be used to generate the code from the initial model. This example is good to illustrate the code generation idea. It is though too simple because the transformations show just the use of the JavaBeans convention, which defines that for each class attribute, methods *set* and *get* should be defined to give access to this attribute.

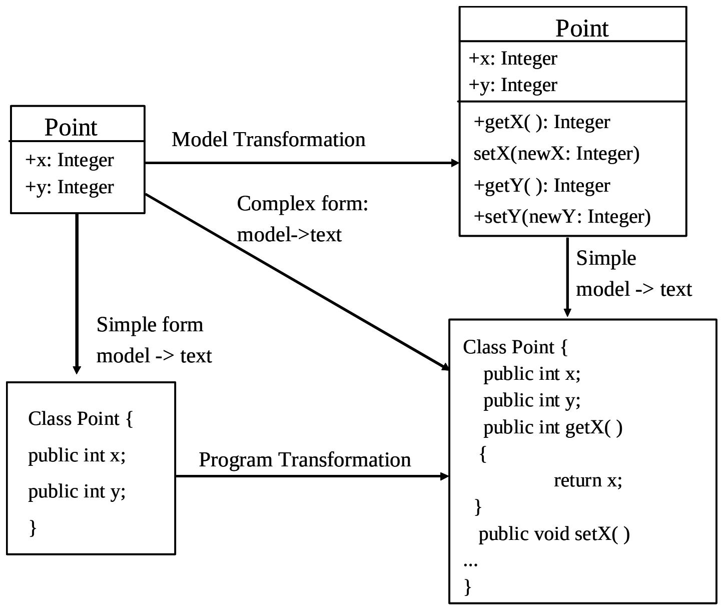


Figure 3.2: Approach for code generation (BJÖRKLUND, 2005)

3.1.1 Code generation: existing tools

The tools proposed for code generation from UML models can be divided in two classes, structural and behavioral. This division was initially proposed by Björklund (2005). In a structural code generation, only structural diagrams are used, i.e. class diagrams, where classes have attributes and relations. The tools that follow this approach generate only skeleton of code, and the strategy is available since the first UML tools. For example, they can map all constructions (elements) in a class diagram to Java or C++ programs. On the other hand, the behavioral code generation is based on behavioral UML diagrams, such as state, sequence and collaboration diagrams. Most of the available tools provide code generation only from UML state diagrams, as for example, Artisan Studio, Rhapsody, UniMod and BridgePoint UML Suite.

To be able to generate complete code from UML diagrams, designers are asked to add information to the model, e.g. specifying the action correspondent to state (activity) in a state (activity) diagrams or specifying the method behavior in sequence diagrams. Some code generators use the target implementation language to describe these methods and actions, which turn the model not independent of the target language. Other tools use actions languages to complement the state and activity diagrams in order to generate complete code. However, as the Actions Semantics proposed for UML 1.5 defines only an abstract syntax, tool vendors use proprietary action languages. Such approach is used in iUML (KENNEDY CARTER, 2005), BridgePoint UML Suite (MENTOR GRAPHICS, 2005), and Telelogic Tau Architect/Developer (TELELOGIC, 2004). As an example, BridgePoint uses the Object Action Language (OAL) and provides full code generation, in which the designer uses state diagrams to represent the system behavior and specifies actions correspondent to all states using OAL.

Other common approach to bridge the gap between model and implementation is the use of intermediate languages. Such approach is used by Björklund (2004) and Hubbers and Oostdijk (2003).

In order to support model verification, simulation, and synthesis, Björklund (2004) proposes the use of Rialto as the intermediate language during the model design. This is illustrated in Figure 3.3. This language has a formal semantics that allows the capture of the semantics in UML behavioral diagrams. Thereby, the language can be used as an execution engine for UML models and to generate code too. Rialto can also be used to combine multiple MoCs because different scheduling policies are defined in this language. In this work, the authors consider that the activities diagrams have dataflow as their underlying model of computation and these diagrams can be interpreted as a statechart. In those, all computation is performed in state activities and the transitions are triggered by completion events. However, a statechart is control flow like and is not the more adequate representation for dataflow models. Moreover, as this is an ongoing work, it supports only some UML diagrams.

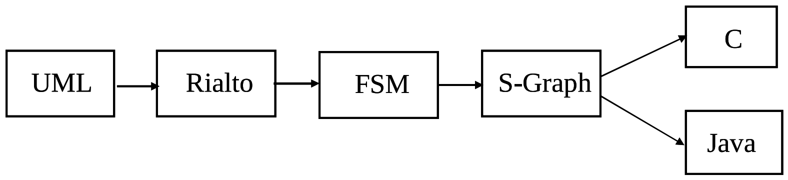


Figure 3.3: Rialto-based code generation approach (BJÖRKLUND, 2004)

Hubbers and Oostdijk (2003) highlight the difficulty of verifying if the implementation behavior agrees with the specification. In this context, the authors propose the use of JML (Java modeling Language) specifications in order to facilitate this verification. A JML specification allows formal verification to check if the generated code implements the specified model. In this project, a tool called AutoJML has been developed, which automatically derives JML specifications from UML state diagrams represented in the XMI format, beyond the Java code. The combination of the JML specification and the skeleton code can be formally verified using the ESC/Java (FLANAGAN, 2002).

UML2 provides some constructions that aid the modeling of the complete execution flow, as for example the *ref* operator that allows to link fragments in different sequence diagrams. This new version of UML also provides the operators *alt*, *opt* and *loop*, which permits representing conditions and loops in sequence diagrams. These new constructions allow the proposal of code generation approaches based on sequence diagrams, as in (BABU, 2005) and (REICHMANN, 2004). Matilda (BABU, 2005) is a model-driven development platform that accepts the UML2.0 class and sequence diagrams as input. This platform provides capabilities for model checking against the

UML meta-models for syntax and semantic correctness, besides code generation. In this approach, UML models are mapped to the abstract syntax tree from which the code is generated. Java constructions are used on the sequence diagrams and thus, full code can be generated from the model. Reichmann (2004) proposes a code generator, which uses Velocity engine (APACHE SOFTWARE, 2005) to generate Java or C++ code from UML models, as class and sequence diagrams. In this work, in order to complete the behavioral diagrams, the language called MeDeLa is used to specify methods behavior. MeDeLa is based on Java syntax and consequently, the use of this language does not provide a higher abstraction level than that of Java or C++.

3.2 Analyzing the gap between UML model and a Java program

The diagrams and graphical notations provided by UML help the designer specify the behavior of complex systems, without demanding the definition of details usually required by the programming languages. In a desired flow, a model compiler (or code generator) must be able to generate these details from the model, producing an implementation in the target programming language.

A code generator could be considered as a function that maps artifacts from a modeling language to lines of code in a programming language. To design a code generator, the definitions of the models to be supported as input are required, as well as the used features of the target programming language and the mapping between both. In this section, we study the mapping from UML models to Java code and we discuss the gap existing between both specifications.

As highlighted by Erikssom (2004), when Java is the target language, a natural progression from the logic classes to code components is possible. As Java and UML are object-oriented languages, some UML constructions can be directly mapped to Java ones. For example, a class in UML is translated to a class definition in a Java code and, for each class defined in a class diagram, a *.java* file is created. Table 3.1 shows basic rules to map UML constructions to statements in Java.

Table 3.1: Mapping UML to Java

UML Constructions	**Java Constructions**
Attribute	instance variable
Operation	*Method*
Abstract Class	abstract *class*
Interface	*interface* key-word
Package	package declaration (*package*)
Subclass/Generalization	*extends* key-word
Realization	*implements* key-word
Dependency, <<*uses*>>	*Import*
Multiplicity	*Array*
Role	Instance variable from the type of the class associated with the role

Source: MADISETTI, 2005.

Besides the mapping proposed by Madisetti (2005) and summarized in the Table 3.1, other simple mapping rules can be defined, as for example, the generation of *get* and *set* methods for all class private attributes. In this way, the single way to access these attributes is through these methods. That is a convention in JavaBeans.

However, the direct mapping is only possible from structural diagrams. The UML behavioral diagrams include many concepts, such as actions, events, and states, which are not present in most programming languages. This means that there is not a one-to-one mapping between behavioral diagrams and its implementation (BJÖRKLUND, 2003).

3.2.1 Experiments

In order to discuss the gap between the UML model and an implementation on a target language, we analyzed two embedded applications developed in Java. Those are the Crane control, which is also used as case study in chapter 2, and the Address Book, which includes calendar, alarm, and calculator. In this experiment, firstly, we analyze both applications regarding the number of lines of code that can be automatically generated from the model using structural code generators. That means using only class diagrams. For these lines, we use the term "Automatic Generated" (AG). The rest of the lines were classified as "Written by designer" (WD). After that, these WD lines were classified according to the operation that they are evolved or the behavior that they describe.

To determine the lines marked as "Automatic Generated" (AG), the mapping presented in Table 3.1 was used. In this way, lines responsible to define classes, interfaces, methods and attributes are considered in this group. Besides that, lines of code used to define the use of an API and the definition of packages are also classified as AG. In addition, lines of code responsible to initialize attributes and define *get* and *set* methods for all class attributes are also considered as AG in this analysis.

This study aims to define an abstraction that could be used to complement the UML diagrams in order to obtain the complete code generation from the model without losing abstraction. Then, after the identification of the AG lines of code, the remaining lines are considered WD. The WD ones are then analyzed in more detail in order to evaluate how they could be specified in a higher abstraction level and, consequently, automatically generated. To do this, we classified these WD lines of code in 20 groups, as presented in Table 3.2. Firstly, seven simple groups were defined (e.g. <co>, <cm>, <mat>), which were combined to define eleven complex groups (e.g. <if+mat>, <for+cm>, <for+ds+mat>). The <co> and <io> groups are reserved for the lines of code used to dynamically create and initialize objects, respectively. The <cm> and <ret> represent method invocation and method return, respectively. The groups <dv>, <iv> and <incv> are used to represent the declaration, initialization and increment of variable, respectively, as well as the group <im> represents the lines used to initialize matrix or vectors. Finally, math operations are classified as <mat>.

In the proposed classification, control structures like conditional and loops were divided in several groups according to the correspondent control operation, e.g. the group <for+cm> represents a loop with known number of iterations and with method calls inside. For conditionals, similar classification was proposed for the command *If*, defining the group <if+cm>. Besides *For* and *If*, similar structures like *Switch/Case* and *While* were also considered and classified, as well the *Try/Catch* used for exception handling in Java code. For example, the *While* was classified as <loop+cm>, which means a loop with conditional and that have method call inside. In addition, <for+ds> classified the loop used to manipulate a data structure (matrix or vector). An example of these lines of code is illustrated in Figure 3.6(a).

Table 3.2: WD lines of code classification

Classification	**Description**
<co>	Create objects - Lines of code used to create an object. Ex: classA obj = new classA();
<io>	Initialization of object (object is already allocated, only will be updated). Ex: currentObj = objA;
<cm>	Call methods - Lines of code used to indicate method invocation.
<ret>	Lines of code to represent a method return.
<dv>	Lines of code used to declare an auxiliary variable. Ex: int temp;
<iv>	Lines of code dedicated to give values for variables. Ex: a = 4647;
<incv>	Lines of code used to increment variable value. Ex: a=a+1;
<im>	Lines of code used to initialize matrix and vectors. Ex: X[0]=1;
<mat>	Math operations. Ex: num = u - y; Ex: num = sqr (a);
<if+cm>	Conditional with method call. Ex: if (test) method( );
<if+incv>	Conditional with a variable increment. Ex: if (test) temp+=2;
<if+iv>	Conditional with initialization of variable. Ex: if (EmergencyStop) vc=0;
<if+mat>	Conditional with math operations. Ex: if (num >= max) v = max – sqrt (num); else v = sqrt (num) + min;
<if+mat+ds>	Conditional with data structure manipulation and math operations. Ex: if (a) z = posCar*q [1];
<for+cm>	Loops with method invocations. Ex: for (int j = 0; j < max; j++) { init(j); .... }
<for+ds>	Loops used to manipulate a data structure. Ex: for (int i = 0; i < 5; i++) { q[i]= q1[i]; }
<for+ds+mat>	Loops with math operations under data stored in matrix or vectors.
<for+ds+cm>	Loops with method calls used to define contents for a data structure.
<loop+cm>	Loops with conditional test, in which methods are invoked. Ex: while (test) { }
<switch-case>	Lines of code used to define a switch-case conditional structure.
<try/catch>	Lines of code used to exception treatment.

Figure 3.4 (a) shows a block of Java code, which performs the copy of the content from a vector to another vector. According to the classification presented in Table 3.2, it was classified as <for+ds>. The same behavior can be described in a single code line in Python and Matlab, as shown in Figure 3.4 (b). This simplification is possible because the interpreters and compilers provide by these tools are able to treat that. This allows, for example, that the user manipulates a matrix as a primitive type in Matlab.

```
for (int i=0; i<5; i++)
{
            q[i] = r[i];
}
```

a) Java Code

```
q=r
```

b) Python Code

Figure 3.4: Example of <for+ds> lines of code

From this analysis, we observed also lines of code that could be automatically generated using templates, e.g. the lines for exception treatment and conditional structure of type *switch-case* in Java code. These structures were classified as <try/catch> and <switch/case> in Table 3.2. Skeleton of code can be automatic generated using templates for both cases.

3.2.1.1 Crane results

The crane is used as case study in chapter 2 and the same UML model and Java implementation are used in the analysis presented here. The analysis results for the Crane control application are summarized in Table 3.3. It is important to notice that this implementation reuses a library to solve floating point operations. As the library was reused, these lines of code were not considered in this analysis.

Table 3.3: Crane analysis results

Classes/Interfaces	**Total**		**AG**		**WD**	
Crane	20	8.93%	6	30%	14	70%
Controller	77	34.38%	22	28.57%	55	71.43%
CraneInitializer	10	4.46%	6	60%	4	40%
ConsoleInterface	11	4.91%	7	643.34%	4	36.36%
BreakInterface	8	3.57%	5	62.5%	3	37.50%
AngleSensorInterface	19	8.48%	12	63.16%	7	36.84%
positionSensorInterface	19	8.48%	12	63.16%	7	36.84%
MotorInterface	6	2.68%	6	100%	0	0%
SWPosCarMin	6	2.68%	5	83.33%	1	16.67%
SWPosCarMax	6	2.68%	5	83.33%	1	16.67%
PosCarMin	6	2.68%	6	100%	0	0%
PosCarMax	6	2.68%	6	100%	0	0%
DesiredPosition	6	2.68%	6	100%	0	0%
DeltaPosCar	6	2.68%	6	100%	0	0%
VcCheck	6	2.68%	6	100%	0	0%
ParameterTimeOut	7	3.13%	5	71.43%	2	28.57%
Diagnoser	5	2.23%	5	100,00%	0	0%
TOTAL	**224**	**100%**	**126**	**56.25%**	**98**	**43.75%**

Table 3.3 presents the total number of lines of code, the number of lines of code automatic generated (AG) and written by designers (WD) for each class used in the crane implementation. From these results, we observed that more than 40% of the total lines of code of the crane were classified as WD, which means that these lines of code must be described by the designer. Moreover, the results show that for 8 classes from the 17 classes that compose the application, the number of lines of code written by designer (WD) is too small, being around 0 and 17%. This is because theses classes represent shared resources and define only attributes and methods to access its attributes. For the classes that present a larger number of WD lines of code, a detailed analysis was made and is presented in the remaining of this section.

The *Controller* class has 77 lines of code, which represents 34.38% of the total lines of code in the whole application. The main part of the application behavior, which is the control algorithm, is encapsulated in this class. A block of code from the *Controller* class is illustrated in Figure 3.5. The analysis results show that 28.57% of the lines of code of the *Controller* class can be automatically generated. The remaining 55 lines of code (71.43%) are classified as “Written by Designer” (WD) and then, they must be described by the designer. The result obtained for the Controller class is illustrated in Figure 3.6(a). From the 55 lines of code, 10 lines are used to vector initialization (<im>), 6 lines are loops to vector manipulation (<for+ds>) and 10 lines are loops where vector are manipulated through methods (<for+ds+cm>). In addition, 11 lines are conditionals and 14 are method invocations.

```
public class Controller extends RealtimeThread {
...
public Controller ()   {
     super(null, relParams);
     // do initializations (A, B, X, K, kp, q...)
   }
private int controll() {
     int posdesired = Crane.desiredPosition.get();
     poscar= Crane.positionSensorInterface.read();
     mul_Bx();
     mul_y();
     if (EmergencyMode)
          z= softfloat.floatAdd(poscar, softfloat.floatMul(0x4500, q[1]));
     else
          z= softfloat.floatAdd(poscar, softfloat.floatMul(0x4500, alfa));
     ...
     for (int i=0; i < 5; i++) {
          q[i]= q1[i];       }
    return(VC_temp);
}
public static void mul_Aq() {
for (int i= 0; i < 5; i++) {
     q1[i]= 0;}
int lin=0;
for (i=0; i < 5; i++) {
     for (int j=0; j < 5; j++)
          q1[i]= softfloat.floatAdd(q1[i], softfloat.floatMul(A[lin+j], q[j]));
     lin+=5;
     }
   }
... // more code ...
```

Figure 3.5: Example of analyzed code: Controller class

The analysis results for the *Crane* class are illustrated in Figure 3.6(b). From the 20 Java lines of code of this class, 6 can be automatically generated because they represent the class header and method declarations. From the 14 remaining lines of code, classified as WD, 8 are responsible for the object creation (<co>), 5 are method calls (<cm>), and 1 is a loop in which there is method call inside (<loop+cm>). Figure 3.7 illustrates the results for the analysis of the remaining classes.

```
Controller Class
Total: 77 (22 AG + 55 WD)
55 = 3 <iv> + 14 <cm> + 1 <ret>+ 10 <im>
      + 6 <for+ds> + 10 <for+ds+cm>
      + 6 <for+ds+cm>+ 4<if+cm> + 1 <if+iv>
```

(a)

```
Crane Class
Total: 20 (6 AG + 14 WD)
14 = 8 <co> + 5 <cm> + 1 <loop+cm >
```

(b)

Figure 3.6: Analysis results obtained of the *Controller* and *Crane* classes

```
CraneInitializer
Total: 10 (6 AG + 4 WD)
4 WD= 4  <cm>
BreakInterface
Total: 8 (5 AG + 3 WD)
3WD =3 <iv>
ConsoleInterface
Total: 11 (7 AG + 4 WD)
4 WD = 2 <cm> + 2 <co>
AngleSensorInterface
Total: 19 (12 AG +7 WD)
7WD = 2 <co> + 1 <cm> + 1<iv> +3 <for+cm>.
                    PositionSensorInterface
Total: 19 (12 AG +7 WD)
```

Figure 3.7: Analysis results for Crane classes

3.2.1.2 *Address Book results*

An Address Book application is another case study. The application allows the storage of information about contacts, such as name, address, phone, and birthday. Besides that, it checks for birthdays in the month or in a given-day. This application includes an alarm and a calculator that performs simple math operations. Figure 3.8 presents the class diagram for our Address Book. Besides the classes presented in this figure, a class called *Console* was used in the implementation to facilitate reading the values from the keyboard. Since the behavior implemented by this class could be reused from a pre-exiting class or library, this class was not considered in this analysis.

Table 3.4 presents the complete results obtained for the analysis of the lines of code for each Address Book class (total number of lines of code, number of AG and WD lines of code. The results show that 66.21% of the Address Book lines of code must be written by the designer. We observed that the AddressBook presents a better distribution of lines of code among the classes when compared to the Crane, where many classes have little number of lines of code and the algorithm behavior is encapsulated in a single class that contains 28% of all lines of code.

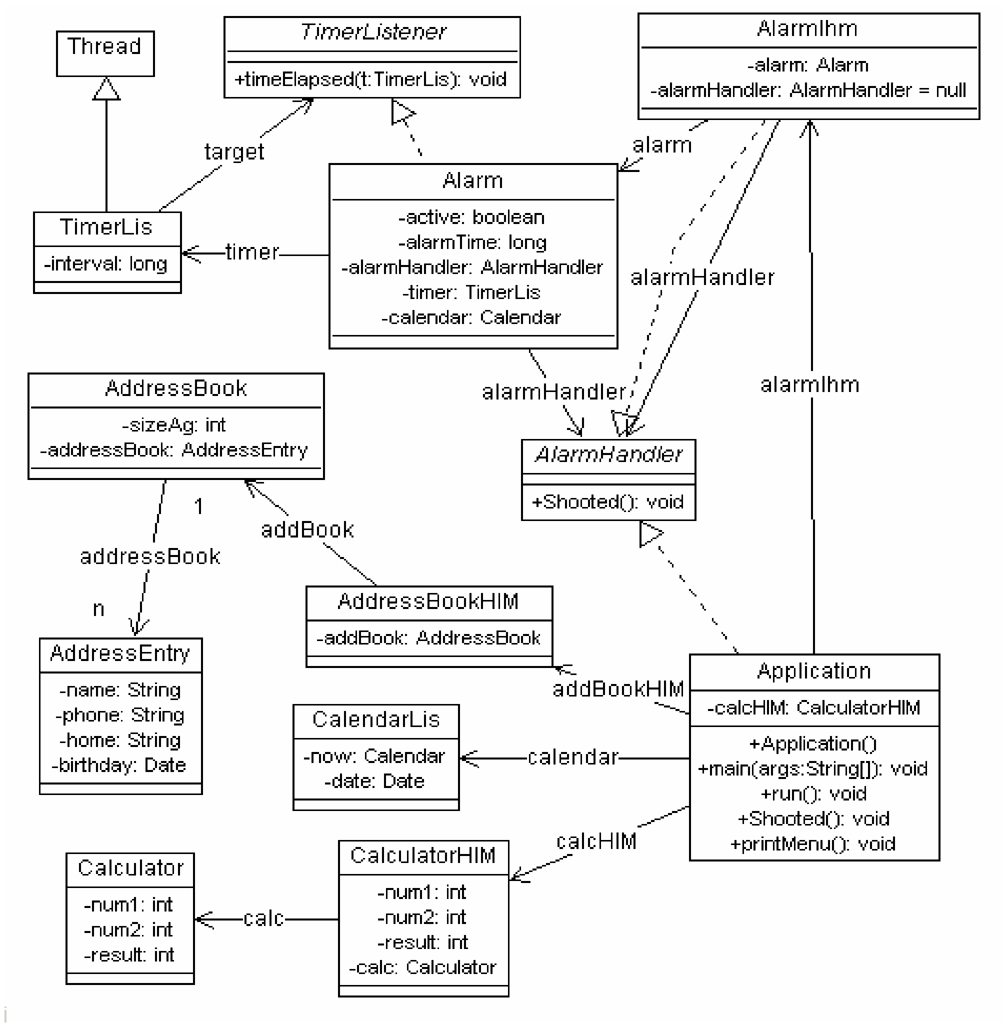


Figure 3.8: Address Book Class diagram

The results show that for simple classes, complete code can be automatically generated. The entities *AlarmHandler* and *TimeListener*, for example, are interfaces and define only the methods that must be implemented in the classes that implement these interfaces (*Alarm* and *Application* in the Address Book). For that reason, 100% of the code for them can be automatically generated. The class *AddressEntry* represents the contact of the Address Book, defining the attributes (fields) stored for each contact and the methods used to access these fields. In this way, full lines of code can be generated for this class from the class diagram.

The class *AddressBook* has 100 lines of code, in which 21 are classified as AG and 79 as WD. This class defines the data structure required to store all *AddressBook* contacts, here represented as *AddressEntry* objects, and provides methods to insert, remove, and search elements in this structure. However, depending on the used data structure, the implementation of these methods differs. As this implementation does not use the Java collection libraries, these methods were defined by the designer. The Java API has several classes to store collection of data and provides methods to add, remove, search elements for each of theses classes. The number of lines of code written by the designer could be reduced with the use of classes from the Java library. In the class *AddressBook*, the use of a pre-defined data structure, instead of a simple array, could reduce in 63% the number of lines of code written by the designer.

Table 3.4: Address Book analysis results

Classes/Interfaces	**Total**		**AG**		**WD**	
Application	49	9.68%	15	30.61%	34	69.38%
Calendar	37	7.31%	18	48.65%	19	51.35%
Calculator	17	3.36%	7	41.18%	10	58.82%
CalculatorIhm	61	12.06%	16	26.23%	45	73.77%
AddressBook	100	19.76%	21	21%	79	79%
AddressEntry	32	6.32%	32	100%	0	0%
AddressBookIhm	113	22.33%	20	17.70%	93	82.30%
Alarm	35	6.92%	18	51.42%	17	48.57%
AlarmIhm	42	8.30%	12	28.57%	30	71.43%
Timer	16	3.16%	8	50%	8	50%
TimerListener	2	0.40%	2	100%	0	0%
AlarmHandler	2	0.40%	2	100%	0	0%
Total	**506**	**100%**	**171**	**33.79%**	**335**	**66.21%**

3.2.1.3 Results Analysis

The experiments presented in section 3.2.1 demonstrate that a large number of lines of code cannot be automatic generated from the UML model. Those lines represent most of the method behavior. In the Crane case study, this number represents 40% of the whole application, but this number can vary according to the used SW architecture. In the Crane model, several classes are used to model shared resources and that do not encapsulate any behavior. For that reason, 100% of code can be generated for these classes, increasing the percentage of AG lines of code for the Crane application.

In the Address Book case study, only 171 (33.79%) lines of code out of 506 (100%) can be automatically generated. Consequently, 66.21% of the lines are written by the designer. It shows that the challenge of generating code depends on the application and how it is organized. This case study demonstrates also that the choice of data structure impacts on the number of lines of code generated automatically. This comes from the fact that, when data structures are reused, their methods can be reused, avoiding the need for the designer to implement methods for data structure manipulation.

This study aims to propose an appropriate abstraction that could be used to complement the UML models, in such way that complete code could be generated from them. Usually tools use programming or action languages to do this. Both have the disadvantage of the fact that the designer needs to specify the behavior by a code block that is not smaller than the used in the final implementation. This happens because action languages do not provide a higher abstraction level than those provided by the programming languages themselves (e.g. C++, Java, etc). The analysis presented in section 3.2 was useful to observe the kind of WD lines that are usually found in embedded applications. In the next section, we propose forms to abstract these lines of code in order to improve the capabilities of code generation from behavioral UML diagrams and allow full code generation from them.

3.3 Proposed code generation approach

Most part of the approaches for code generation were defined for the first versions of UML (1.4 and 1.5). They do not have formal semantic, allowing different interpretation from UML models. In addition, these versions do not provide a way to link the several behavioral diagrams in order to allow an easy capture of the system behavior. The latest version of UML2 proposes a way to link several sequence diagrams in order to allow the capture of an execution sequence, which turns able the definition of code generation methods from sequence diagrams.

In addition, the previous sections discussed the existing gap between UML models and the final implementation in the target programming language, showing that additional information should be inserted in the model in order to allow the complete code generation from that. In this context, we propose here an approach for full code generation from UML2 models, which uses abstractions to describe the behavior of the methods. The flow of the proposed approach is presented in Figure 3.9, which starts from the application model described using UML diagrams. After that, the designer refines this model, specifying the behavior for methods using an abstracted language called BRISA (*BRIdging the Semantic Abstraction*). Finally, the resulting model is used as input for the code generator that generates code in the target programming language.

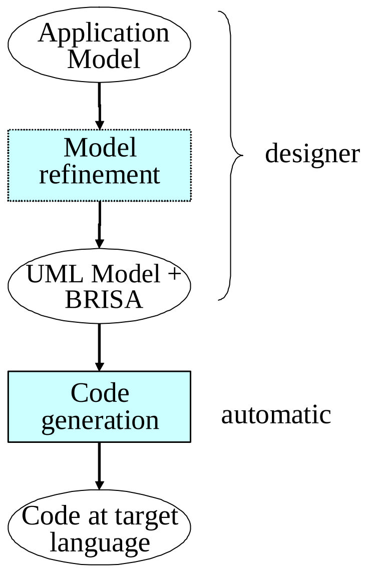


Figure 3.9: Proposed UML-based code generation flow

The ultimate objective is the definition of a code generation method able to generate complete code from a high-level model of an embedded application, which provides an automatic way to obtain the final implementation from the model. The use of the UML2 notations to define the system behavior is addressed in the section 3.3.1. Section 3.3.2 presents the abstraction that must be provided by BRISA.

3.3.1 Using UML2 notations for code generation from sequence diagrams

UML2 defines thirteen types of diagrams, divided into three categories: six diagram types represent static application structure; three represent general types of behavior; and four represent different aspects of interactions. As the focus here is on the improvement of capturing the dynamic application behavior from the UML model, only behavioral and interaction diagrams are cited.

Behavior Diagrams include the Use Case Diagram (used by some methodologies during requirements gathering); Activity Diagram, and State Machine Diagram.

Interaction Diagrams, all derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

The UML2 specification puts more emphasis on the semantics and, in particular, in the key area of basic behavioral dynamics. With an evolution of the UML modeling language, new notations and constructions were proposed. Since we are interested in generating code from sequence diagrams, our focus here will be on the new features included in this diagram.

Figure 3.10 shows an example of UML2 sequence diagram. It looks much the same as the sequence diagrams in UML 1.x, as they still have lifelines, messages and other similar notations, but there are some apparent differences. In the UML2, the sequence diagrams can be divided in fragments. Notice the first box at the upper left hand corner of the diagram. It is a new notation specifying the name of the fragment by the use of the operator <sd>, in the example, "sd Q". Using the name of the fragment and another operator called <ref>, as shown in Figure 3.10, other sequence diagram can reference this fragment. In this way, different sequence diagrams can be linked, defining the application execution flow.

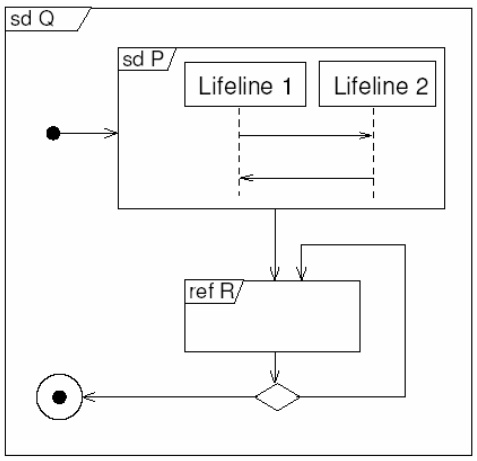


Figure 3.10: Combining different iterations notations in a same diagram

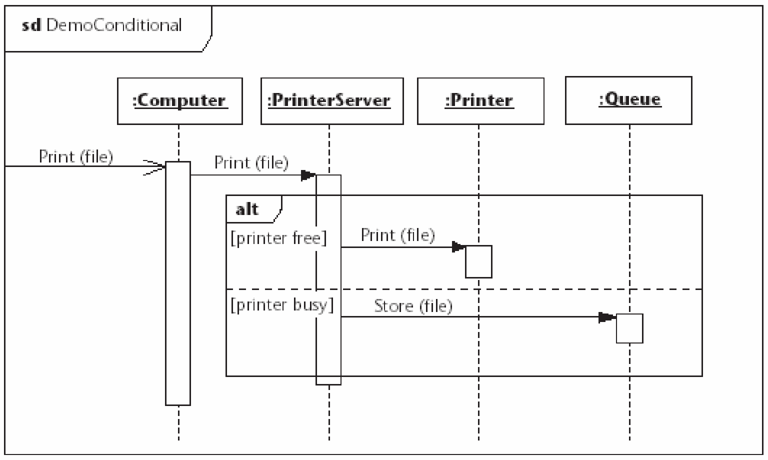


Figure 3.11: Representing conditionals in UML 2.0

In Figure 3.11, there is another sequence diagram nested within a larger one. It has the operator <alt>, which is short for "alternative" (*if/then/else*) and applies to the nested fragment. The dashed line is used to delimitate the alternative fragments and, if the guard evaluates to TRUE, then the upper part of that fragment is executed. Otherwise, the lower part will be executed. The loops are indicated by the operator <loop>. Together with the operator, a boolean expression (conditional) or a minimum and maximum index can also be specified. Figure 3.12 shows an example of the use of the <loop> operator, where the operations inside of the loop are repeated four times. These notations allow specifying conditionals and loops in the sequence diagrams. Besides that, the <par> operator can be used to specify parallel (concurrent) behavior.

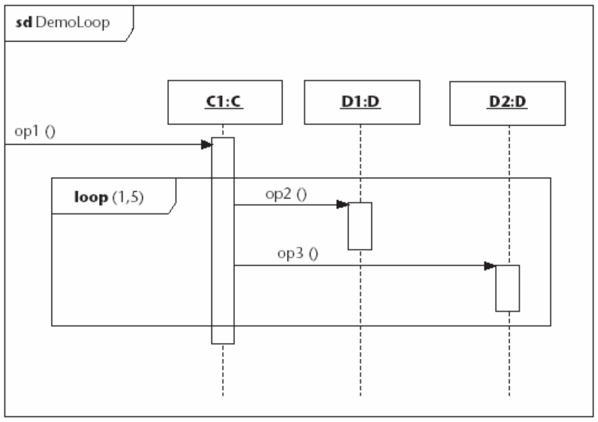


Figure 3.12: Representing loops in UML 2.0

We considered the UML2 notations for the sequence diagrams previously presented as an important improvement for the UML behavioral diagrams. They enable the capturing of method invocation sequences in a scenario or whole application execution flow, barely from the sequence diagrams. These new notations make possible to do links between model and code, reducing the gap between both and facilitating the code generation.

Besides the sequence diagram, the activity diagram has gained attention in UML2. Activity diagrams are similar to flowcharts and can be used to define the behavior of methods (algorithm), once these diagrams also allow the specification of loops and conditionals. However, it is important to notice that the use of very detailed diagrams require so much time to build that the designer may prefer to specify the algorithm directly as code in the target programming language. In UML 2.0, the activity diagram semantics is oriented with Petri nets semantics. It defines activities and actions that produce and consume tokens rather than on state charts. The explicit modeling of control and object flows are new in UML 2.0, replacing the use of state transitions in previous versions of UML activity diagrams. Green (2005) proposes the use of this diagram to specify dataflow.

3.3.2 Bridging the semantic gap

As the experiments in section 3.4 demonstrated, a huge number of lines of code must be written by the designer using the traditional code generation approaches. In order to address it, a more abstract language could be used to make the lines of code specified under the UML model more abstract than the programming language. This could motivate the designer to use the UML-based code generation approaches.

The analysis of two applications developed in Java, a widely used programming language, allows identifying and classifying the line codes. Observe that several lines of code can be replaced by only one line in languages like Matlab or Python, which provide abstractions to manipulate matrix and vector (see Figure 3.1). In this case, a library that provides functions to perform operations under matrix can be used to facilitate the production of implementation for these operations. As embedded applications involve math operation with matrixes, the use of higher abstraction to describe these operations allows reducing the time spent in the specification.

Experimental results show that the use of a component library with the UML model can reduce the number of hand written lines of code in 30% (BRISOLARA, 2005b). To indicate the reuse of components, stereotypes can be used in the UML diagrams. In this way, the designer does not need to describe the behavior for the methods marked as reused, since an implementation is already available in a library. In addition, as proposed in (MATTOS, 2004), a library with pre-defined components implemented in different ways and pre-characterized for a given architecture can support design space exploration and the generation of more efficient code for this architecture.

In the AddressBook application, several routines such as search in a data structure, sorting elements, insertion and removal of elements, also can be reused from libraries, avoiding the hand-coding. The results of the analysis performed on this application show that a reduction of about 63% on the lines of code of a class can be achieved when operations to manipulation of data structure are reused. According to the classification of the lines of code required to be written by designers, abstractions are proposed in order to facilitate the specification of the method behavior and reduce the total lines of code that the designer is asked to specify.

Some lines of code responsible to create objects (<co>) can be generated automatically, using the information from the class diagram like the definition of attributes and the relationship between classes. Moreover, the creation of static objects also can be automatically generated. However, object-oriented implementation can have also dynamic allocation and, in this case, the creation of objects must be specified by the designer.

All the lines of code that represent method invocation (<cm>) can be obtained from the sequence diagrams, where method calls are used to show the iteration between objects. The instructions classified as <for + cm> can be specified with a sequence diagram or with an activity diagram. For example, in an UML 2.0 sequence diagram, loops can be described and the method calls can be specified inside of them. Loops and conditionals can also be captured from sequence or activity diagrams, as exemplified in section 3.5.1. In these cases, only the graphical notation is required.

On the other hand, the instructions <for + de>, which normally are described in 2 lines in Java, could be described in a single line using a language that facilitate the manipulation of matrixes and vectors. An example is shown in Figure 3.4, where a loop (for) is used to copy the elements from a vector to another vector. The new version of the Java language also provides functions to do a copy between vectors, so a single code line can do the same. Similar abstraction can be used in loops that perform a vector initialization. The example illustrated in Figure 3.13(a) and (b), show two version of Java code for a vector initialization. The same code could be described in Python or Matlab using a single line like as a=0 or a=[0; 0; 0; 0; 0]. In this case, a loop is not required to describe the initialization.

```
For (i:=0;i,<5;i++)
{
        a[i] = 0;
}
```

(a)

```
    a[0] = 0;
    a[1] = 0;
    a[2] = 0;
    a[3] = 0;
    a[4] = 0;
```

(b)

Figure 3.13: Matrix/vector initialization in Java

Matrix and vector multiplication are common operation in embedded application that evolves signal processing. Figure 3.14 illustrates an example of vector multiplication in Java, where two nested loops are used to do the operation. A function can be defined to facilitate the specification of a vector or matrix multiplication, as shown bellow.

Mul(a,q);   // multiply vector *a* and vector *q*

```
for (i=0; i < N; i++)
{
     for (int j=0; j < N; j++)
         tmp[i] += a[j] * q[j];
}
```

Figure 3.14: Vector multiplication in Java

In this case, a pre-compiler can be used to verify the number of lines and columns of variables *a* and *q* and to generate the correspondent Java code using a template. The same approach can be used to perform matrix multiplications, simply using the pre-compiler to determine the appropriate template through the analysis of the number of lines and columns found in the matrix. Figures 3.14 and 3.15 show an example of vector multiplication and matrix multiplication, respectively. Both Java codes could be produced through the use of templates.

```
for (i = 0; i < N; i++)
{
        for (j = 0; j < N; j++)
    {
         temp[i][j] = 0;
         for (k = 0; k < N; k++)
             temp[i][j] += m1[i][k] * m2[k][j];

     }
}
```

Figure 3.15: Matrix multiplication in Java

Furthermore, notations could be used in the UML diagrams to indicate the necessity of creating structures like <switch/case> and <try/catch>, as exemplified in Figure 3.16. In this way, from the UML diagrams, skeleton of code could be automatically generated.

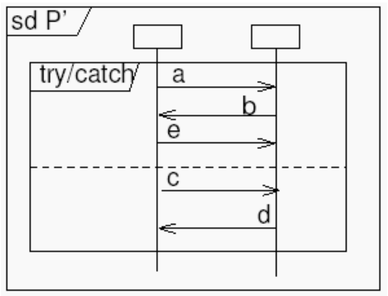


Figure 3.16: *Try/catch* notation in sequence diagrams

3.4 Concluding remarks

In this chapter, UML-based software generation approaches were discussed and a proposal to solve a limitation found on these approaches was presented. However, this proposal leads to the extension of the programming language or the definition of a new one. We consider that the definition of another language could deviate the main objective of this thesis, once our main focus is on the modeling approach and strategies for automating the embedded software design from models.

Moreover, a more detailed analysis of the evolution of the Java language allowed us to observe that some abstraction proposed here are already treated by the new versions (Java 5 and Java 6) of the language. This analysis shows that, in the future, programming languages will also provide very high abstraction.

So, we have given up these ideas, although we believe that this proposal could obtain good results. This happened when the author had the opportunity to work in the development of a code generator based on Simulink. This has shown to be a very interesting study, so we decide to follow this new thread.

In the next chapter, a Simulink-based code generation approach will be presented, which allow one to generate multithread code targeting multiprocessor architectures, something that is not provided by RealTime Workshop (MATHWORKS, 2004).

4 SIMULINK-BASED EMBEDDED SOFTWARE GENERATION

Nowadays, several embedded systems make extensive use of digital signal processing, requiring a language that supports the dataflow model of computation. However, despite several efforts to extend UML for modeling dataflow applications, UML still does not cope very well with this model of computation, as discussed in chapter 2. In this context, we propose a Simulink-based embedded software generation approach targeting multiprocessor systems.

The main motivation of this work is the fact that heterogeneous multithreaded multiprocessor SoC (MPSoC) architectures are becoming an attractive solution for embedded systems. As indicated by Jerraya (2005), they provide highly concurrent computation and flexible programmability. However, making software for heterogeneous multiprocessors in MPSoC platforms is now becoming a major challenge. The main causes for this are the difficulty of parallelizing target applications, the software adaptation to different processors and protocols, the short design time, and low cost implementation.

In addition, the majority of MPSoC applications require a large amount of memory that may heavily affect the cost and the power consumption. Communicating threads are distributed in a MPSoC architecture and the communications among them impact also on system performance. This indicates that an automated code generation method, which can generate efficient multithreaded code and automatically adapt it to the heterogeneous processors and protocols, is indispensable.

We propose a Simulink-based multithread code generation approach. Our goal is to address those software programming difficulties and support the development of efficient embedded software targeted to heterogeneous MPSoC platforms. To meet hard requirements for memory size and performance commonly found when designing embedded systems, memory usage and communication optimizations are proposed to be applied during the code generation. Some results were published in (BRISOLARA, 2007a).

We have chosen Simulink as a tool for specification and simulation mainly because it is widely accepted to specify complex systems, and can be considered as a standard tool in the signal processing domain. It offers a set of algorithms for a variety of applications, and is powerful to specify data-intensive and control-dependent algorithms. From a Simulink model, one can generate a single-thread code targeting a single processor platform using Real Time Workshop (RTW). Another tool called Real-

Time Interface for Multiprocessor Systems (RTI-MP) (DSPACE, 2005) automatically generates software code from a specific Simulink model for multiprocessor systems. However, the generated software code aims at a specific architecture consisting of several commercial off-the-shelf (COTS) processors boards, where the main purpose is high-speed simulation of control-intensive applications.

The proposed multithread code generation approach was developed during a PhD internship, being part of a major project developed at TIMA Laboratory. The project proposed a new MPSoC design flow based on Simulink, which is detailed in (HUANG, 2007). The Simulink-based multiprocessor SoC design flow is presented in Figure 4.1 and starts with Simulink modeling (step 1) to make a Simulink application model from a target application specification. The Simulink application model is transformed into a Simulink combined application/architecture model (CAAM). That is an unified model, which combines aspects related to the architecture model, i.e. processing units available in the chosen platform, into the application model, i.e. multiple threads executed on the processing units. This happens in step 2. In step 3, *Simulink parser* parses a Simulink CAAM and generates a Colif CAAM, which is a XML-based intermediate representation, as defined in (CESARIO, 2001). Afterwards, *Hardware architecture generator* (step 4) produces the multiprocessor hardware architecture models. These models are composed of CPU subsystems, HW subsystems, memory subsystems, and communication network between them, all at different abstraction levels. On the other side, *Multithread code generator* (step 5) generates a multithreaded code and a main code. The latter is responsible for creating threads and initializing communication channels through hardware dependent software (HdS) primitives.

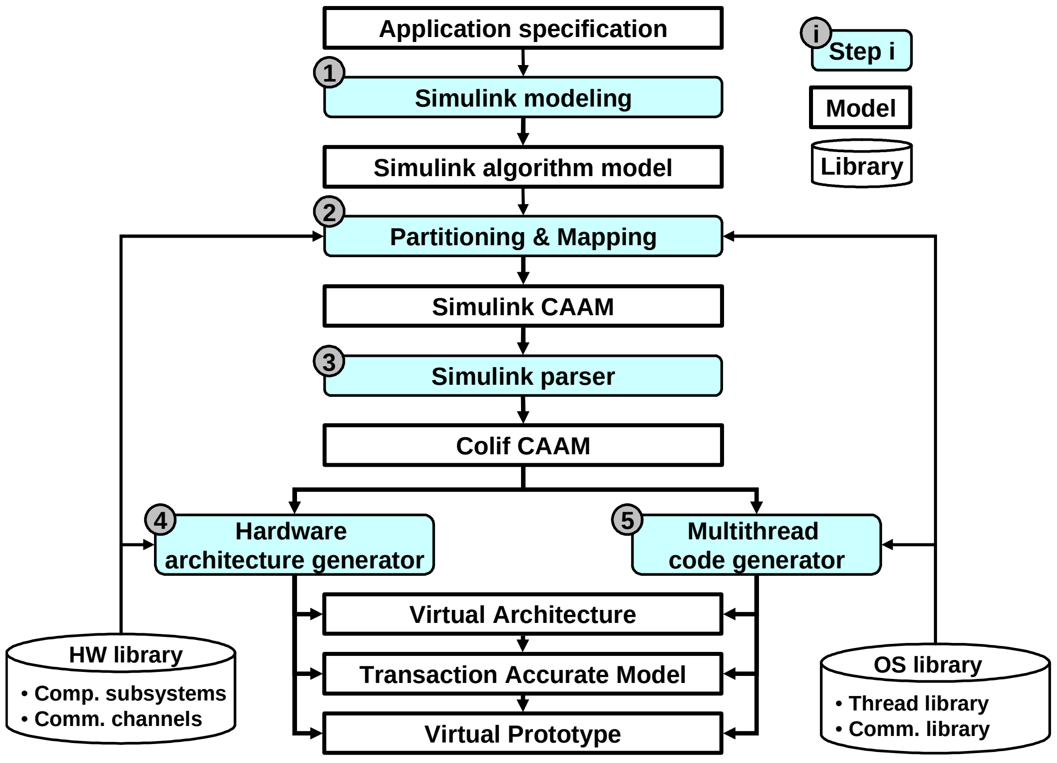


Figure 4.1: Simulink-based MPSoC design flow (HUANG, 2007)

The main objective of the Simulink-based MPSoC design flow is to support a mixed hardware software refinement procedure. It starts from the Simulink CAAM and uses three abstraction levels to refine the system, comprising a high-level specification down to detailed low-level implementation. These abstraction levels are *Virtual Architecture*, *Transaction-accurate model*, and *Virtual Prototype*, and are generated by the *Hardware architecture generator*. The *Hardware architecture generator* is detailed in (HUANG, 2007) and is out of the scope of this work. Since the focus here is on software generation, the *Multithread code generator* will be detailed here.

Firstly, section 4.1 describes the combined application architecture model (CAAM), which is used as input for the *Multithread code generator*. The multithread code generation flow is presented with detailed steps in section 4.2. Proposals for optimization on memory and communication are presented in section 4.3 and 4.4, respectively. Section 4.5 presents experiments performed with the *Multithread code generator*.

4.1 Combined application architecture model

Traditional design flow makes use of two separate models: application and architecture. The application is generally specified as an application model made of a set of multiple cooperating threads (or tasks). Each of them performs a subset of functions of the application. The multiple threads of the application will be mapped on the target architecture, which can be specified as a set of processor subsystems interacting via communication network. The processor subsystem contains processing unit, specific I/O and different hardware components to speed up communication.

Popovici (2007) proposes combining these two models in a mixed hardware software architecture, where the software threads are mapped on the abstract CPU subsystems, as shown in Figure 4.2. The result is a mixed hardware software architecture model at a very high-level representation, which is called combined algorithm/architecture model (CAAM).

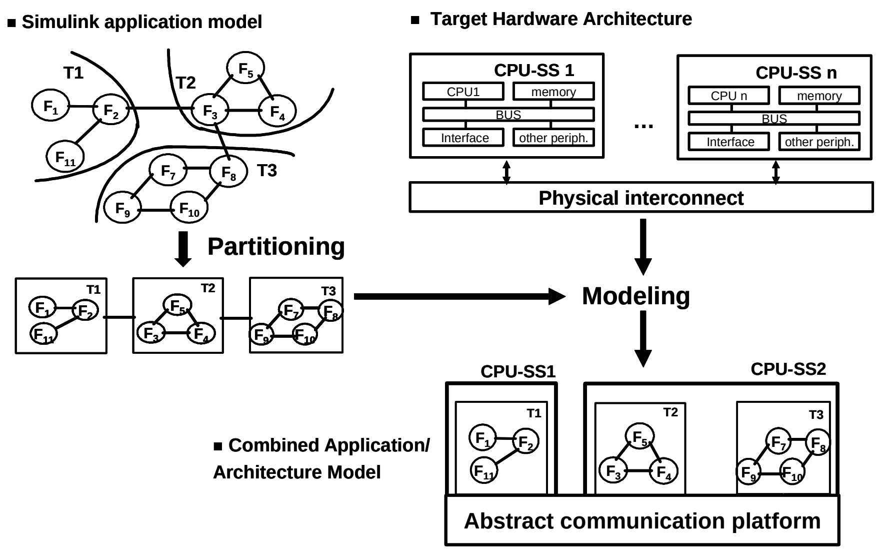


Figure 4.2: Combining application and architecture models (POPOVICI, 2007)

In the proposed Simulink MPSoC design flow, we specify the CAAM using a three-layered hierarchical Simulink model. The first layer describes a system architecture, which contains CPU subsystems and inter-subsystem communication channels (Inter-SS COMM). The second layer describes a CPU subsystem architecture, composed of software threads and intra-subsystem communication channels (Intra-SS COMM). The third layer describes a software thread using Simulink blocks and data links.

Figure 4.3 shows an example of CAAM. In this example, there are four CPU subsystems (CPU0-CPU3) and six Inter-SS COMMs (CH0-CH5) in the first level, and seven threads (i.e. T0-T6) and three Intra-SS COMMs (CH6-CH8) in the second level. To simplify the view, the Figure 4.3 only illustrates the Simulink blocks that compose the threads T0 and T1, allocated for CPU0 and CPU1, respectively. To represent mixed hardware and software model in Simulink, four kinds of specific Simulink subsystems are defined as followings.

- *Processor subsystem*, which includes one or more thread subsystems. It is a processing element such as RISC processor and DSP. A processor subsystem is refined to a CPU subsystem, e.g. processor, local bus and local memories, by the *Hardware architecture generator*.
- *Thread subsystem* represents a thread on a processing unit. This subsystem includes one or more Simulink blocks used to represent the thread functionality. A thread subsystem is refined to an OS dependent thread by the *Multithread code generator*.
- *Inter-Subsystems Communication (Inter-SS COMM)*, which includes one or more Simulink data links, represents the communication channels between CPU subsystems. An Inter-SS COMM is refined to a hardware communication channel by the *Hardware architecture generator* and software communication port(s) to access the channel by the *Multithread code generator*. HWFIFO is a communication protocol that transfers data via hardware FIFO. GFIFO is another one that transfers data via a shared memory and a global bus, and synchronizes via mailboxes.
- *Intra-subsystems Communication (Intra-SS COMM)*, which includes one or more Simulink data links, represents communication channels between threads on the same CPU subsystem. An Intra-SS COMM is refined to OS communication channel(s) by the *Multithread code generator*. SWFIFO represents a software FIFO.

These subsystems are normal Simulink subsystems, which do not affect the original functionality, annotated with several architecture parameters, e.g. processor type and communication protocol. Currently, this transformation is manually performed by using the Simulink graphical interface and relies on the designer's experience. For example, to make a thread subsystem, the designer can cluster several Simulink blocks into a Simulink subsystem by a shortcut key and then annotate "Thread" as type to the subsystem through a parameter setting.

Currently, the environment supports three communication protocols: GFIFO, HWFIFO, and SWFIFO. GFIFO (Global FIFO) is an inter-subsystem communication protocol that transfers data using a global memory, a bus, and mailboxes. The data transfer is divided into two steps. First, the CPU in the source subsystem writes data to a global memory, and sends an event to the mailbox in the target subsystem. After receiving the event, the CPU in the target subsystem reads the data from the global memory, and sends another event to the mailbox in the source subsystem, notifying the completion of the read operation. HWFIFO is also an inter-subsystem communication protocol that transfers data via a hardware FIFO. SWFIFO is an intra-subsystem communication protocol based on software FIFO.

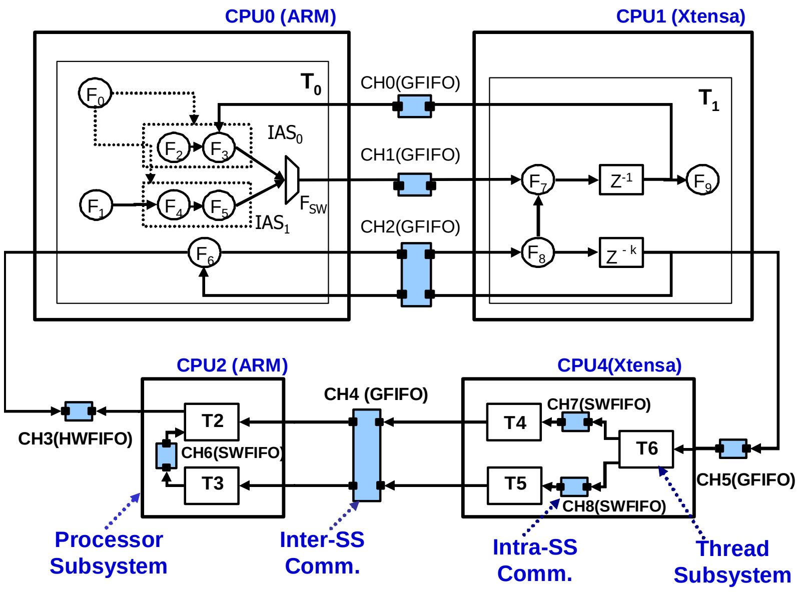


Figure 4.3: A Simulink CAAM example (HUANG, 2007)

4.2 Multithread code generation

Our multithread code generation method was designed as an extension for the code generation method presented in Han (2006b), which is able to generate sequential C code from Simulink models. We used a restricted Simulink subset in our modeling, which was defined in (HAN, 2006a) to represent global data and control dependencies precisely. This Simulink subset includes blocks, delays, links, If-action subsystems (IAS), and For-iterator subsystems (FIS), as well as a global clock that controls the execution of blocks and delays. This model is based on the Abstract clock Synchronous Model, ACSM (HAN, 2006a), and can be statically scheduled and its memory can be also statically allocated during the code generation.

Multithread code generator produces a set of C thread codes, a main C code and a Makefile for each CPU subsystem. The proposed software code generation is made in three steps, as illustrated in Figure 4.4. Firstly, the *Simulink parsing* traverses the Simulink CAAM and generates a Colif CAAM that is used as intermediate format. In the second step (Thread code generation), the blocks within a thread-SS are scheduled statically according to data dependency and the code generator produces a C code. The generated threads are dynamically scheduled by the OS scheduler according to the availability of data for the input port or space for the output port. In the third step (HdS adaptation), a main code and a Makefile is generated for each CPU-SS. The main code is responsible to initialize the threads and the communication channels among them. To build an executable software stack, the generated Makefile compiles the thread codes, the main code and links them with an appropriate HdS library built for the target CPU subsystem, as shown in Figure 4.4. This approach avoids that the designer needs to adapt the software code to different processors/protocols, and distributing data and code.

Designing embedded systems requires concern with hard constraints for memory size and performance issues. Hence, we propose applying memory and communication optimizations techniques to reduce memory size and improve performance, during the code generation. Both optimization proposals are presented in section 4.4 and 4.5.

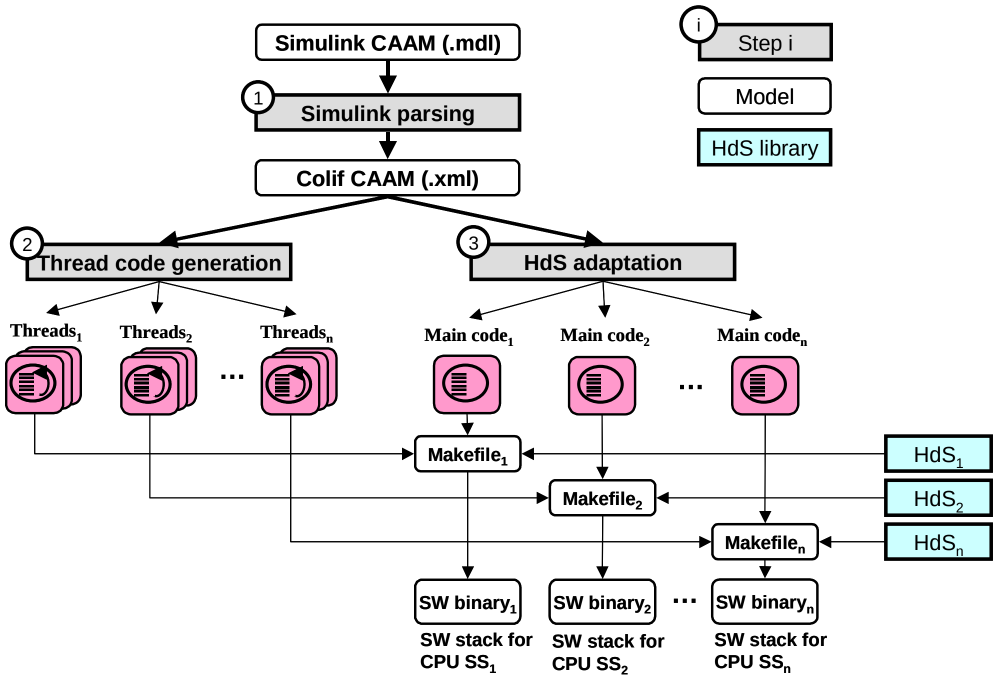
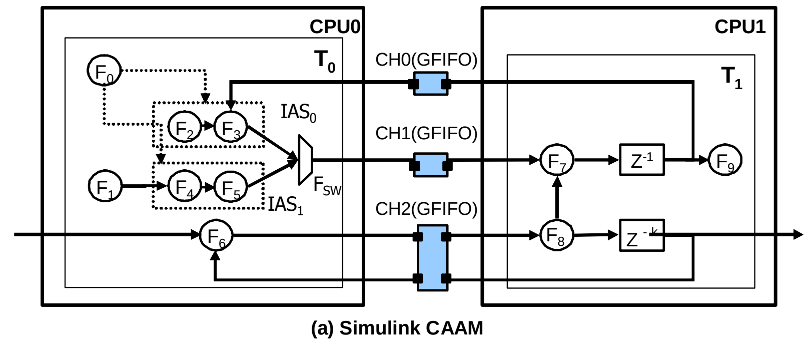


Figure 4.4: Multithread code generation flow

4.2.1 Simulink parsing

Simulink Parser parses a Simulink CAAM model (Figure 4.5(a)) and generates an equivalent intermediate format called Co-design Language Independent Format (Colif) (CESARIO, 2001), shown in Figure 4.5b. Colif is a XML-based meta-model used as intermediate format in the whole proposed Simulink-based design flow.

To generate a multithreaded code communicating with each other, the Simulink data links with Inter-SS COMM or Intra-SS COMM are translated to a pair of send and receive operations. Simulink parser reads an input Simulink CAAM (Figure 4.5(a)) and inserts send ("S" in Figure 4.5(b)) and receive ("R" in Figure 4.5(b)) blocks into a Colif CAAM. These *send* and *receive* blocks are scheduled together with the other blocks in the Thread code generation, as will be explained in section 4.2.2.



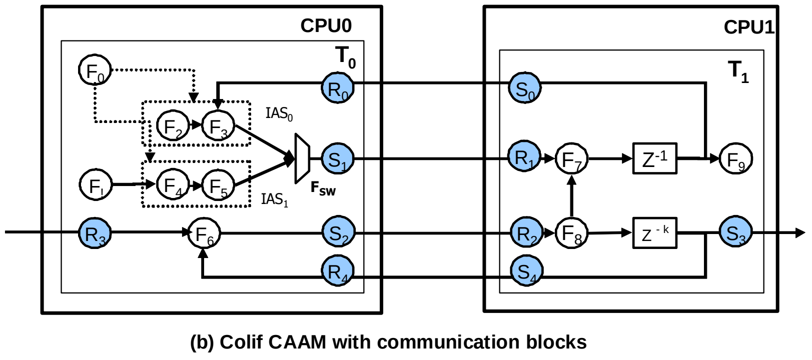
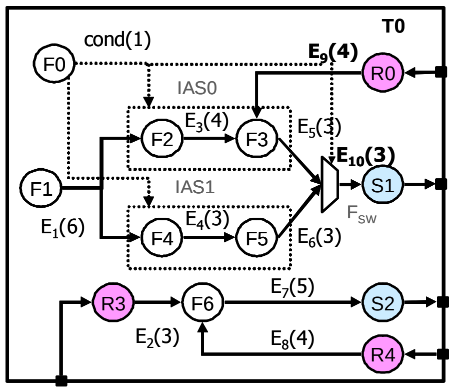


Figure 4.5: Simulink parsing

4.2.2 Thread code generation

The thread code generator automatically produces a C-code for each thread, which includes memory declaration and behavior code for user-defined blocks, communication blocks, and pre-defined blocks. First, our tool generates memory declaration(s), where a memory space is declared for each data link according to its data type, e.g. char, short, int, etc. The allocated memory is used to store the input and output data of Simulink blocks. Afterwards, a behavioral code for each thread is generated according to the scheduling result, which statically determines the invocation order of blocks according with data dependency.

Figure 4.6 illustrates an example of Thread code generation. Each link in the Figure 4.6 (a) is annotated with a buffer name and its size. For example, E2(3) means buffer E2 with size 3. Figure 4.6 (b) shows the code generated for thread T0. Line 1 declares port data structures used to promote the communication. In line 2-4, buffer memories are declared. For a user-defined block (i.e. Simulink S-function), our tool generates a function invocation corresponding to the block (F0-F6 in example) and maps the allocated memories for the input and output links to the function arguments. When a pre-defined Simulink block is used, e.g. adder or If-action subsystem (IAS), C codes corresponding to the specific blocks are generated (if-else for the IAS in example). The code generator can handle a large subset of pre-defined Simulink blocks such as mathematical operations, logical operations, discrete blocks, etc.



a) Colif CAAM

```
// port declaration
1: extern port_t *in0, *in1, *in2, *out0, *out1;
2: int cond, int E1[6]; // mem declaration
3: int[5] E7; int[4] E8, E9, E3;
4: int[3] E2, E4, E5, E6, E10;
5: while(1) {
6: recv_data(& in0, E9, 16);  //R0(E9);
7: recv_data (& in1, E8, 16); //R4(E8);
... // R3(E2);
8:  F0(cond); F1(E1);
9:  if(cond){ F2(E1,E3);
10:     F3(E3,E9,E5); }
11:  else     { F4(E1,E4);
12:      F5(E4,E6); }
13:  if(cond) E10 = E5;
14:  else  E10 = E6;
15:  F6(E2, E8, E7);
16: send_data(&out0, E10, 12); //S1(E10)
 ... //S2E7)
17: }
```

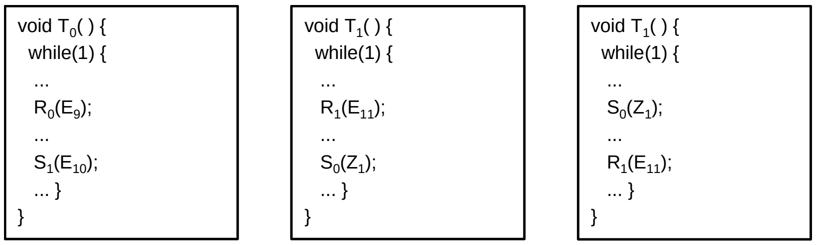
b) Thread Code of T0

Figure 4.6: Example of thread code generation

For communication blocks, e.g. send and receive blocks discussed in section 4.2.1, our tool inserts communication primitive invocations defined in Table 1 (*send_data* and *recv_data* in the example). These invocations promote the communication between different threads, which can be in the same CPU (intra-subsystem) or in different CPUs (inter-subsystems). The arguments of the communication primitives, determined by Simulink Parser, are port data structure address, memory address allocated, and data transfer size. For example, the code generator generates line 6 for R0 block where the associated port data structure is *in0*, output buffer is E9, and the transfer size is 16 bytes, as shown in Figure 4.6(b).

As proposed by Han (2006b), we extended the existing dataflow-based scheduling methods for Simulink models to support nested-conditionals and loops. In the used scheduling algorithm, all blocks in the input model, including all threads, are scheduled together according to their precedence dependency. If R0 is invoked prior to S1 in T0, as shown in Figure 4.7(a), and R1 is invoked prior to S0 in T1, as Figure 4.7(b), a precedence loop is introduced (R0→S1→R1→S0→R0) in the system, causing deadlock. In the proposed scheduling algorithm, R1 must be invoked after S0, as shown in Figure 4.7(c), because they have a precedence dependency even if it is across two threads. Our approach guarantees that any partitioning of the algorithm model has at least one deadlock-free schedule.

To guarantee that, designers are asked to build a model that has no precedence loop without a Delay block, following the ACSM model defined in (HAN, 2006). This model is composed of a network of state-less functions and delays. Delays are used as a temporal barrier, like registers in a synchronous circuit. This makes possible to describe the functionality of a system deterministically independent of the time taken for each function.



```
void T0( ) {
  while(1) {
   ...
   R0(E9);
   ...
   S1(E10);
   ... }
}
```

(a) Thread T_0 code

```
void T1( ) {
  while(1) {
   ...
   R1(E11);
   ...
   S0(Z1);
   ... }
}
```

(b) Thread T_1 code with deadlock

```
void T1( ) {
  while(1) {
   ...
   S0(Z1);
   ...
   R1(E11);
   ... }
}
```

(c) Thread T_1 code without deadlock

Figure 4.7: Multithread deadlock problem

4.2.3 HdS adaptation

The Hardware-dependent software (HdS) is responsible to provide architecture-specific services such as scheduling of application threads, communication inter and intra-CPU, hardware resources management and control. *Multithread code generator* produces a high-level multithread code independent of the architecture details through the use of high-level primitives provided by an HdS library. To execute the generated code on a target MPSoC platform, the thread codes should be linked with the appropriate HdS library that provides architecture dependent implementations for the high-level primitives.

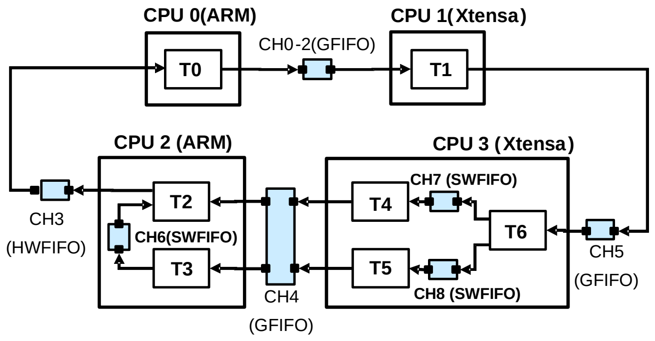
The HdS library should provide the high-level primitives summarized in Table 4.1. Using these primitives, *Multithread code generator* generates a main code, which initializes thread and channel data structures. A Makefile, linking the generated thread codes and main code with an appropriate HdS library, is also produced.

The HdS library includes HdS APIs, an Operating System (OS), communication software and a HAL (Hardware Abstraction Layer). The Operating System is composed of a Thread Scheduler and an Interrupt Service Routines (ISR). We first assume that there are pre-built HdS libraries, each of which is targeted to a specific CPU. Currently, we have targeted the HdS library to ARM7 and Xtensa processors. As mentioned before, the current HdS library supports three communication protocols: GFIFO, HWFIFO, and SWFIFO.

Table 4.1: HdS primitives

Types	**Primitives**	**Description**
Thread	thread_create	Create software thread
	thread_resume/thread_suspend	Resume/suspend thread
Communication	send_data/recv_data	send/receive data from/to port with specific protocol
	send_event/recv_event	send/receive event, e.g. data transfer completion, from/to port with specific protocol
	port_init/channel_init	initialize port/channel data structure
Interrupt	ISR_attach/ISR_dettach	attach/detach interrupt service routine
	intr_enable/intr_disable	enable/disable interrupt

Figure 4.8 shows an example of the main code and Makefile generation. Figure 4.8(a) shows a Colif CAAM example that contains four CPU subsystems and seven threads. Figure 4.8 (b) and (c) illustrate the main code and the Makefile for CPU0, respectively. The main code performs interrupt registrations (*ISR_attach* in example), channel initializations (*channel_init* in example), initialization (*port_init* in example), and thread creations (*thread_create* in example) according to the CAAM model. The Makefile defines directives for the compilation of the generated code, e.g. setting the compiler to be use and the files to be compiled according to the CAAM model. The Makefile for CPU0 shown in Figure 4.8(c) compiles T0 code and the main code with ARM compiler and links them with the ARM HdS library since the processor type for this subsystem was set as ARM in the CAAM model (Figure 4.8(a)).



(a) Simulink CAAM

```
channel_t ch3, ch0, ch1, ch2;
void main( )  {
channel_init(&ch0,GFIFO,  …);
channel_init(&ch1,GFIFO,  …);
 …
channel_init(&ch3, HWFIFO,  …);
thread_create(T0,  …);
 …
thread_exit( );  }
```

(b) Main code for CPU0

```
CC = arm-elf-gcc    // ARM C compiler
...
SRCS=T0.c   main.c   // file to compile
...
FLAGS= -DCPU=ARM7 -DDEBUG
...
LIBS=libhds-arm.a  // library HDS
...
```

(c) Makefile for CPU0

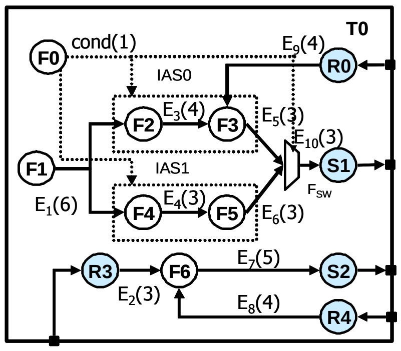
Figure 4.8: Main and Makefile code generation

The Makefile also enables to link the generated multithread code and main code with application library including user-defined function bodies and appropriate HdS library. In this way, with the proposed software programming environment, one can build binary files that are executable on the target heterogeneous MPSoC, making designer free from laborious programming work.

4.3 Memory optimization

Since the majority of MPSoC applications require a large amount of memory that heavily impacts on the cost and the power consumption, software memory optimizations are essential techniques to design cost and power effective embedded systems. In this section, we focus on memory optimization techniques in generating thread code. As proposed in (HAN, 2006b), two memory optimization techniques: **copy removal** and **buffer sharing** can be applied to reduce the required data memory size during the code generation. These techniques, firstly proposed for single-thread code generation, were extended for multithread case and integrated in our *Multithread code generator*. With this integration, the *Thread code generation* is composed of four steps, as explained bellow.

The example illustrated in Figure 4.8 is used to explain these optimization techniques. Figure 4.9(a) represents Colif CAAM of thread T0 and Figure 4.9(b) shows the generated code without optimizations. Figure 4.9(c) and 4.9(d) shows generated code with copy removal and buffer sharing, respectively.



(a) Colif CAAM da T_0

```
1: while(1) {
2: recv_data(& in0, E9, 16); //R0(E9);
3: ... //R3(E2); R4(E8);
4:  F0(cond); F1(E1);
5:  if(cond){ F2(E1,E3);
6:     F3(E3,E9,E5); }
7:  else    { F4(E1,E4);
8:      F5(E4,E6); }
9:  if(cond) E10 = E5;
10:  else  E10 = E6;
11:  F6(E2, E8, E7);
12:  send_data(&out0, E10, 12) //S1(E10);
13: ...//S2(E7);
14: }
```

(b) Original C code of T_0

```
1: while(1) {
2: recv_data(& in0, E9, 16); //R0(E9);
3: ... //R3(E2); R4(E8);
4: F0(cond); F1(E1);
5:  if(cond){ F2(E1,E3);
6:     F3(E3,E9,E10); }
7:  else    { F4(E1,E4);
8:      F5(E4,E10); }
9:   F6(E2, E8, E7);
10: send_data(&out0, E10, 12) //S1(E10);
11: //S2(E7);
12: }
```

(c) C code of T_0 after copy removal

```
1: extern port_t *in0, *in1, *in2, *out0, *out1; // port declaration
2: int mp[15];                                          // memory pool
3: void T0( ) {
4:   while(1) {
5:     F0(cond);                                       // time 0
6:     …
7:     F6(&mp[1], &mp[4], &mp[8]);          // time 3, F6(E2, E8, E7)
8:     send_data(&out1, &mp[8], 20);           // time 4, S2(E7)
9:     recv_data(&in0, &mp[5], 20);            // time 5, R0(E9)
10:     …
11:    send_data(out0, &mp[5], 12);            // time 9, S1(E10)
12:   } }
```

(d) C code of T_0 after buffer sharing

Figure 4.9: Thread code generation with memory optimization techniques

Step 1. Copy removal: A Simulink CAAM may include control blocks (e.g. “Switch” and “Selector”) and delays (e.g. “Unit delay”) that introduce copy operations between the input buffer(s) and the output buffer(s). These pre-defined Simulink blocks are required to represent explicit conditionals or loops. Copy removal technique allows the input and output buffers to share the same memory space. After applying it to the model, the input buffers “E5”(line 6 in Figure 4.9(b)) and “E6’’(line 8 in Figure 4.9(b)) of switch “Fsw” in Figure 4.9(a) are merged with its output buffer “E10” (see line 6 and 8 in Figure 4.9(c)). This merge operation removes the lines of code 9 and 10 of Figure 4.9(b), as shown in Figure 4.9(c).

Step 2. Scheduling: The original static scheduling was modified in order to maximize buffer sharing in step 3. Figure 4.10(a) shows a buffer lifetime chart for the

T0 illustrated in Figure 4.9(a). In this chart, the horizontal axis indicates the invocation sequence, i.e. scheduling result, and the vertical axis indicates the buffer memory address location. Each rectangle denotes the lifetime interval of a buffer memory. Intuitively, the scheduling objective is to make the fattest point as thin as possible.

Step 3. Buffer sharing: The code generator performs a lifetime-based buffer sharing algorithm for each thread. This technique allows two buffers within the same thread to share the same memory space if their lifetimes are disjoint. Since buffer sharing problem is NP-complete (OH, 2003), an heuristic algorithm is required to solve it. We use an extension of the LOES heuristic algorithm proposed by Oh (2003) that can consider the conditionals in a Simulink model. Figure 4.10(b) shows a buffer lifetime chart after applying buffer sharing to the T0 model (Figure 4.9(a)).

Step 4. Code Generation: Thread code generator produces thread codes according to the results of the previous steps. As the buffer sharing is applied in the model, the memory declarations into the code follow the buffer sharing results.

Han (2006b) proposes some memory optimization techniques during single thread code generation. We extended here these optimizations in order to apply them in the multithread code generation. The used memory optimization techniques are extensions of the existing dataflow based scheduling methods (RITZ, 1995)(BALASA, 1995) for handling data-intensive and control-dependent target applications.

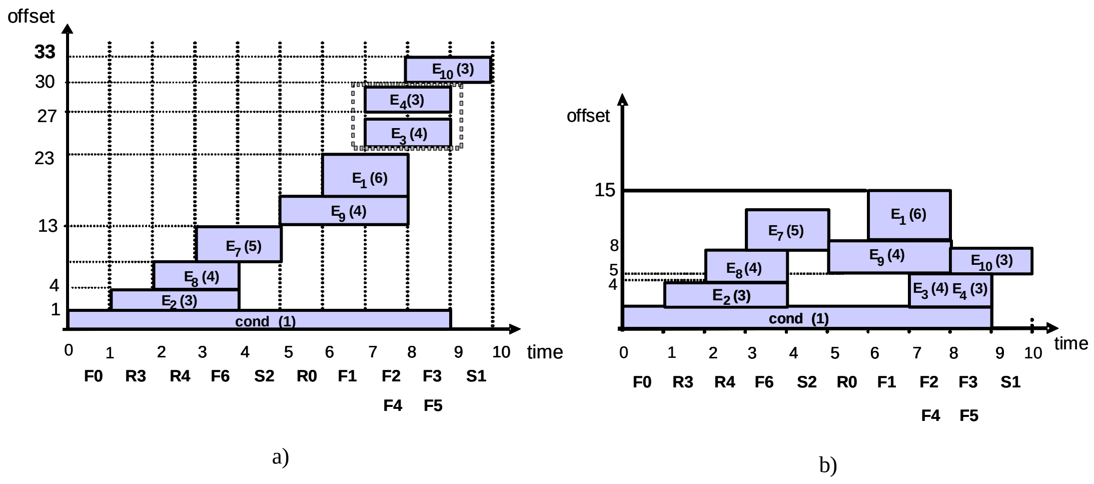


Figure 4.10: Lifetime chart of T0 (a) after scheduling, (b) after buffer sharing

Our multithread code generation supports only discrete model with a global clock. We do not handle any other models such as discrete model with multiple clocks and event-driven model, since the conventional memory optimization is hard to apply to them.

4.4 Communication optimization

When the number of processors increases in a MPSoC, the overall system performance heavily depends on the performance of communications among the processors. Therefore, communication optimization techniques are required to improve the system performance. Message Aggregation (MA) was firstly proposed in

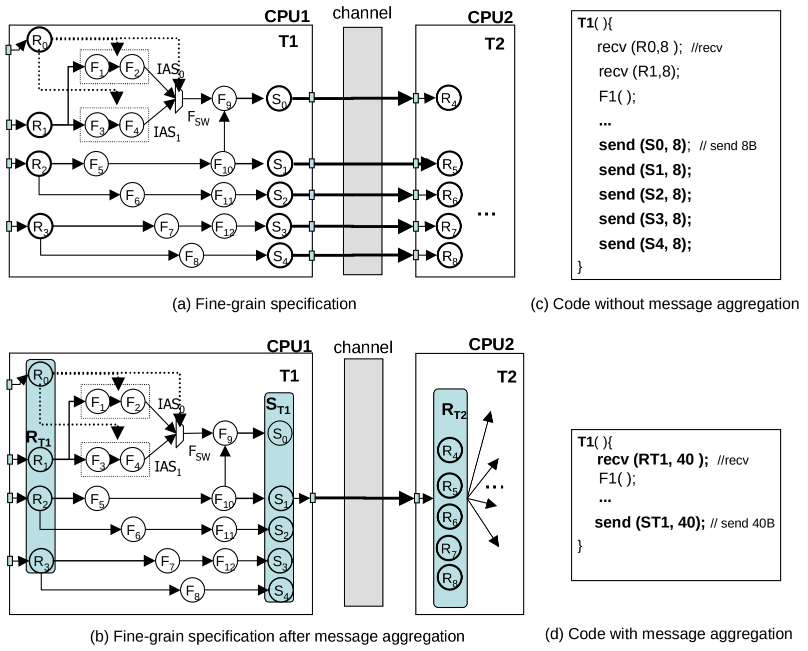
(HIRANANDANI, 1992) and it is a well-know communication optimization in the parallel computing and distributed systems domain. After that, a compiler that integrates several communication optimizations, such as Message Aggregation and Message Coalescing, was proposed for distributed-memory multi-computers in (BANERJEE, 1995).

In the proposed multithread code generation, when a Simulink functional model consists of fine-grain functions and it is partitioned into several processors, the Simulink parser will insert a large number of communication nodes that exchange messages through communication channels. Consequently, the communication overhead increases, which impacts on the system performance and the required memory size. In this context, Message Aggregation can be applied to increase the granularity of data transfers, reducing the communication overhead.

The cost for a data transfer in terms of execution time can be divided in start-up cost (synchronization cost) and effective data transfer cost (rate *length). The start-up cost does not depend on the number of bytes sent. Message Aggregation (MA) combines messages with the same source and destination, increasing the granularity of the data transfers and amortizing the start-up cost. Consequently, this technique can reduce the total amount of communication overhead in terms of execution time. Moreover, this technique can reduce the software data structures used to represent the channels to promote and manage the inter-processors communications. For example, a H.264 decoder Simulink CAAM with 6 CPUs requires 85 data structures for communication channels, which impacts on data memory size.

Figure 4.11 presents a motivational example. Figure 4.11(a) shows a partitioned high-level model, which consists of functional nodes (*Fx*), communication nodes (*Sx* for Send operation, and *Rx* for Receive operation), and links between them. After applying Message Aggregation technique on the model depicted in Figure 4.11(a), the high-level model shown in Figure 4.11(b) is obtained. Figure 4.11(c) and 4.11(d) illustrate the codes obtained from the two models. As result of this optimization, the five *Send* nodes (*S0-S4*) were grouped in a single node (*ST1)*, as shown in Figure 4.11(b). Consequently, the five *Send* primitives of Figure 4.11(c) are replaced for only one *Send* in Figure 4.11(d), which sends all the five messages in a single one, thereby reducing the communication overhead in execution time and the required software infrastructure by the use of larger messages and by the reduction on the number of channels.

In order to reduce the cost for inter-processor communication, we integrated the Message Aggregation optimization technique in our Simulink-based *Multithread code generator*. In this way, our code generation method allows one to amortize the synchronization cost by reduction on the number of messages, thereby reducing the total amount of communication overhead in the execution time. This optimization also decreases the memory size by the reduction of data structures required to represent the communication channels. Figure 4.12 shows the global flow of our *Multithread code generator,* after the integration of Message Aggregation step.



```
T1( ){
    recv (R0,8 ); //recv
    recv (R1,8);
    F1( );
    ...
    send (S0, 8); // send 8B
    send (S1, 8);
    send (S2, 8);
    send (S3, 8);
    send (S4, 8);
}
```

```
T1( ){
    recv (RT1, 40 ); //recv
    F1( );
    ...
    send (ST1, 40); // send 40B
}
```

(a) Fine-grain specification

(c) Code without message aggregation

(b) Fine-grain specification after message aggregation

(d) Code with message aggregation

Figure 4.11: Motivational example

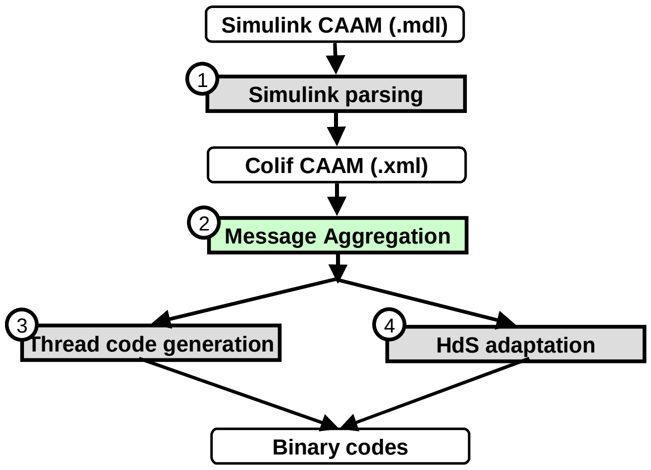
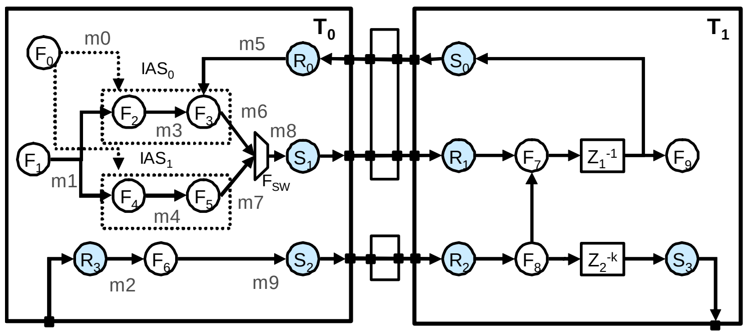


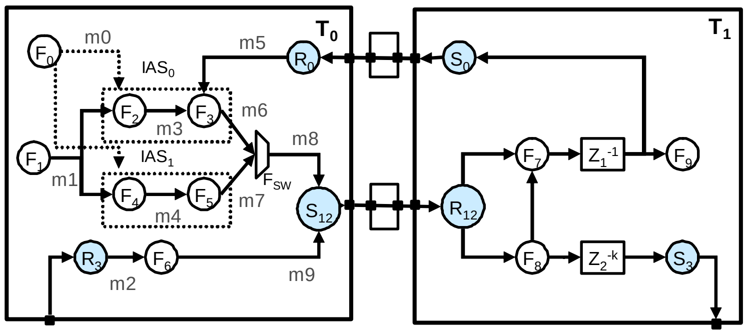
Figure 4.12: Multithread code generation flow after Message Aggregation integration

Message Aggregation traverses the Colif CAAM and merges messages whose source and destination are identical, and with no dependencies between them. Applying Message Aggregation on the Colif CAAM illustrated in Figure 4.13(a), the CAAM illustrated in Figure 4.13(b) is obtained. In this example, the Send nodes *S1* and *S2* in T0 have the same source and destination threads, and then they are merged in a single node (*S12*). As the result, two messages are grouped into one, reducing the start-up cost and the software data structures to perform the data transfer. Similar group operation is performed for the receive nodes *R1* and *R2* in T1, as shown in Figure 4.13(b).

Thread code includes memory declarations for links and behavior codes for nodes in the CAAM. With the integration of the Message Aggregation step, the *Thread code generator* produces memory declarations according to the CAAM resultant of the Message Aggregation step. When Message Aggregation is not applied, a buffer memory is declared for each data link with its data type as line 1 of Figure 4.13(c). Otherwise, a structure is declared to combine all buffer memories connected to the input (output) port of a merged Send (Recv) node. As an example, the data structure *m10* is declared for the merged node *S12* in line 3 of Figure 4.13(d). This structure combines the input buffer memories *m8* and *m9* of node *S12*.



(a) Colif CAAM



(b) Colif CAAM after MA

```
1: char m0[1]; int m1[4];
2: // decl m2,m3,m4, m5, m6, m7
3: int m8[4]; int m9[8];
4: T0 ( ) {
5: while (1){
6:   F0 (m0); F1 (m1);
7:   recv (m5,8);  //R0
8:   if (m0) {
9:       F2(m1,m3); F3(m3,m5,m6); m8=m6;
10:  else
11:      F4(m1,m4); F5(m4,m7);  m8=m7;}
12:  recv (m2,32);  F6(m2, m9);
13:  send ( m8,4);  //S1
14:  send ( m9,32);  //S2
15: } }
```

(c) T_0 Code without MA

```
1: char m0[1]; int m1[4];
2: // decl m2,m3,m4, m5, m6, m7
3: struct {int m8[4]; int m9[8]; } m10;
4: T0 ( ) {
5: while (1){
6:   ...
7:   recv (m5,8);  //R0
8:   if (m0){
9:      F2(m1,m3); F3(m3,m5,m6); m10.m8=m6;
10:  else
11:     F4(m1,m4); F5(m4,m7);  m10.m8=m7;}
12: recv (m2,32);  F6(m2, m10.m9);
13: send (m10,36 );   // S12
14: } }
```

(d) T_0 Code with MA

Figure 4.13: Thread code generation with Message Aggregation

After memory declaration, a behavior code is generated for each thread according to the scheduling result. For communication nodes, the code generator produces communication primitives calls (*send_data/recv_data*), as shown in the line 13 of Figure 4.13(d), where the source for the merged node *S12* is the data structure *m10*. Consequently, the functions that produce data for this merged node use elements of this data structure as output, as shown in line 12 of Figure 4.13(d), where *F6* generates part of the data to be sent for this node. Similarly, the Recv nodes can be also grouped and, in this case, a data structure should be declared to store the received data.

As previously mentioned, Message Aggregation technique reduces software channel structures and consequently, reduces the required data memory size. However, this technique can increase buffer memories. For example, when a Send node (e.g. *S1*) is grouped in two different merged nodes (e.g. *S12* and *S13*). Both of them are connected to different thread destinations, its buffer memory becomes to be duplicated in two data structures, and used for each Send operation. This effect is discussed in the experiment section 4.5.4.

To avoid deadlock, out tool merges Send (or Recv) nodes into another Send (or Recv) node only when all of them have no precedent dependency. Figure 4.14 illustrates the deadlock problem. As the node *R2* has precedent dependency with *R0* in Figure 4.14(a), when both are grouped in the same merged node, a deadlock has occurred, as shown in Figure 4.14(b).

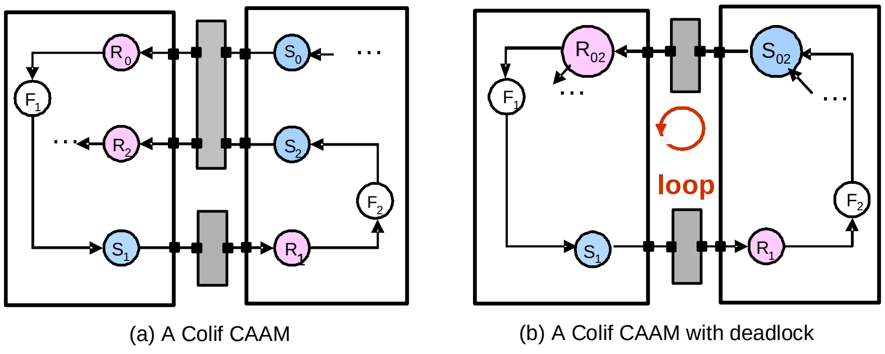


Figure 4.14: An example of deadlock by Message Aggregation

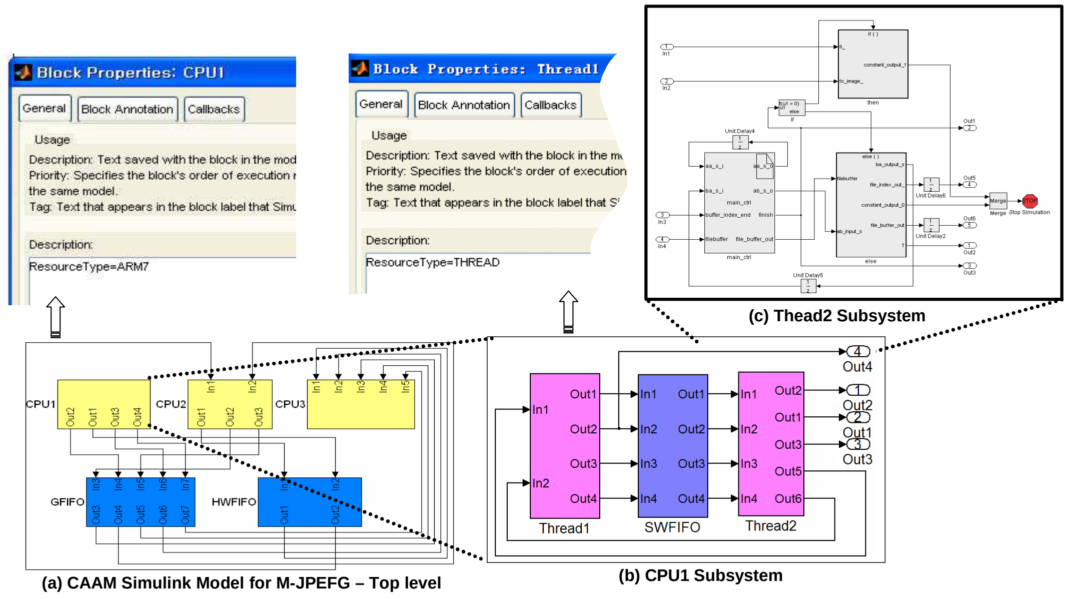
4.5 Experiments

To show the applicability of our software generation flow and the effectiveness of the proposed optimizations, we used two data-intensive applications: Motion-JPEG video decoder and H.264 video decoder. For both applications, we developed a Simulink functional model, and validated their functionalities with Simulink simulation environment. After that, we transformed the Simulink models into Simulink CAAMs according to the chosen platforms. Section 4.5.1 presents the MJPEG and H264 applications and the built CAAMs, while section 4.5.2 presents the used platforms. Memory optimization and Message Aggregation results are presented in section 4.5.3 and 4.5.4, respectively.

4.5.1 Applications description

4.5.1.1 Motion-JPEG video decoder

M-JPEG decoder decodes a bit stream encoded by JPEG still-image compression algorithm. From reference C code, we developed a Simulink application model, which has 7 S-Functions (user-defined blocks), 7 delays, 26 data links, and 4 if-action-subsystems. From this Simulink application model, a Simulink CAAM was built using Simulink graphic interface. Figure 4.15 illustrates the built CAAM. This model contains one ARM7 and two Xtensa CPU subsystems communicating through one GFIFO and one HWFIFO, as shown in Figure 4.15(a). CPU1 subsystem contains two threads communicating through software FIFO, as shown in Figure 4.15(b). Figure 4.15(c) shows the Thread2 subsystem, which is composed of Simulink blocks and links. These figures are presented in detail in Appendix B.


Block Properties: CPU1

Usage

Description: Text saved with the block in the mod

Priority: Specifies the block's order of execution r

the same model.

Tag: Text that appears in the block label that Simu

Description:

ResourceType=ARM7

Block Properties: Thread1

Usage

Description: Text saved with the block in the mo

Priority: Specifies the block's order of execution

the same model.

Tag: Text that appears in the block label that S

Description:

ResourceType=THREAD

Figure 4.15: Simulink CAAM for Motion-JPEG decoder

4.5.1.2 H264 video decoder

The H.264/AVC video coding standard has been developed and standardized collaboratively by both the ITU-T VCEG and ISO/IEC MPEG organizations (WIEGAND, 2003). In our experiment, we used an H.264 decoder, which is based on the Baseline Profile for video conference and videophone applications.

H.264 decoder receives an encoded video bit stream and iteratively executes macroblock-level functions. They are variable length decoding (VLD), inverse zigzag and quantization (IQ), inverse transform (IT), spatial compensation (SC), motion compensation (MC), reconstruction (REC), and deblocking filter (DF) to construct a video image sequence (WIEGAND, 2003), as illustrated in Figure 4.16.

From the dataflow illustrated in Figure 4.16, a Simulink functional model of the H264 decoder was built. This model includes 83 S-Functions, 24 delays, 310 data links, 43 if-action-subsystems, 5 for-iteration subsystems and 101 pre-defined Simulink blocks. Each functional block of Figure 4.16 consists of one or more S-Functions or pre-defined Simulink blocks. From this functional specification, we built five different CAAM models, varying the partitioning and the number of processors from two to six CPU subsystems. The motivation for that was the exploration of the design space of the H264 video decoder. Section 4.5.3.2 and 4.5.4 show results obtained in this exploration.

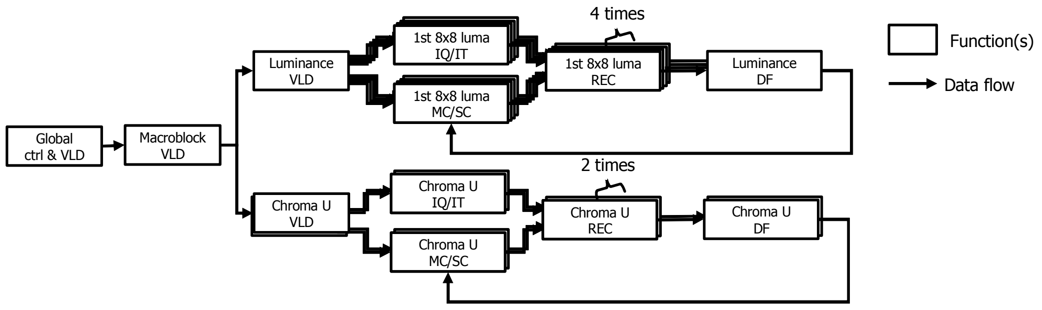


Figure 4.16: H.264 decoder block diagram

4.5.2 Target platform

Each CPU subsystem defined in the CAAM model is composed of Processor, Local Bus, Local Memories, PIC, Timer, Mailbox, and Network Interface (NI). In order to support simulation, the Simulink-based design flow (HUANG, 2007) provides SystemC TLM models for these Hardware components by a component library. This includes instruction-set simulator (ISS) for Xtensa and ARM processors.

The multiprocessor platform architecture is built by *Hardware architecture generator* through instantiation of several CPU subsystems, all connected to a bus. Figure 4.17 shows a platform architecture used for the Motion-JPEG decoder, composed of three CPUs and a global memory. In this architecture, the GFIFO and HWFIFO protocol are used for inter-processor communication.

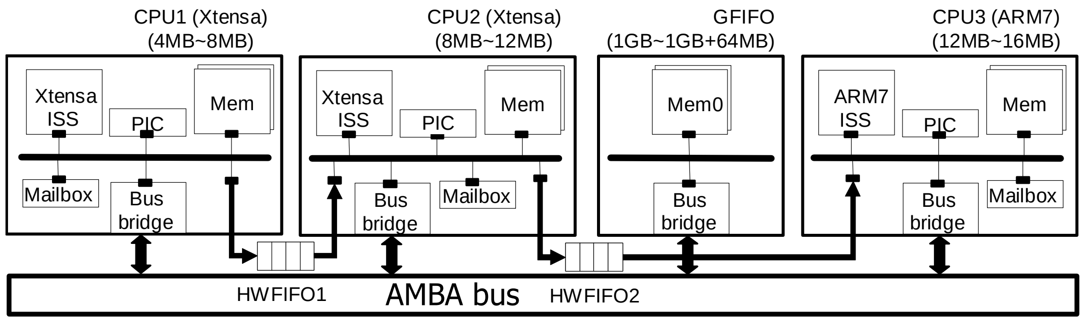


Figure 4.17: MPSoC Platform used for the Motion-JPEG decoder

Similar multiprocessor platforms were built for H264 video decoder. In the experiment with this application, we have modified the number of processors to explore the design space of the H264 decoder and to observe the effects of these optimizations in different MPSoC platforms. These platforms are composed of Xtensa processors communicating through GFIFO channels. At the beginning, we profiled the execution cycle with a single processor system (SS1). We partitioned the Simulink algorithm model and built a Simulink CAAM with two processor subsystems (SS1, SS2) based on the profile result. Similarly, we continued to build Simulink CAAMs by increasing the number of processors from two to six. The different partitioning versions were done manually.

4.5.3 Memory optimization

For checking the effect of memory optimization techniques, we generated seven versions of C codes for each Simulink CAAM: one single-thread version with Real Time Workshop (RTW), three single-thread codes with the *Multithread code generator,* and three multithread ones with the same generator. Table 4.2 specifies all configurations used in the experiments. We compiled each generated thread code by ARM GNU C compiler and Xtensa C compiler and measured data memory and code memory sizes. In both applications, we mapped the image buffers into a global memory and we traced only on-chip memory that heavily affects on the chip area and cost. Memory optimization results obtained for Motion-JPEG and H264 are presented in section 4.5.3.1 and 4.5.3.2, respectively. Besides the memory size, performance obtained for the generated codes are also presented in these sections to show the impact of the proposed memory optimizations on this issue.

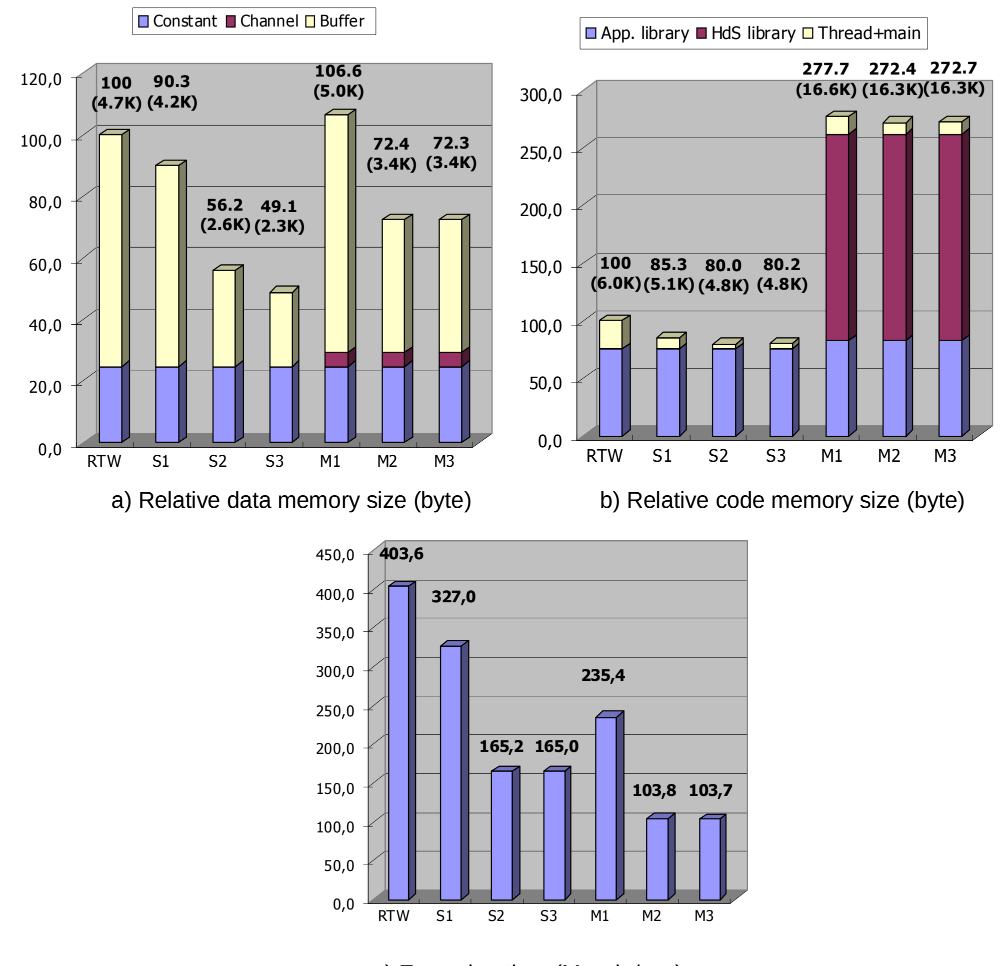
Table 4.2: C code generation with 7 configurations

#	Name	Configuration for code generation
1	RTW	RTW
2	S1	Single-thread without optimization options
3	S2	Single-thread with copy removal
4	S3	Single-thread with copy removal and buffer sharing
5	M1	Multi-thread without optimization options
6	M2	Multi-thread with copy removal
7	M3	Multi-thread with copy removal and buffer sharing

4.5.3.1 Motion-JPEG video decoder

Figure 4.18(a) shows the relative data memory sizes of Motion-JPEG decoder for the seven configurations defined in Table 4.2. In the single-thread case, the data memory is composed of buffer and constant memories. The buffer one represents the memory necessary to implement the Simulink data links, while the constant memory represents the memory for Huffman table in the Motion-JPEG library. Our code generator with full optimization options (S3) reduces the total data memory size by 50.9% compared to RTW. Note that RTW provides only limited memory minimization techniques, so the data memory size of the C code generated with RTW is relatively close to that with our tool without optimization options (S1). In the multithread case, the reduction obtained for configuration M3 compared to RTW is 27.7%. In the multithread case, the reduction obtained for configuration M3 compared to RTW is 27.7%. Notice that, even though the multithread code requires additional buffers and channel memories, it gave such gains against the single-threaded code generated with RTW.

In the single-thread case, one thread and one application library represent the whole implementation code. However, for multithread case, the total code size is increased because it is the sum of all thread codes, main codes, application library, and HdS library. Our memory optimization techniques also reduce the code size as a consequence of using the copy removal techniques. Figure 4.18(b) shows the relative code memory sizes of Motion-JPEG for the seven configurations. Compared to configuration S1, S3 achieves 6.2% of reduction on code size. In multithread case, M3 presents 1.8% code memory size reduction compared to configuration M1. Experiment results show that the proposed memory optimization techniques are effective for multithread code generation, reducing both data and code sizes.



a) Relative data memory size (byte)

b) Relative code memory size (byte)

c) Execution time (Mcycle/sec)

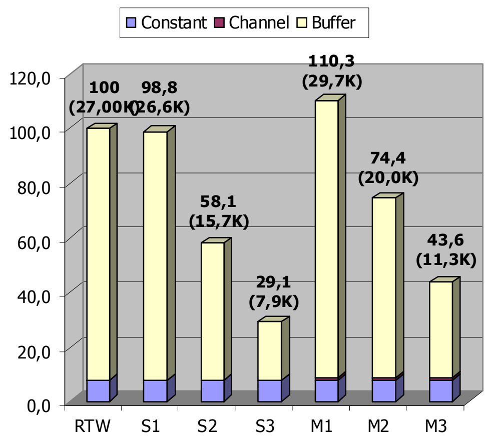
Figure 4.18: Data memory size, code memory size and execution time of Motion-JPEG decoder with single- and three-processor platforms

Multithread multiprocessor solutions are used to achieve better performance. To evaluate the impact on performance, we obtained the number of cycles required to decode 30 frames QVGA Unicycle JPEG stream for each configuration, which are presented in Figure 4.18(c). Regarding copy removal technique, configuration S2 (M2) shows 49.4% (55.9%) execution time reduction compared to S1 (M1). This result shows that copy removal technique improves significantly the performance of the generated code, especially when there are copy operations between large-sized arrays. Compared to RTW, the configuration M3 shows 3.89 times faster performance because of the concurrent execution and the memory optimization, which also impacts in performance. The multithread solution with all optimization options (M3) is 1.60 times faster than single thread one with all optimization options (S3). This result is less than our expectation mainly because two subsystems transfer massive data through global memory using processor load/store instructions, i.e. GFIFO. The required bandwidth is 19.0 MB/sec and the processors averagely spent 53.3% and 25.3% of the run time for computation and communication, respectively. The rest is idle time, waiting for available data or space.

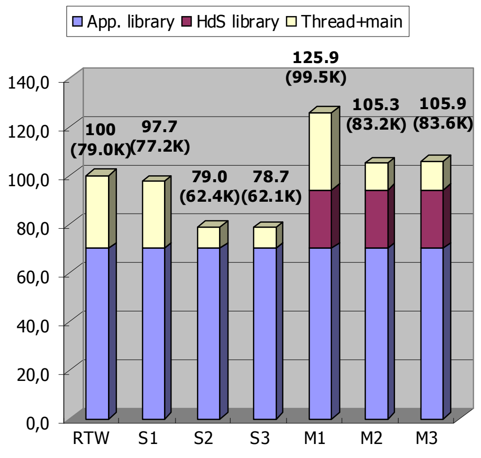
4.5.3.2 H264 video decoder

Firstly, A H264 Simulink CAAM with four CPU subsystems was used to show the effects of memory optimization on the code generated with different tool configuration (see Table 4.2). Figure 4.19(a) shows the relative data memory size, where "Constant" represents VLD tables. In the single-thread case, the configuration S3 achieves 70.9% data memory size reduction compared to RTW. In the multithread case, the code generator with full optimization (M3) reduced the data memory size by 66.7% compared to that without optimization (M1). Regarding code memory size, shown in Figure 4.19(b), configurations S3 (single-thread case) and M3 (multithread case) show 19% and 20% code size reductions compared to S1 and M1, respectively. These results also show the effectiveness of the proposed memory optimization techniques in automatic code generation for both single-thread and multithread cases.

Figure 4.20 presents the performance results obtained from the H264, with four processors for each code generation configuration. It shows the number of cycles required to decode 30 frames QCIF H.264 stream. Multiprocessor implementation with configuration M3 is 2.15 times and 3.04 times faster performance compared to the single-processor one with configuration S3 and to RTW, respectively. The required bandwidth is 12.1 MB/sec, and the processors spent around 63.7% of the run time in computation and 13.7% in communication.



(a) Relative data memory size (byte)



(b) Relative code memory size (byte)

Figure 4.19: Data memory size and code memory size of H.264 decoder with single- and four-processor platforms

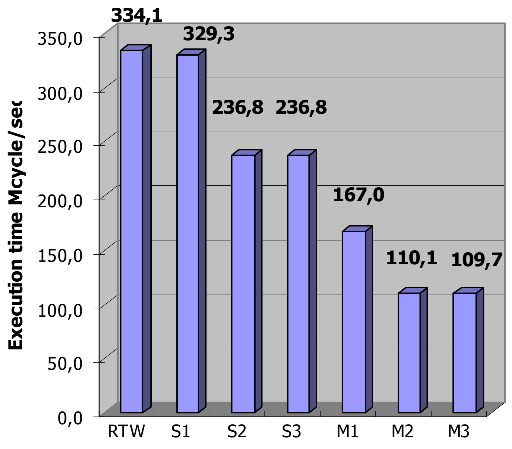
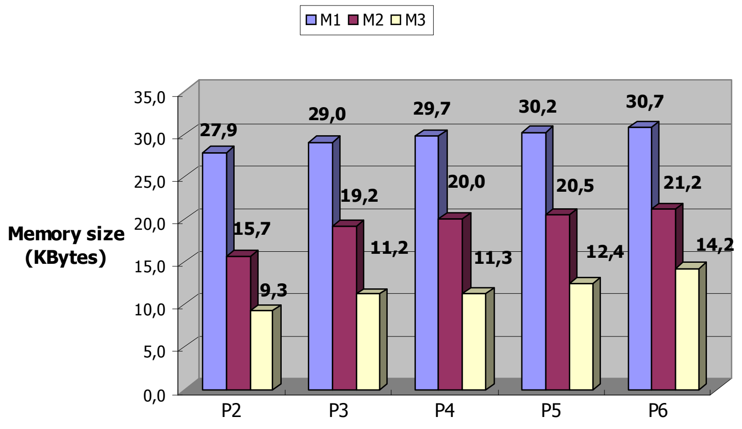


Figure 4.20: Execution time of H.264 with single- and four-processor platforms

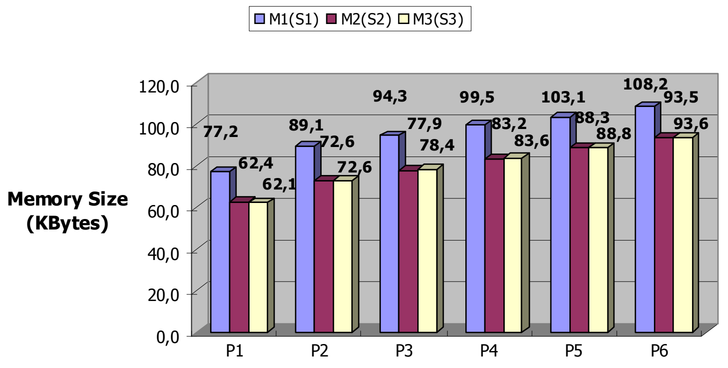
To explore the design space of the H.264 decoder, we designed several multiprocessor platforms by increasing the number of Xtensa processors from two to six. Figure 4.21 presents memory sizes with different numbers of processors. In the figure, P*x* represents a multiprocessor platform with *x* processors, varying from 2 to 6 Xtensa subsystems. Figure 4.21(a) shows data memory sizes obtained varying the number of processors and the configurations options for M1, M2 and M3. It shows that, when the number of processors grows, the data memory size also increases due to the increasing of the number of required channel buffer memories and channel data structures. Regarding code size, similar effect can be observed in Figure 4.21(b), because the number of threads also increases along as the number of processors grows. This, as a consequence, increases the number of line codes.

The performance results obtained for each platform were also evaluated. To obtain performance results, we simulated the execution of the generated codes under the chosen platform (P2-P6) using instances of Xtensa ISS simulator. Figure 4.22 illustrates the number of cycles required to decode QCIF H.264 stream at a frame rate of 30 frames/second for each platform. The multiprocessor platform with six Xtensa subsystems (P6) and configuration M3 (multithread with all optimization options) shows 2.3 times higher performance compared to single processor platform (P1) with configuration S3 (single-thread with all optimization options). We also compared our multiprocessor solutions to a single-processor one and we found that the version P6 achieved 56.4% of performance improvement compared to the single-processor one (236.8 Mcycles/second). From the design space exploration, we found that VLD parts (frame, slice, and macroblock VLD in Figure 4.16) limit the performance because they are sequential, and it does not pay off to add extra processors.

The performance result obtained for the H264 decoder is not appropriate for real systems, where a frame rate of 15 frames/second can be required. It shows that optimizations are necessary in the generated code in order to improve its performance. Observing that a considerable time is spent with communication, we propose here to apply a communication optimization technique to reduce the communication overhead. Section 4.5.4 presents the results obtained with the integration of Message Aggregation in the *Multithread code generator*.



(a) Relative data memory size (Kbyte)



(b) Relative Code memory size (Kbyte)

Figure 4.21: H.264 decoder data memory size and code memory size with different memory optimization configurations and different number of processors

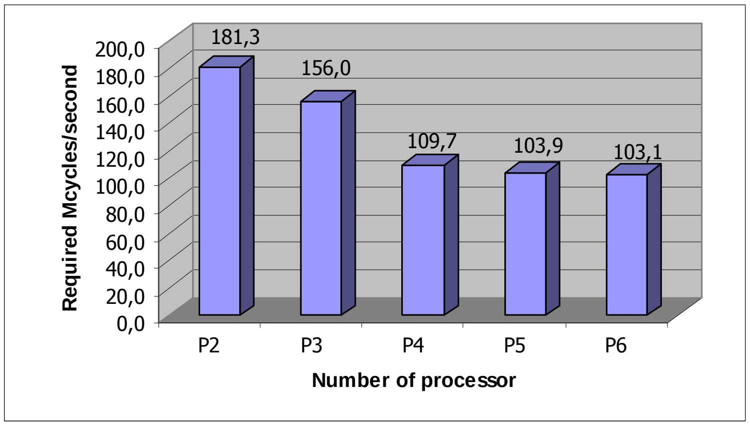


Figure 4.22: Execution time of H264 decoder (Mcycles/sec)

4.5.4 Communication optimization

In this section, the H264 video decoder is used as a case study. It shows that performance improvements and memory reductions are achieved when Message Aggregation (MA) technique is integrated in the code generation flow used by the *Multithread code generator*. In this experiment, the same H264 CAAM models with two, three, four, five and six CPU subsystems used in section 4.5.3.2 were also employed. For each one of these CAAMs, we generated code using the *Multithread code generator* and evaluated the performance and the memory improvements achieved when MA is applied during the code generation.

Firstly, we analyze the impact of Message Aggregation on the execution time for the different multiprocessor solutions. Performance results were obtained by simulation of the execution of the generated codes under the chosen platform through the use of Xtensa ISS simulators. In this way, for each version of generated code, we obtained the number of cycles required to decode a QCIF foreman at a frame rate of 30 frames /second.

Figure 4.23 illustrates the performance results for the generated codes for the five different CAAM models (P2-P6), with and without Message Aggregation. The results show that when MA is applied in our code generation flow, the performance increases for all five configurations, with improvements from 14% until 21%. For example, comparing the performance results for P6 with MA and without MA (w/o MA), we found a performance improvement of 21.2% obtained by the Message Aggregation technique. Comparing our multiprocessor solutions with a single-processor one, we found that the P6 version without MA achieved 56.4% of performance improvement, while the configuration P6 with MA achieved 65.7%.

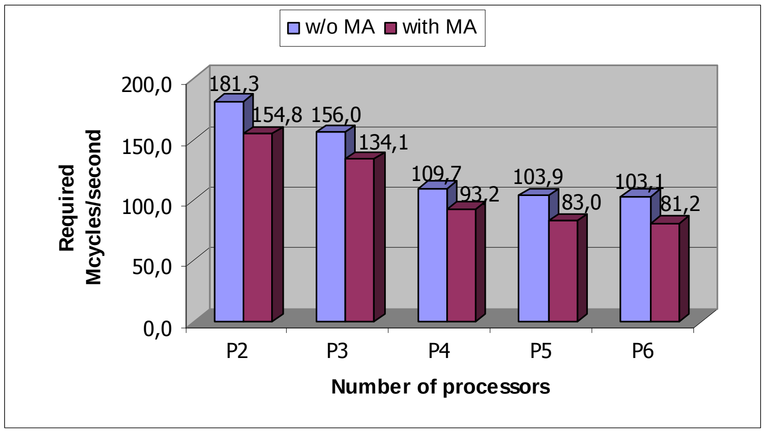


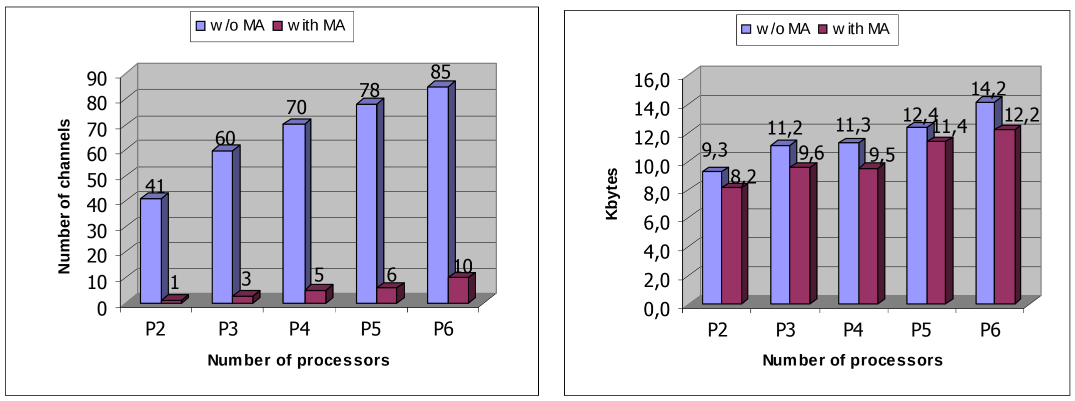
Figure 4.23: Performance results for H264 decoder

In order to analyze this optimization in more detail, we divided all processor operations into three classes of different functions: Computation (Comp), Communication (Com) and Idle. All operations in the application, including computation and some memory access, are defined as computation class. The communication class represents the operations for inter and intra-thread communication. In this class, most of operations are launched by *load* or *store* instructions executed in a processor. Except for Computation and Communication, the remainder operations, which consist of thread switching and waiting for synchronization, are classified as Idle. Table 4.3 shows the percentage of computation (comp), communication (comm) and idle per second of the application execution time and the communication Speed in Bytes/cycle (average for 1cycle) for each multiprocessor platform (P2-P6). These results show that Message Aggregation decreased the time spent with communication and accelerate the communication for all multiprocessor platforms.

Table 4.3: Computation, Communication and Idle time of H264 decoder with different number of processors

	with MA				w/o MA			
	comp	comm	idle	speed	comp	comm	idle	speed
P2	76%	3.7%	19.9%	0.65B/s	70.5%	12.5%	17%	0.17 B/s
P3	59%	5.2%	35.7%	0.56 B/s	55%	14.7%	30.3%	0.17 B/s
P4	64%	7.4%	28.4%	0.57 B/s	58.7%	18.6%	22.6%	0.19 B/s
P5	57%	7.4%	34.9%	0.56 B/s	49.9%	17.4%	32.6%	0.19 B/s
P6	44%	7.4%	47.8%	0.49 B/s	48.6%	19.2%	44.2%	0.15 B/s

Secondly, we analyzed the impact of the Message Aggregation in the number of required communication channels. Figure 4.24(a) illustrates the effect of this technique for the different configurations of the H264 model (P2-P6). The results show that Message Aggregation achieved a reduction on the number of inter-processor channels of around 90% for all configurations. For example, in the case with four CPU subsystems (P4), the achieved reduction is from 70 to 5 channels (92.8%). These reductions depend on the granularity of each block that composes the Simulink model and the chosen partitioning. The reduction on the number of channels impacts on the software infrastructure required for communication, reducing data memory size. Figure 4.24(b) shows the results for data memory size obtained for the five versions of the H264 CAAM. These results show a reduction of 15.9% and 14% in the four CPUs (P4) version and in the six CPUs (P6) version, respectively, when Message Aggregation is applied.



a) Reduction on the number of channels   b) Reduction on the data memory size

Figure 4.24: Reduction on the number of channels and on the data memory size

Table 4.4 shows the data memory size of the generated code for four CPUs (P4). As it is a multiprocessor solution, the data memory is composed of Constant, Buffer and Channel memories. The constant memory represents constant tables such as VLD table used in the decoding algorithm. The buffer memory represents the memory required to implement the Simulink data links. At last, the channel memory represents the channel data structures required to promote the communication. The results show that Message Aggregation can achieve a large reduction on the data structures used to manage channels (channel in Table 4.4), e.g. 92.8% in the case of version P4, by the reduction on the number of required channels. It means a reduction of 14% in the total data memory size. Note that the required buffer memories increase by 17% with Message Aggregation. The reason for this small increase is briefly explained in section 4.4.

Table 4.4: Data memory size in bytes for the solution P4

	Without MA	**With MA**
Constant	2172	2172
Channel	3360	240
Buffer	6006	7320
Total	**11538**	**9732**

In addition, MA also improves code size by the reduction on the lines of code required to declare and initialize channels and to invoke communication primitives in Main and Thread codes. As in this experiment, these codes represent a small part of the total code size, which also includes HdS and application libraries, this improvement is too small. In case of P4 version, where Thread and Main codes represent only 11.5% of the total code size, MA achieves a reduction of only 0.5% of the total code size. Regarding only Thread and Main codes, a reduction of 4.4% was observed.

4.5.5 Experiment analysis

Our *Multithread code generator* extracts necessary information such as number of threads, types of processors, communication channels from the input Simulink CAAM, and then produces a set of software binaries, each of which executes on a target processor. Consequently, our multithread code generator can avoid the designers to do laborious programming work.

In addition, from the experimental results, the effectiveness of the proposed memory optimization techniques integrated in our multithread code generator was shown. The data memory with all optimization options was 34.3% less for a Motion-JPEG decoder with three processors and 68.0% less for an H.264 decoder with four processors than that without optimizations. We can achieve more memory reduction in the H.264 decoder than in Motion-JPEG decoder because a H.264 decoder includes a relatively larger number of buffers with disjoint lifetimes. Our memory optimizations also impact the code size, reducing the application code size in 19.4% and 15.8% for H.264 decoder single-thread and multithread cases, respectively. More results for the design exploration of these applications can be found in (HUANG, 2007).

Moreover, experimental results show that MA can achieve a large reduction on the number of inter-processor data transfers for a fine-grain system specification. However, this optimization cannot achieve proportional reduction on the number of cycles required to process one macroblock. One reason for this is because MA can increase the message latency in some cases, thereby decreasing performance. In terms of data memory size, MA presents a reduction of around 14%. Compared to H264, the Motion-JPEG is a simple algorithm and has a very small number of channels. This is the reason for the Message Aggregation technique could not achieved a large performance improvement for this application, and then we do not present the Motion-JPEG results here.

However, the performance of the presented multiprocessor platforms is still not enough for real systems. For example, the digital video broadcasting system requires H.264 QVGA decoding with a frame rate of 15fr/sec, which is about one and a half times faster than the platform with four processors at 93.2 MHz for QCIF 30fr/sec decoding. The QVGA format is about three times larger than QCIF format. The platform is pure software approach and thus its performance is somewhat limited to process data-intensive applications. In order to achieve the required performance, we need to adopt multiprocessor platforms with configurable processors such as Xtensa with customized instructions to specific applications (TENSILICA, 2006). Moreover, it is important to develop a communication architecture that can efficiently handle high-rate data with large-latency wires to implement the high-performance heterogeneous MPSoCs.

Currently, we analyze the effect of Message Aggregation in the inter-processor communication using the GFIFO protocol, which is easy to implement both in hardware and in software. Experiments with other communication protocols will be considered as future work.

5 INTEGRATION OF UML AND SIMULINK

UML was defined in the software engineering domain and is by far the most-used modeling notation for conventional computational systems. The comparison between UML and Simulink presented in chapter 2 shows that UML presents some advantages for requirements specification and represents a higher abstraction level when compared to Simulink. Moreover, UML provides all benefits from the OO paradigm i.e. modularity, encapsulation, and reusability. However, using UML-based tools, designers are asked to write code for some methods in order to obtain the complete application code. In addition, although some efforts to extend UML, it continues to be not well suitable to model dataflow systems.

On the other side, Simulink supports dataflow and continuous time, and the whole code can be automatically generated from a Simulink model. Real-time Workshop (RTW) can be used to automatically generate sequential code from a Simulink model. In addition, the Simulink-based code generation approach proposed in chapter 3 can be used to generate multithread code targeted to an MPSoC architecture from the Simulink CAAM, which combines algorithm and architecture.

UML and Simulink present advantages for the embedded software development, which motivates researchers to find a way to simultaneously exploit the benefits of both. Recent efforts show that both languages are considered attractive for Electronic system-level design (BOLDT, 2007) (SANGIOVANNI-VICENTELLI, 2006) (BRISOLARA, 2005b). Reichmann (2004) proposes the integration of different models in a same design flow. In another effort to integrate Simulink and UML, the Rhapsody UML2.0 tool has been integrated with Matlab/Simulink, allowing the building of UML mixed models which can have modules described in Simulink (BOLDT, 2007). This allows the use of Simulink resources to describe signal processing algorithms and simulation of heterogeneous models that can include physical models like a plant, while at the same time UML is used for requirements specification. Both approaches focus on the use of different modeling languages to specify each system module.

However, we believe that UML is the preferred language for software engineers, and that it could be interesting to use UML as a single language for initial specification. In this context, we propose a way to integrate UML and Simulink in a single design flow, where UML is used to model whole system and other models can be obtained from UML diagrams by model transformation in order to allow the use of different code generation approaches for each system modules. The UML-based code generation can be used to generate code for event-based (control-flow) modules, using available commercial tools that generate code from state diagrams or FSM models. On the other hand, Simulink-based strategies can be used to generate code for the dataflow modules. Besides that, the same UML model can be reused for different code generation strategies to generate code for different platforms. To support this, mappings from UML to Simulink and to FSM are required. Figure 5.1 illustrates the proposed design flow for embedded software development.

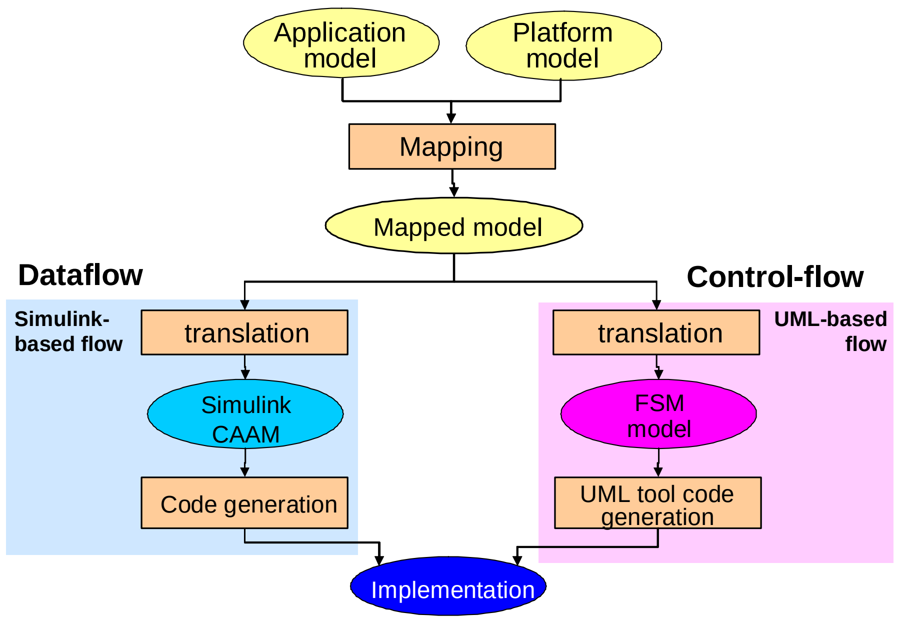


Figure 5.1: Proposed flow for embedded software development

We also propose the use of UML as front-end for the Simulink-based design flow, allowing one to exploit the benefits of UML, while generating executable code for MPSoC from high-level models (BRISOLARA, 2007b). This way, one can avoid the use of Simulink graphical user-interface to build of the Simulink CAAM required for the proposed multithread code generation, which can be an error-prone task.

To support the proposed software development flow, a model transformation mechanism was defined in (BRISOLARA, 2007b). Figure 5.2 illustrates the proposed flow defined to capture UML and transform it in other modeling language notation. This flow has two main steps and its input is an UML model built using an UML editor tool. So, the first step is made by the designer using an UML tool graphical interface. In the second step, the UML model is traversed to find constructions that can be directly mapped to the target modeling language e.g. Simulink, which is defined in a meta-model. According to the mapping rules, the UML model is translated to the target language, as a model-to-model transformation. In order to be flexible, technologies for model transformation, such as smartQVT (SMARTQVT, 2007) and ATL (ECLIPSE DEVELOPMENT TEAM, 2007), should be used to promote this translation. This step produces another XML file that follows the target language meta-model, which can be Simulink or FSM, as illustrated in Figure 5.2.

The third and fourth steps shown in the proposed flow are specifically tailored to the generation of a Simulink model from an UML one. The third step receives as input the model resulting from the model-to-model transformation, which follows the Simulink meta-model semantic, and performs some optimizations before generating the final Simulink model. After that, from the optimized model, an *mdl* file is generated using model-to-text transformation in the fourth step. Although we have focused on generating the Simulink model from an UML one the proposed transformation approach can be extended to support the mapping to other languages, such as UML state diagrams, other FSM-like languages, or KPN.

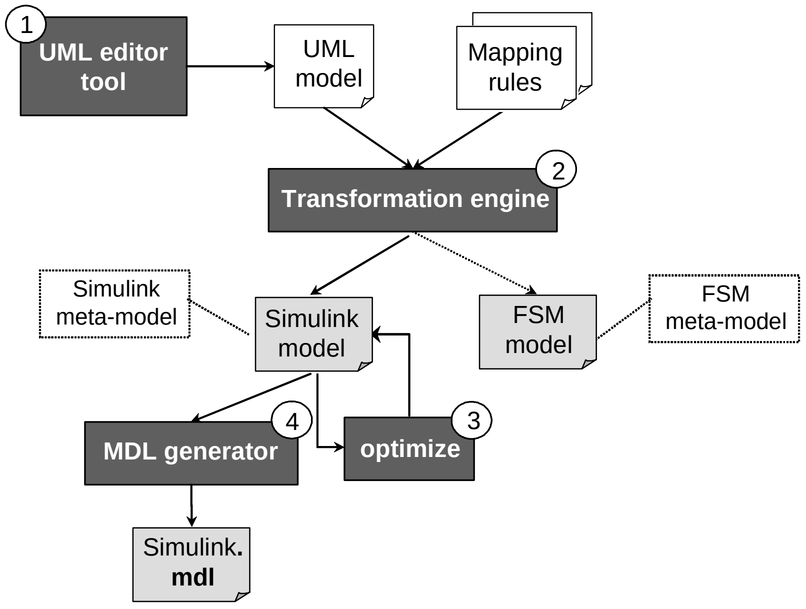


Figure 5.2: Flow for the proposed model transformation

To show the feasibility of our proposal, we defined mapping rules able to transform an UML model in a Simulink CAAM model used as input for the multithread code generation. Section 5.1 explains the proposed mapping. In addition, a prototype was developed and experiments were performed using this prototype, which are presented in section 5.2 and 5.3, respectively.

5.1 Proposal of mapping from UML to Simulink CAAM

When the Simulink-based MPSoC design flow presented in chapter 3 is used, the Simulink CAAM is built manually by a Simulink GUI Interface. From the Simulink functional model, the designer partitions functions into tasks and groups them into different subsystems, thus defining threads and mapping them to processors. To maintain UML high abstraction capabilities and eliminate the necessity of manually building the Simulink CAAM, we propose the mapping from UML to Simulink CAAM. It allows software engineers to employ UML to model the system, which is their preferred language, besides giving them high abstraction. The use of the proposed mapping avoids the necessity of building or modifying Simulink models directly, which means abstracting low-level details like signals and ports.

The proposed mapping can be applied in the flow illustrated in Figure 5.2, allowing one to automatically generate a Simulink CAAM from an UML model. Then, multithread code can be generated from that. As shown in Figure 5.2, to apply the proposed model transformation, the target language needs to be defined as a meta-model. We defined a meta-model for the Simulink CAAM. This meta-model is similar to another Simulink meta-model already published in (NEEMA, 2003), differing mainly regarding the constructions only required in the CAAM. As the Simulink CAAM is an extension of the default Simulink model, the proposed mapping and the proposed meta-model can be used to generate both conventional and CAAM Simulink models.

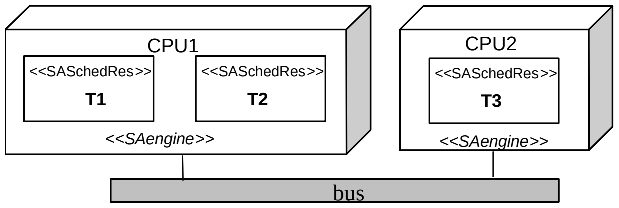
The proposed mapping uses information from the UML deployment and sequence diagrams to obtain the Simulink CAAM. Following our approach, a sequence diagram must be defined for each thread that composes the system. Both diagrams are used in the mapping in order to capture the necessary information to generate the Simulink CAAM. Besides the sequence diagrams, activity diagrams could also be used to detail the behavior of complex algorithms. A didactic example is used here to explain the proposed mapping. Figures 5.3(a) and (b) depict the deployment diagram and sequence diagram for the T1 and T2 threads, respectively. After apply the mapping, the Simulink CAAM shown in Figure 5.3(c) is obtained.

From the deployment model, the definition of the threads that compose the system is captured, as well as the mapping of these threads to processors. In our proposal, processors and threads are indicated by the <<*SAengine*>> and <<*SAschedRes*>> UML-SPT stereotypes, respectively, as illustrated in Figure 5.3 (a). For each processor, a Simulink hierarchical subsystem is created in the CAAM model representing a CPU subsystem (*CPU-SS*), as can be observed in Figure 5.3(c). For each thread mapped to a processor, a *Thread* subsystem (*Thread-SS*) is created inside the corresponding CPU-SS.

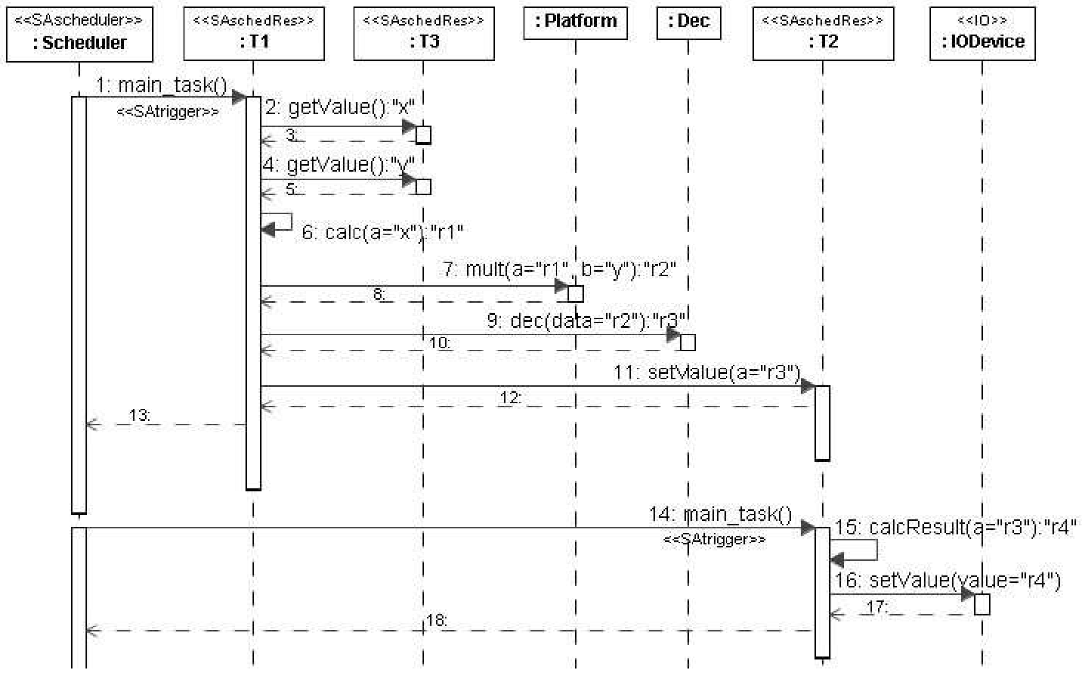
The *Thread-SS* is composed by Simulink blocks that are used to specify its behavior. To capture the thread behavior, these Simulink blocks and the data flow between them must be captured. We propose to capture it from sequence diagrams, once this diagram represents the messages exchanged between objects. For this reason, each thread should have a sequence diagram to describe its behavior in our proposed mapping. The <<*SAtrigger*>> stereotype used in the sequence diagram depicted in figure 5.3 (b) indicates a time event and the invoked method for which the *Scheduler* selects a thread to run.

Method calls in the sequence diagrams are translated to Simulink blocks (user-defined and user-defined blocks) or to communication blocks in the Simulink CAAM. When a method of a passive object is called from a thread, a Simulink block is instantiated. To use pre-defined Simulink blocks, the designer needs to indicate its usage by the invocation of a method from the special object *Platform*. The name of the method needs to be equal to the name of the reused component in the Simulink library. If the method name does not match with the pre-defined component names, a Simulink S-function block is instantiated. An S-Function can have its behavior described in a C code that is compiled and linked to the model. In the example illustrated in Figure 5.3(b), the *dec* and *mul* methods are invoked from the *Dec* and *Platform* objects, respectively, by the thread T1. Notice that in the resulting Simulink, shown in Figure 5.3(c), a *Product* block and an S-function were instantiated in the T1 subsystem.

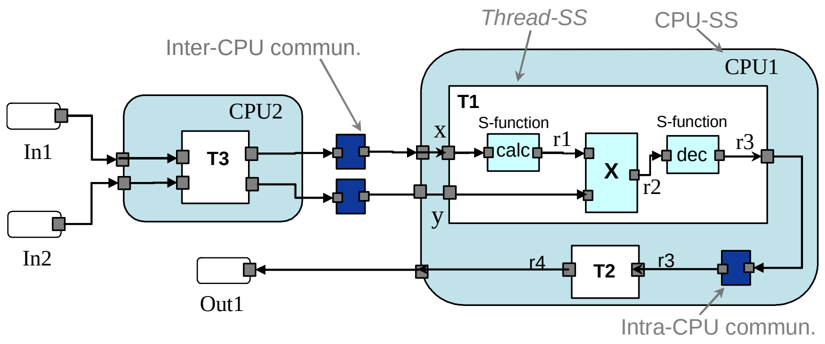
The direction of method parameters (in/out) and the return are used to define input and output ports of subsystems and blocks, and message arguments indicates the connection (data links) between ports of different Simulink subsystems/blocks. The a parameter from *calc* method has the direction set as in, so an input port is created in T1 subsystem, as shown Figure 5.3(c). In the same way, its return is mapped to an output port in T1 subsystem. The *r1* argument is passed as output for *calc* and also is used as input for *mult*, which indicates that the value produced by first is used by the second one and a connection is created between these ports when generating the Simulink model.



(a) UML Deployment diagram



(b) Sequence diagram



(c) Generated Simulink CAAM

Figure 5.3: Example of mapping from UML to Simulink CAAM

When a thread invokes a method from another thread, this indicates a communication between them. In this case, the designer is asked to use a default prefix in the method name, *Set* or *Get*, to indicate send or receive operations, respectively Ports are created in the Thread-SS and an intra-SS or an inter-SS COMM subsystem is instantiated, according to the thread mapping. After that, connections are created between the ports of these subsystems.

In the sequence diagram illustrated in Figure 5.3(b), T1 invokes the method *getValue( )* from T3, which indicates that T1 receives data from T3. As both threads are allocated in different processors, an inter-SS COMM block is instantiated in the Simulink model, as shown in the Figure 5.3(c). The method call *setValue(r3)* in Figure 5.3(b) indicates that the thread T1 sends data to T2. The same argument *r3* is also used by the decode method, indicating that the value produced by this method must be sent to T2. As well as, the output of the decode method must be connected to the T1 output. This communication is translated to an output port in T1 as well as an intra-SS communication channel is instantiated, since both threads are mapped to the same CPU.

To indicate that an object communicates with external systems, we defined a modeling rule. The external system is represented as an object in the sequence diagram decorated with the stereotype *<<IO>>*, which is a new stereotype we have defined. To indicate the reading and writing operations between an object and the IO object, methods with the prefix *get* and *set* are used, indicating the message exchange between the two objects. During the mapping, these *get* and *set* methods are mapped to input and output ports for the system. In Figure 5.4, the thread T3 invokes the method *getValue()* from the object sensor that is marked as <<IO>>, which is translated for a system input port in the Simulink CAAM, as shown in Figure 5.3(c). It should be also used in the sequence diagrams for the threads T2 to generate the output system port shown in the correspondent Simulink CAAM.

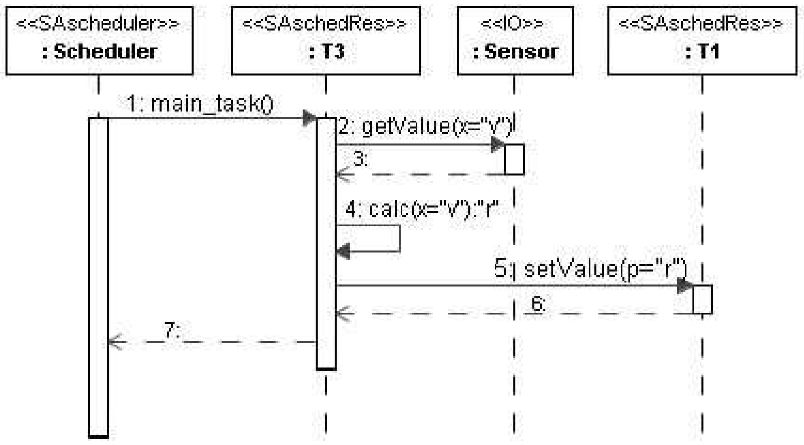


Figure 5.4: Sequence diagram for thread T3

The deployment diagram defines the number of processors and threads. Thus, to build this diagram, the designer is asked to partition the system in threads and define the mapping of threads to processors. We propose the automation of the thread mapping decision by the use of an optimization algorithm that can determine the number of required processors and the mapping of threads to the processors. The use of this optimization can make the deployment diagram unnecessary and, therefore, only the sequence diagram can be considered compulsory to generate the Simulink CAAM from an UML model. To validate the proposed mapping, a prototype was developed, which is detailed in section 5.2.

5.2 Prototype

We developed a prototype that implements the mapping proposed in section 5.1. Figure 5.5 shows the flow used in this prototype, where the input is an UML model. The first step of the flow is the building of the UML model using MagicDraw or other EMF/UML2 compliant tool. After that, a XML file is obtained for the UML model. During the second step, the UML model is traversed and translated to a Simulink model. This step produces another XML file, which follows the Simulink CAAM meta-model. In this prototype, this transformation was implemented in Java using the API provided by the Eclipse EMF, according to the required mapping rules described in section 5.1. The third step has as input the resulting Simulink CAAM model represented using the E-core format (XML-like) and performs some optimizations before generating the final Simulink CAAM model. These optimizations are detailed in section 5.2.1. After that, from the resulting model, we generate a file that follows the *mdl* format used as input in the Simulink environment.

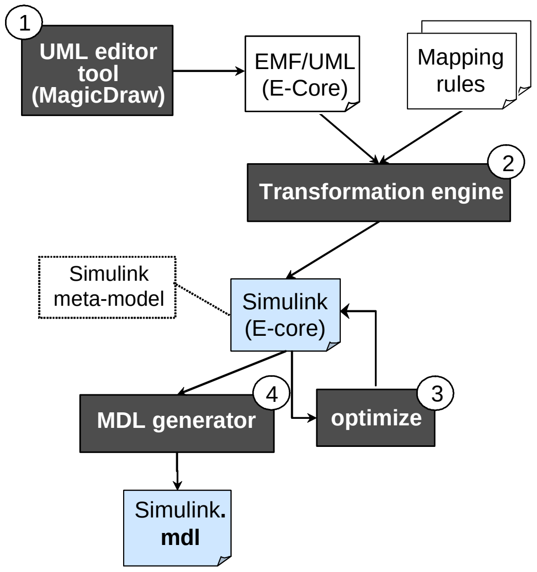


Figure 5.5: Prototype for the mapping from UML to Simulink

5.2.1 Model optimization

During the optimization step, our tool can perform three kinds of optimizations: inference of communication channels, loop detection, and thread grouping. The inference of communication treats of the instantiation of communication blocks in the Simulink CAAM when in a sequence diagram there are method invocations between different threads. In this case, the tool captures the kind of communication (inter-SS COMM or intra-SS COMM) and set the appropriated protocol. When a variable is used as input and output of a function, we have a cyclic path (or loop). In a Simulink model, to avoid deadlock, one needs to insert a temporal barrier (*Delay*) to guarantee that a valid value is available for the input function. The tool looks for cyclic paths in the model and inserts temporal barriers in the generated Simulink model. Furthermore, our tool analyzes the model and groups threads whenever possible, in order to reduce the communication overhead. The proposed optimizations are detailed in the section 5.2.1.1, 5.2.1.3 and 5.2.1.4.

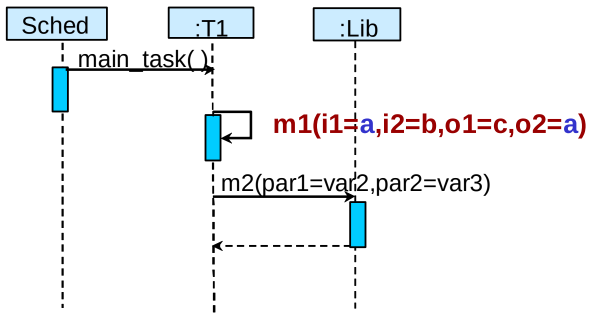
5.2.1.1 Inference of communication channels

In the Simulink CAAM, the communication is explicitly defined and represented by communication channels that can be either inter-SS or intra-SS. To capture these channels from the UML model, we use information from the sequence diagrams and from the deployment diagram or from the result of the grouping thread algorithm. When the communicating threads are in different CPUs, an inter-SS channel is required. Otherwise, an intra-SS channel is instantiated.

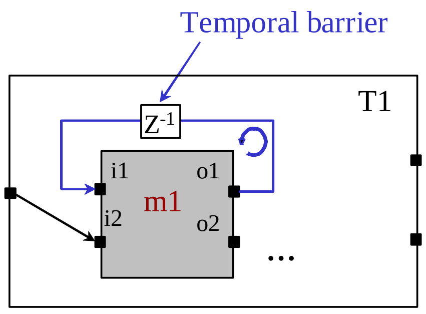
The communication protocol is indicated explicitly in the Simulink CAAM using a specific block parameter. At present, we use only two different communication protocols, the SWFIFO for intra-SS channels and the GFIFO for inter-SS channels. Our tool determines the type for each communication channel and sets their parameters. These protocols are detailed in chapter 4. In the future, different communication protocols can also be supported. In the example illustrated in Figure 5.3, T1 sends data to T2 and an intra-SS channel was instantiated to build the Simulink CAAM shown in Figure 5.3(c), since both threads were allocated in the same CPU-subsystem.

5.2.1.2 Insertion of temporal barriers

When describing a dataflow model, cyclic paths need to be found and temporal barriers are required to avoid deadlocks. In this step, the Simulink model obtained from the translation (step2) is searched for cyclic paths. Simulink Delay blocks are then inserted in the resulting Simulink model. Two different cases of cyclic path can be found. In the case 1, the output of a functional block is connected to its input, as shown in Figure 5.6. In the case 2, the cyclic path is between different sub-systems or different hierarchical levels, as shown in Figure 5.7. Our tool automatically detects these cases and inserts temporal barriers to avoid deadlock. To represent a temporal barrier, a Simulink Delay block is inserted in the data link where the loop is detected.



(a) UML sequence diagram



(b) Simulink model

Figure 5.6: Example of insertion of delay – case 1

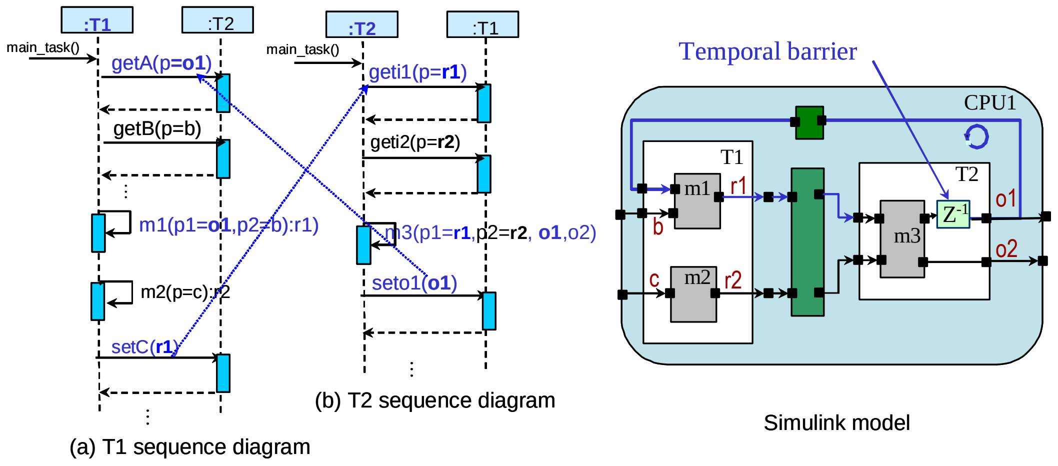
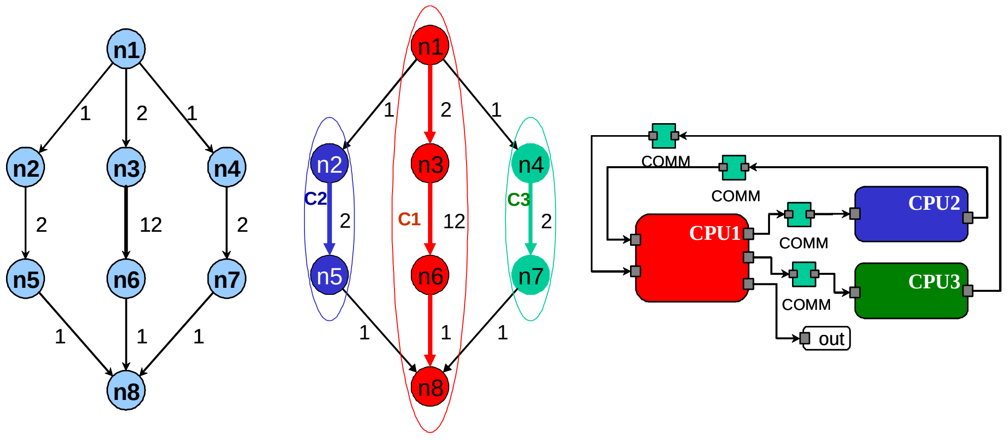


Figure 5.7: Example of insertion of delay – case 2

5.2.1.3 *Grouping threads*

This optimization allocates threads with data dependencies to the same processor, in order to reduce the inter-processor communication. When this optimization is applied, the deployment diagram is not necessary to generate the Simulink CAAM. To observe the data dependency between threads, we use the information captured from the sequence diagrams. This information is used to build a task graph. In this graph, the nodes are threads and the edges have a cost that is determined by the size of data multiplied by the number of transferred data, as illustrated in Figure 5.8(a).



a) Task graph used as input b) Task graph after grouping c) Simulink CAAM: top-level

Figure 5.8: Example of the thread allocation by the linear clustering algorithm

This optimization was implemented in our prototype and the used algorithm is based on Linear Clustering. Figure 5.9 shows the pseudo code of this algorithm. It evaluates the costs for the edges in the graph, grouping threads with more data dependencies. Threads grouped into the same cluster are allocated to the same processor. Figure 5.8 illustrates an example, where 5.8(a) shows a thread graph and 5.8(b) shows the resulting graph after running the optimization algorithm. The resulting graph shows how the eight threads were grouped in three different clusters, indicating that three processors will be used. In this example, as the nodes *n2* and *n5* are in the same cluster, these threads will be allocated to the same processor.

This optimization algorithm is used to optimize the mapping of threads to processors. The result of this optimization step is used to generate the top-level description of the Simulink CAAM, where processors are connected through inter-SS COMM blocks, as shown in figure 5.8(c). This step is optional, and when the designer wants to decide the mapping by himself, information from the deployment diagram can be used to generate the Simulink CAAM top-level, instead using the result of the linear clustering.

```
1. Choose the heaviest edge;
2. If nodes n3 or n6 are not taken
        1. Add nodes n3 and/or n6 to cluster C1;
3. Find incoming edges of node n3;
4. Choose the heaviest edge of step 3;
5. If node n1 is not taken
        1. Add node n1 to C1;
6. Find outgoing edges of node n6;
7. Choose the heaviest edge of step 6;
8. If node n8 is not taken
        1. Add node n1 to C1;
9. Repeat steps 1-8 while possible;
10. Store cluster C1 and create a new one;
11. Goto step 1;
12. Stop when every node has a cluster;
```

Figure 5.9: Pseudo code of the used linear clustering algorithm

It is interesting to note that this algorithm allocates all threads that are in the system critical path to the same processor. This is a good practice to reduce the communication cost, once the cost for intra-CPU communication is lower than the cost for communication between different CPUs (inter-SS COMM).

5.3 Case study

Two case studies are used to validate the proposed mapping and the built prototype. They are the crane control system and a synthetic example, presented in section 5.3.1 and 5.3.2, respectively.

5.3.1 Crane control system

The crane control system, proposed in (MOSER, 1999) and used as case study in chapter 2, shows the capabilities to capture a dataflow from an UML model and the generation of the corresponding Simulink CAAM. In addition, we also show that our tool can automatically insert the required temporal barriers in the generated Simulink model.

The UML model for the Crane control algorithm was developed, which is a module of the Crane system used in chapter 2. In this experiment, we partition the system in three threads, each one specified using UML sequence diagrams. We have decided to map the three threads to the same processor, as shown in the deployment diagram illustrated in Figure 5.10. The grouping algorithm is not applied for this example. Figure 5.11, 5.12, and 5.13 illustrate the sequence diagram for the thread T1, T2 and T3, respectively, from which a Simulink dataflow diagram can be obtained using our rules. Figure 5.13 is not well presented here, due to the limited space. Therefore, this figure is presented in an expanded way in Appendix B.

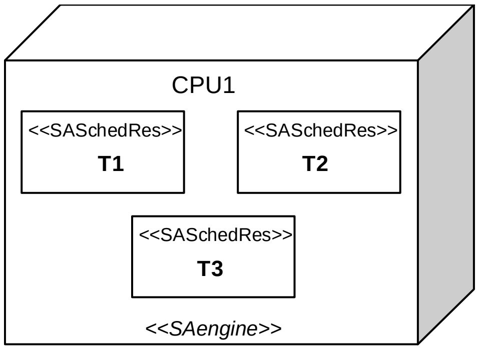


Figure 5.10: Crane system: UML deployment model

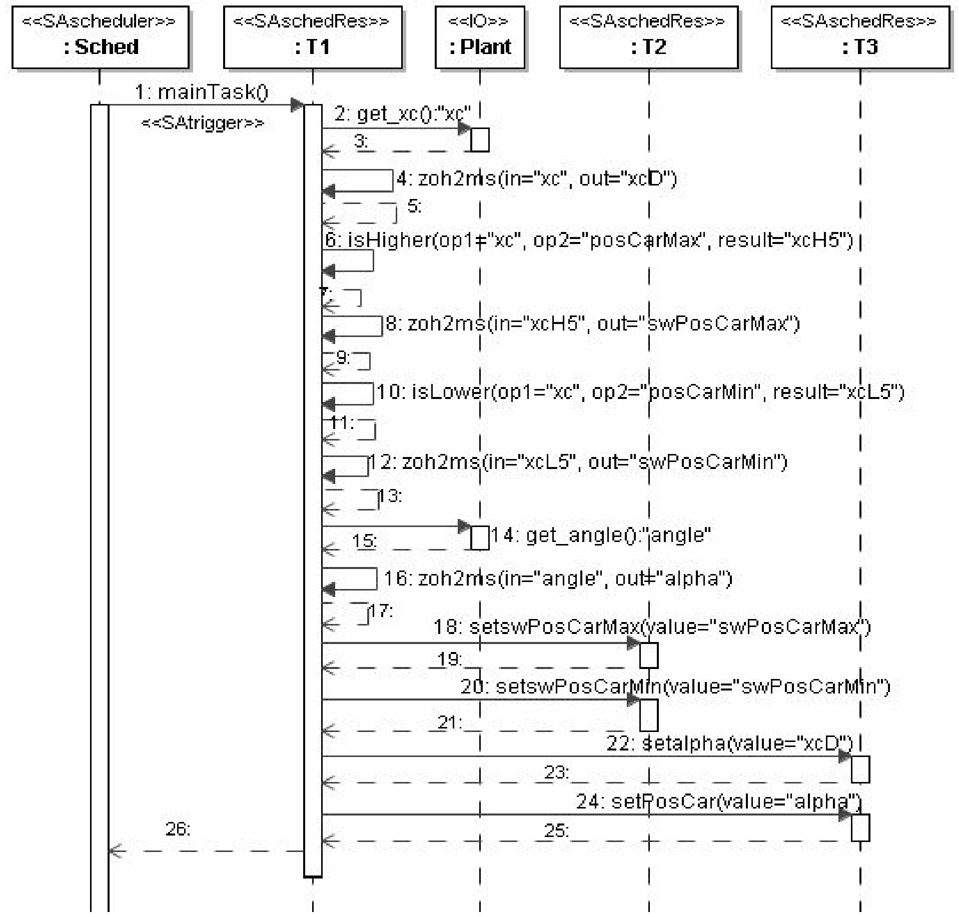
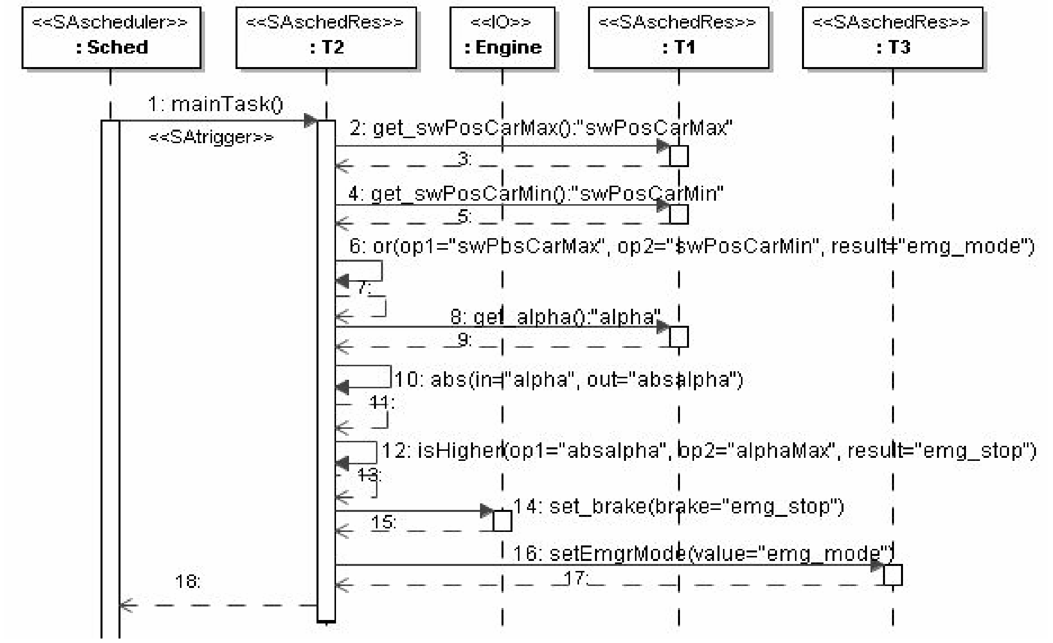
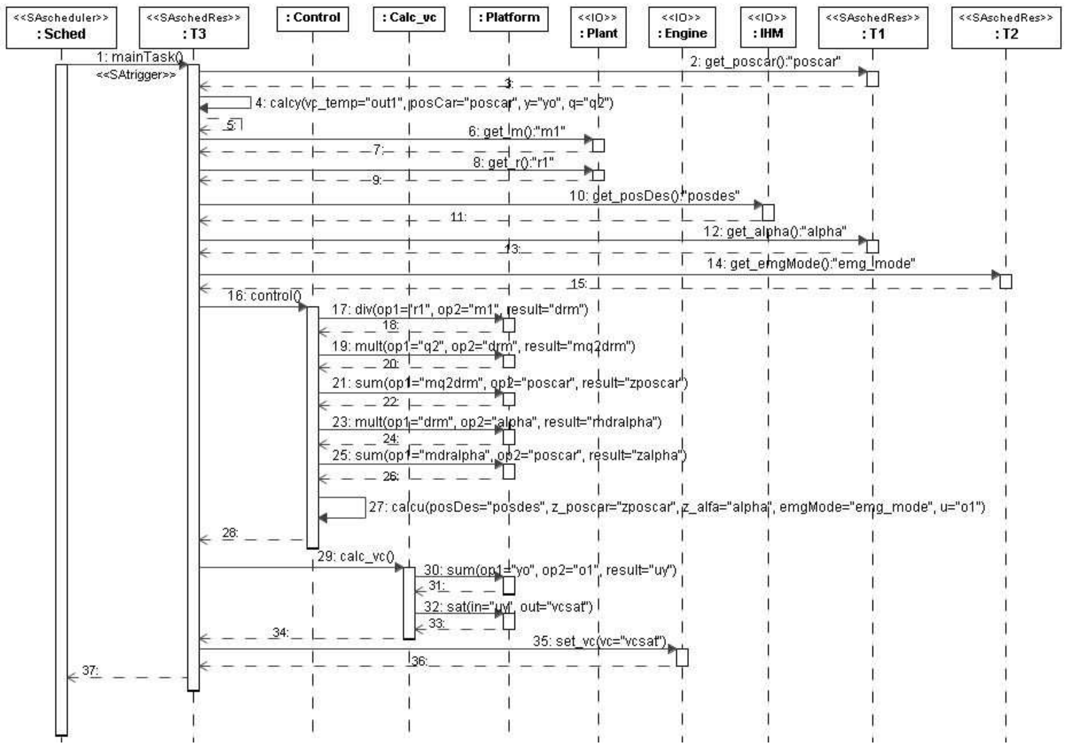

<<SAscheduler>> : Sched
<<SAschedRes>> : T1
<<IO>> : Plant
<<SAschedRes>> : T2
<<SAschedRes>> : T3
1: mainTask()
<<SAtrigger>>
2: get_xc():"xc"
3:
4: zoh2ms(in="xc", out="xcD")
5:
6: isHigher(op1="xc", op2="posCarMax", result="xcH5")
7:
8: zoh2ms(in="xcH5", out="swPosCarMax")
9:
10: isLower(op1="xc", op2="posCarMin", result="xcL5")
11:
12: zoh2ms(in="xcL5", out="swPosCarMin")
13:
14: get_angle():"angle"
15:
16: zoh2ms(in="angle", out="alpha")
17:
18: setswPosCarMax(value="swPosCarMax")
19:
20: setswPosCarMin(value="swPosCarMin")
21:
22: setalpha(value="xcD")
23:
24: setPosCar(value="alpha")
25:
26:

Figure 5.11: Crane UML model: T1 sequence diagram



<<SAscheduler>> : Sched
<<SAschedRes>> : T2
<<IO>> : Engine
<<SAschedRes>> : T1
<<SAschedRes>> : T3
1: mainTask()
<<SAtrigger>>
2: get_swPosCarMax():"swPosCarMax"
3:
4: get_swPosCarMin():"swPosCarMin"
5:
6: or(op1="swPosCarMax", op2="swPosCarMin", result="emg_mode")
7:
8: get_alpha():"alpha"
9:
10: abs(in="alpha", out="absalpha")
11:
12: isHigher(op1="absalpha", op2="alphaMax", result="emg_stop")
13:
14: set_brake(brake="emg_stop")
15:
16: setEmgrMode(value="emg_mode")
17:
18:

Figure 5.12: Crane UML model: T2 sequence diagram



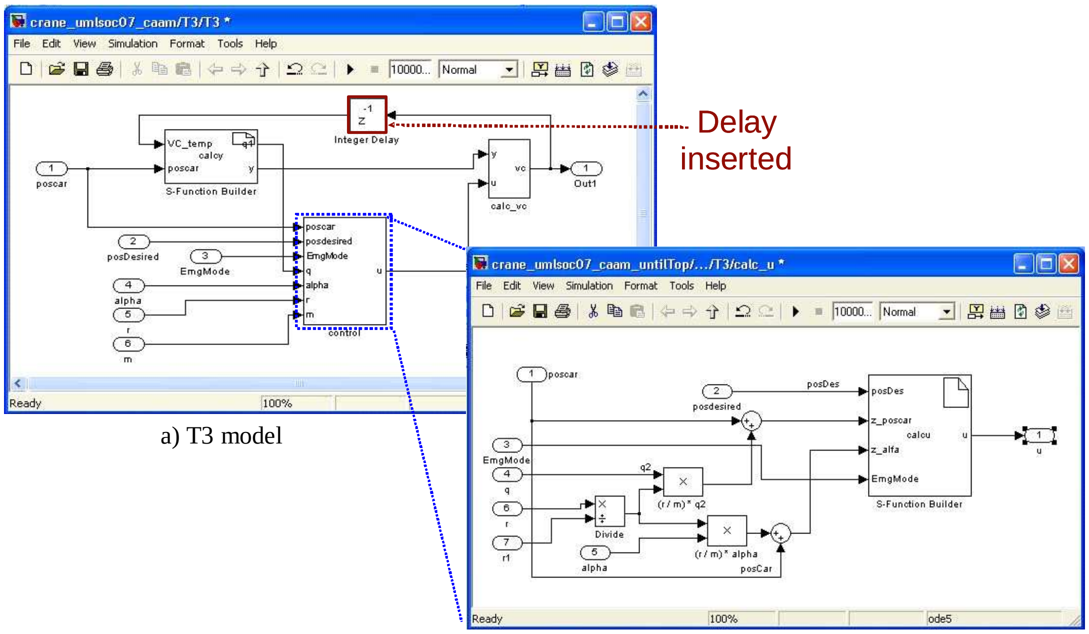
<<SAscheduler>> : Sched
<<SAschedRes>> : T3
: Control
: Calc_vc
: Platform
<<IO>> : Plant
<<IO>> : Engine
<<IO>> : IHM
<<SAschedRes>> : T1
<<SAschedRes>> : T2
1: mainTask()
<<SAtrigger>>
2: get_poscar():"poscar"
3:
4: calcy(vc_temp="out1", posCar="poscar", y="yo", q="q2")
5:
6: get_lm():"m1"
7:
8: get_r():"r1"
9:
10: get_posDes():"posdes"
11:
12: get_alpha():"alpha"
13:
14: get_emgMode():"emg_mode"
15:
16: control()
17: div(op1="r1", op2="m1", result="drm")
18:
19: mult(op1="q2", op2="drm", result="mq2drm")
20:
21: sum(op1="mq2drm", op2="poscar", result="zposcar")
22:
23: mult(op1="drm", op2="alpha", result="mdralpha")
24:
25: sum(op1="mdralpha", op2="poscar", result="zalpha")
26:
27: calcu(posDes="posdes", z_poscar="zposcar", z_alfa="alpha", emgMode="emg_mode", u="o1")
28:
29: calc_vc()
30: sum(op1="yo", op2="o1", result="uy")
31:
32: sat(in="uy", out="vcsat")
33:
34:
35: set_vc(vc="vcsat")
36:
37:

Figure 5.13: Crane system: UML sequence diagram for thread T3

We explain in detail here only the generation of the dataflow for the thread T3, which has a cyclic path (loop) and the insertion of the delay component can be observed. This cyclic path is found between the call message *set* and the *calcy* in the T3 sequence diagram. That is why, the argument *out1* is used as output of the method *set*, while the same argument is used as input for the method *calcy*. Figure 5.14 (a) presents the Simulink block diagram corresponding to the thread T3, where a delay block was automatically inserted between *calc_vc* and *calc_y* blocks.

When method invocations are nested, a hierarchical Simulink subsystem is instantiated to encapsulate the blocks generated to represent these methods. In the example, the subsystem *control* is instantiate to encapsulate the nested invocations for the methods *mult*, *div*, and *sum*. In addition, a subsystem called *calc_vc* and the S-function called *calc_y* are created. The subsystem *control* is detailed in Figure 5.14(b) and is composed of one S-Function and five pre-defined Simulink blocks. The methods invoked from the Platform (e.g. *sum*, *mult*, and *div*) are translated to *adder*, *multiplier*, and *divisor* Simulink blocks, respectively. The method *calcu* is mapped to a S-Function.

In this sequence diagram (Figure 5.13), the method *get_poscar( )* and *getalpha( )* invoked from thread T1 indicate the communication between the thread T3 and T1. The *get* prefix indicates that T1 send data to T3, ports and communication blocks are instantiated in the Simulink model to represent this communication, as shown in Figure 5.15 (right side). The invocations of methods from the objects Plant, Engine and IHM, which are stereotyped as <<*IO*>>, are translated to input and output ports that represent the interface of the system with external devices. For example, the method *get_posdes*( ) is translated to the input *posDesired* in the Simulink functional block diagram, as illustrated in Figure 5.14. The method *set_vc( )* is translated to an output port *out1* that is send to the motor represented by the object <<*IO*>> Engine in the sequence diagram.

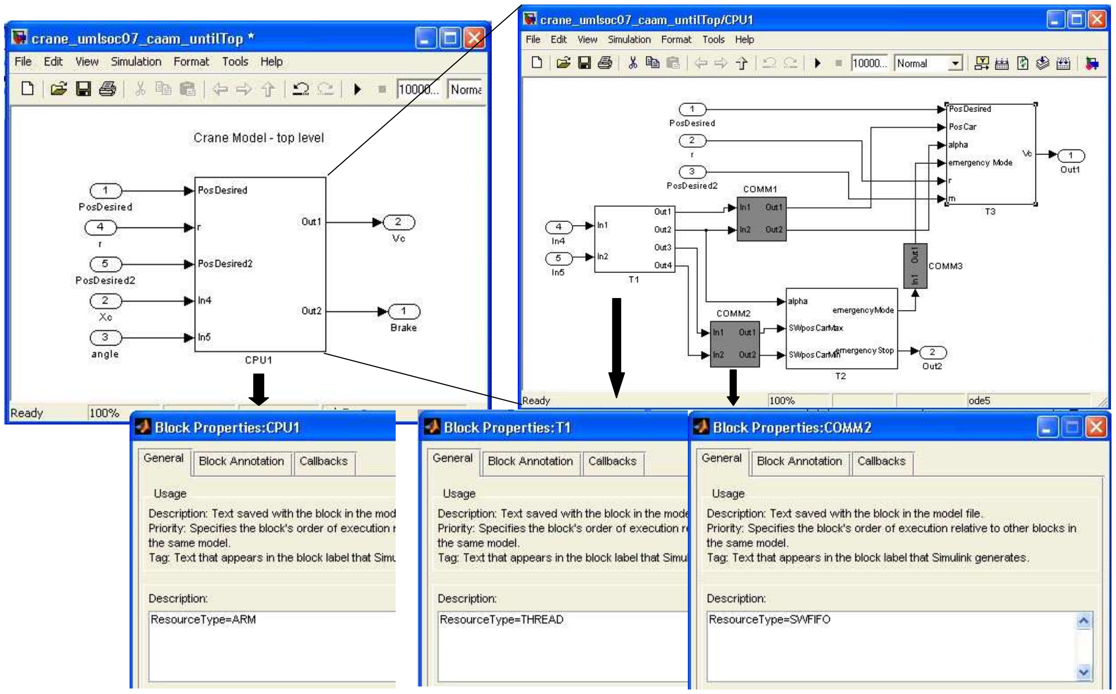


a) T3 model

b) T3: control subsystem

Figure 5.14: Crane Simulink CAAM: Thread T3 model

Finally, the Simulink CAAM higher hierarchical levels obtained from the Crane UML model are illustrated in Figure 5.15. The left side illustrates the top level, where there is only one CPU subsystem. The right side shows the threads allocated to this CPU, where the threads T1, T2 and T3 communicate via intra-SS channels. In the bottom part, this figure shows also the parameters set for the CPU, thread and intra-COMM subsystems.



Block Properties:CPU1

General | Block Annotation | Callbacks

Usage

Description: Text saved with the block in the mod
Priority: Specifies the block's order of execution r
the same model.
Tag: Text that appears in the block label that Simu

Description:

ResourceType=ARM

Block Properties:T1

General | Block Annotation | Callbacks

Usage

Description: Text saved with the block in the mode
Priority: Specifies the block's order of execution r
the same model.
Tag: Text that appears in the block label that Simu

Description:

ResourceType=THREAD

Block Properties:COMM2

General | Block Annotation | Callbacks

Usage

Description: Text saved with the block in the model file.
Priority: Specifies the block's order of execution relative to other blocks in the same model.
Tag: Text that appears in the block label that Simulink generates.

Description:

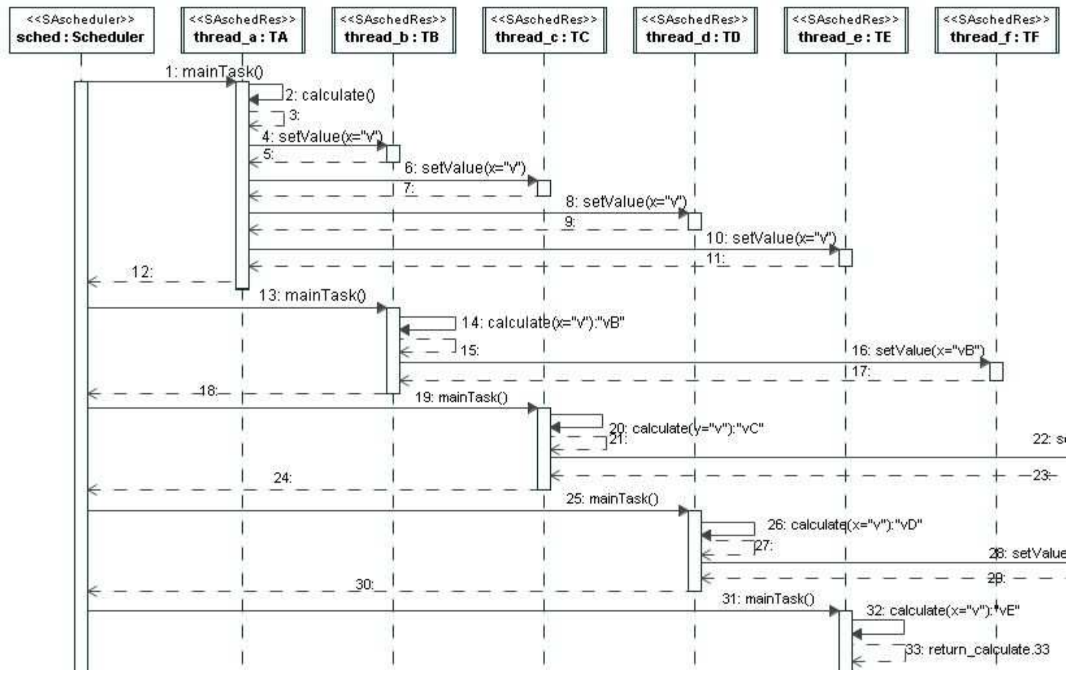
ResourceType=SWFIFO

Figure 5.15: Crane Simulink CAAM – CPU1 subsystem

5.3.2 Synthetic example

To validate the proposed grouping thread optimization, we developed a synthetic example, which has twelve communicating threads. The application was specified using a sequence diagram that expresses the communication between the application threads. Figure 5.16 illustrates a block of interactions of this sequence diagram, since the whole diagram is too big to show here. The complete sequence diagram is presented in Appendix B.

The communications captured from the sequence diagram are used to build a task graph, as shown in Figure 5.17(a), where the nodes represent the threads and the edges represent the communication between them. After the application of the grouping thread algorithm, the nodes of the graph are merged according to the communication between them. The result of this optimization is depicted in Figure 5.17(b), which shows that the twelve threads were allocated in four CPUs.



<<SAscheduler>> sched : Scheduler
<<SAschedRes>> thread_a : TA
<<SAschedRes>> thread_b : TB
<<SAschedRes>> thread_c : TC
<<SAschedRes>> thread_d : TD
<<SAschedRes>> thread_e : TE
<<SAschedRes>> thread_f : TF
1: mainTask()
2: calculate()
3:
4: setValue(x="v")
5:
6: setValue(x="v")
7:
8: setValue(x="v")
9:
10: setValue(x="v")
11:
12:
13: mainTask()
14: calculate(x="v"):"vB"
15:
16: setValue(x="vB")
17:
18:
19: mainTask()
20: calculate(y="v"):"vC"
21:
22: s
23:
24:
25: mainTask()
26: calculate(x="v"):"vD"
27:
28: setValue
29:
30:
31: mainTask()
32: calculate(x="v"):"vE"
33: return_calculate.33

Figure 5.16: Synthetic example: simplified sequence diagram

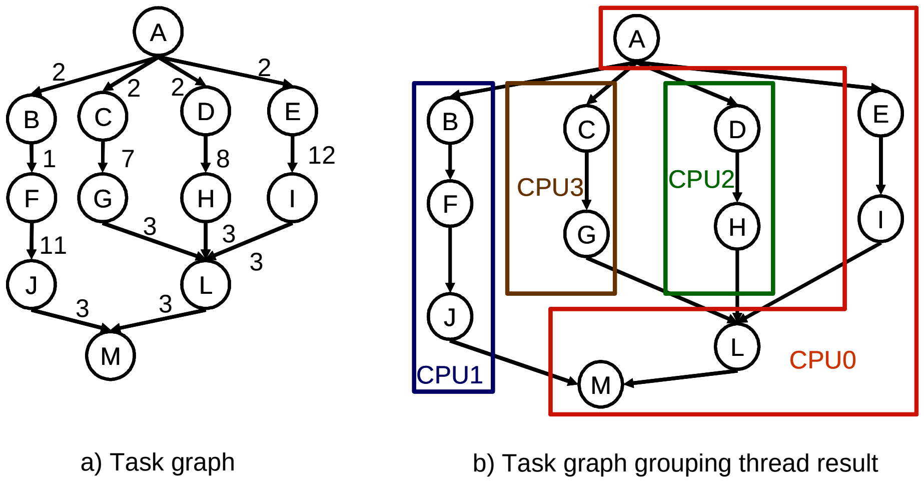
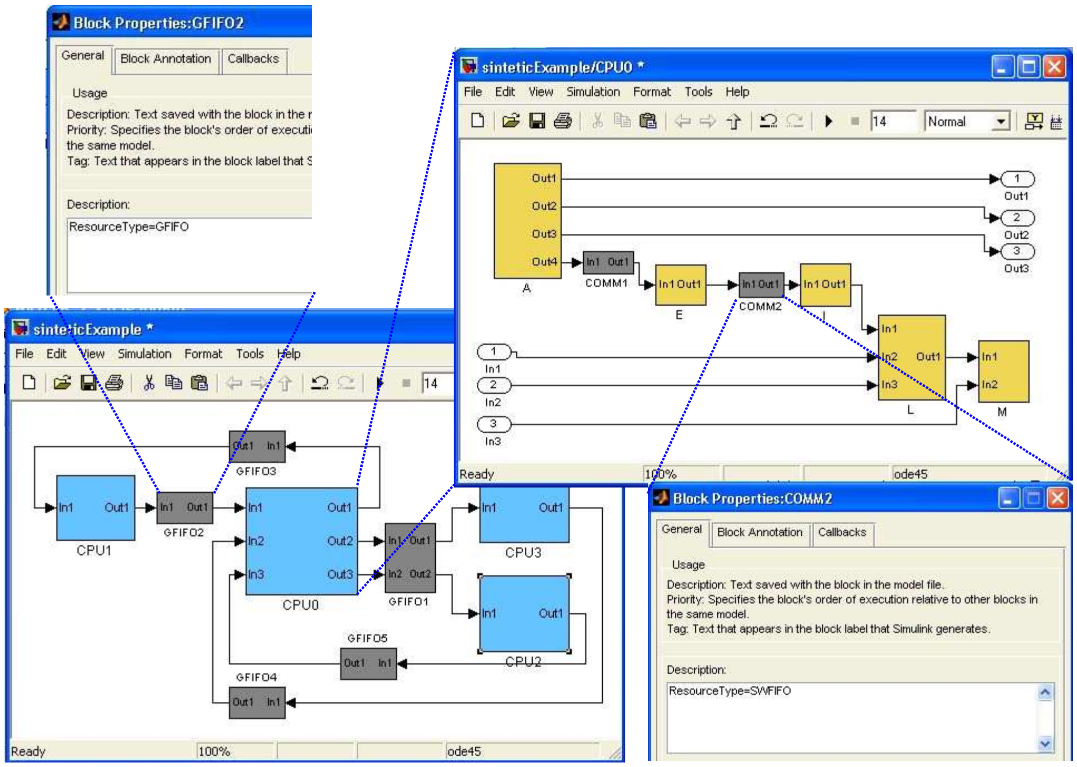


Figure 5.17: Synthetic example: Task graph

After applying the proposed map and the grouping thread algorithm for this application model, the Simulink CAAM model depicted in Figure 5.18 was obtained. Figure 5.18 shows the top level, where four CPU subsystems communicate through inter-SS communication blocks. This Figure shows also the threads allocated to the CPU0, where there are five thread subsystems (A, E, I, L and M) communicating via intra-SS COMM. The inference of communication is also performed to build this Simulink CAAM, in the Figure 5.18 is illustrated also the setting of parameters to indicate the communication protocol used for an intra-SS communication and an inter-

SS communication. The Simulink CAAM models generated by our tool are presented in the Appendix B in detail.



Block Properties:GFIFO2

Usage
Description: Text saved with the block in the r
Priority: Specifies the block's order of executi
the same model.
Tag: Text that appears in the block label that S

Description:
ResourceType=GFIFO

Block Properties:COMM2

Usage
Description: Text saved with the block in the model file.
Priority: Specifies the block's order of execution relative to other blocks in the same model.
Tag: Text that appears in the block label that Simulink generates.

Description:
ResourceType=SWFIFO

Figure 5.18: Synthetic example: generated Simulink CAAM

5.4 Concluding remarks

An automatic mapping from UML to Simulink CAAM was proposed. With it, we eliminated the necessity of manually building the Simulink model used as input for the Simulink-based design flow for MPSoC architectures, which generates multithread C code and the HW platform described in SystemC. The mapping is based on sequence and deployment diagrams. Other diagrams like class and collaboration diagrams could be used during the modeling, but our tool prototype does not capture information from them at the moment.

We show that some UML constructions can have a direct mapping to Simulink. However, the one-to-one mapping is not able to capture the whole model. It is still needed to make inferences performed in the optimization phase of our mapping tool. Two case studies were presented to show the proposed optimizations to be executed during the mapping from UML to Simulink. The first one shows the insertion of temporal barriers and the second one shows the grouping thread algorithm and the inference of communication channels.

The proposed mapping allows one to exploit the benefits of UML for requirements specification and software design, while providing a way to obtain complete executable code for MPSoC architectures from the high-level specification. Moreover, the same

UML model can be used to generate code using either traditional UML tools or a Simulink-based approach.

As future work, this tool will be integrated with an estimation tool to improve design space exploration, allowing that the deployment model can be build during the design space exploration step. Moreover, an analysis tool could be used to automatically determine which fragments of the system are dataflow and for these fragments the proposed mapping must be applied.

6 CONCLUSIONS

This thesis presented a comparison between UML and Simulink, two attractive modeling approaches for embedded system design. However, evaluating the state-of-the-art in embedded system design using high-level model, we found some limitations in the automation provided by available software development tools. In this context, strategies for embedded software generation from high-level models, using Simulink and UML languages, were proposed in order to solve the main limitations found on available design flows and tools.

Our UML-based strategy tried to bridge the gap between model and code though the use of a higher abstraction language. However, although we believe that this proposal could obtain good results, this proposed strategy was not developed because we decided to try a new thread. The author had the opportunity to work in the development of a code generator based on Simulink, which has shown to be a very interesting study. The proposed Simulink-based strategy focuses on the generation of multithread code target to multiprocessor architectures, which is not well addressed by available tools. In addition, this Simulink-based strategy provides a communication optimization technique, which can be used to reduce the communication overhead during the code generation (BRISOLARA, 2007a).

The comparison between UML and Simulink shows that both modeling approaches present pros and cons, which motivated us to find a way to simultaneously exploit the benefits of UML and Simulink modeling languages in a single design process. We proposed a software development flow, which allows to start with an UML model and generate the Simulink model from that. In this way, when a system module is dataflow, it is translated to Simulink, which provides more powerful features to model and simulate dataflow systems. This allows designers to work at a higher abstraction level, avoiding the necessity of building Simulink models directly, which means abstracting about low-level details like signals and ports.

The proposed flow allows to use UML as front-end for the proposed Simulink-based multithread code generation method. To support that, we define the mapping from UML to the Simulink CAAM that is used as input in this method. As the directly mapping is not possible, besides the mapping, the inference of communication channels and thread grouping are performed in order to build the Simulink CAAM model from that multithread code target to MPSoC architecture can be generated. In addition, temporal barriers are inserted when there is a cyclic path in the dataflow model in order to avoid deadlock.

Boldt (2007) and Reichamnn (2004) also proposed the integration of the UML and Simulink. However, differently of the Boldt’s and Reichmann’s approaches, our approach uses UML as modeling language for initial specification, which presents the advantages of using a standard language that is widely accepted in the software engineering community. In addition, the main advantage of the proposed integrated flow is to enable one to start with an UML model and decide which is the most appropriated tool to generate code for a system module, whether by Simulink of FSM based tools. Moreover, the same UML model can be also used to generate code by UML commercial tools or Simulink-based tools, thus enabling the reuse of models in different platforms or a comparison of different design alternatives.

Although the proposed flow can support other mappings than the Simulink one, this work addressed only the mapping from UML to Simulink and Simulink CAAM. However, to completely support the proposed flow, a FSM-like model should be also generated from the UML model in order to allow the use of different tools for code generation for control-flow system modules.

A limitation of the proposed mapping from UML to Simulink is that although the deployment diagram is not necessary when the grouping threads optimization is applied, the definition of threads continues to be required. This means that the designer needs to partition the system in threads and to describe thread behavior using sequence diagrams in order to apply the proposed mapping. As a future work, we plan to integrate an estimation step in the proposed software development flow. The estimation can be used to automatically determine the best partitioning and mapping solution and generate the deployment model. This avoids the necessity of the designer to specify the deployment model and partitioning the system in threads, while supporting design space exploration.

To show the usefulness of the proposed design flow, we developed a prototype, which is able to generate Simulink CAAM from an UML model. Using the developed prototype, we conducted experiments to show the benefits of our proposed mapping. At present, the designer applies the mapping from UML to Simulink for a whole system, but in the future, an analysis tool could be used to determine which fragments of the system are dataflow and control-flow ones, thus the mapping is applied only to the dataflow part.

Moreover, only sequence diagrams are used to capture thread behavior in our mapping. Other behavior diagrams, though, could also be used by a designer, since UML provides them. We plan to extend this mapping to support even other UML diagrams, like activity diagram, that is the closest to functional block diagrams.

REFERENCES

ARDIS, M. et al. A Framework for Evaluating Specification Methods for Reactive Systems: experience report. **IEEE Transactions on Software Engineering**, Los Alamitos, v. 22, n. 6, p. 378-389, 1996.

APACHE SOFTWARE. **Velocity Engine**. Available at: <http://velocity.apache.org/>. Visited on: May 2005.

ARTISAN SOFTWARE. **Artisan Studio**. Available at: <http://www.artisansw.com/products/>. Visited on: Feb. 2007.

BABU, E. M. M.; MALINOWSKI, A.; SUZUKI, J. Matilda: A Distributed UML Virtual Machine for Model-Driven Software Development. In: WORLD MULTI-CONFERENCE ON SYSTEMICS, CYBERNETICS AND INFORMATICS, 9., 2005. **Proceedings…** [S.l.: s.n.], 2005.

BALARIN, F. et al. Metropolis: an integrated electronic system design environment. **IEEE Computer**, [S.l.], v.36, n.4, p. 45-52, 2003.

BANERJEE, P. et al. The Paradigm Compiler for Distributed-Memory Multicomputers. **Computer**, Los Alamitos, v.28, n.10, p. 37-47, Oct. 1995.

BHATTACHARYYA, S. et al. PtolemyII Heterogeneous Concurrent Modeling and Design in Java. Tecnhical Report. Jan. , 2007. Available at: <http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIlatest/ptII6.0.2/doc/design/ptIIdesign1-intro.pdf>. Visited on: Mar. 2007.

BICHLER, L.; RADERMACHER, A.; SCHÜRR, A. Integrating Data Flow Equations with UML/Realtime. **Real-Time Systems**, [S.l.], n. 26, p. 107-125, 2004.

BJÖRKLUND, D.; LILIUS, J.; PORRES, I. A Unified Approach to Code Generation from Behavioral Diagrams. In: FORUM ON SPECIFICATION AND DESIGN LANGUAGES, FDL, 2003. **Proceedings…** [S.l.: s.n.], 2003. p. 21-34.

BJÖRKLUND, D.; LILIUS, J.; PORRES, I. Code Generation for Embedded Systems. In: GÉRARD, S.; BABAU, J. ; CHAMPEAU, J (Ed.). **Model Driven Engineering for Distributed Real-time Embedded Systems**. London: Hermes Science, 2005.

BOAS, G. van Emde. **Template Programming for Model-Driven Code Generation**. July 2004. Available at: <http://www.softmetaware.com/oopsla2004/emdeboas.pdf>. Visited on: Jan. 2005.

BOLDT, R. Combining the Power of MathWorks Simulink and Telelogic UML/SysML-based Rhapsody to Redefine the Model-Driven Development Experience. June, 2006. Telelogic White Paper. Available at: <http://www.ilogix.com/whitepaper-overview.aspx>. Visited on: Feb. 2007.

BRISOLARA, L.; HAN, S.-I.; GUERIN, X.; JERRAYA, A.; CARRO, L.; REIS, R. Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC. In: INTERNATIONAL WORKSHOP ON SOFTWARE AND COMPILERS FOR EMBEDDED SYSTEMS, SCOPES, 10., 2007, Nice. **Proceedings…** [S.l.: s.n.], 2007. p. 81-89.

BRISOLARA, L. B.; OLIVEIRA, M. F. S.; NASCIMENTO, F. A.; CARRO, L.; WAGNER, F. R. Using UML as a front-end for an efficient Simulink-based multithread code generation targeting MPSoCs. In: INTERNATIONAL WORKSHOP ON UML FOR SOC, UML-SoC, 4., 2007, San Diego. **Proceedings…** [S.l.: s.n.], 2007. p. 11-16.

BRISOLARA, L.; BECKER, L. B.; CARRO, L.; WAGNER, F. R.; PEREIRA, C. E. Comparing High-level Modeling Approaches for Embedded Systems Design. In: ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2005, Shanghai, China. **Proceedings…** Piscataway, NJ, USA: IEEE, 2005. v.2, p.986-989.

BRISOLARA L.; BECKER, L. B.; CARRO, L.; WAGNER, F. R.; PEREIRA, C. E. A Comparison between UML and Function Blocks for Heterogeneous SoC Design and ASIP Generation. In: MARTIN, G.; MUELLER, W. (Ed.). **UML for SoC Design**. Berlin: Springer-Verlag, 2005. p. 199-222.

BRISOLARA, L. B.; BECKER, L. B.; CARRO, L.; WAGNER, F. R.; PEREIRA, C. E.; REIS, R. A. L. Comparing High-level Modeling Approaches for Embedded Systems Design. In: ASIA SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2005, Shanghai, China. **Proceedings…** Piscataway, NJ, USA: IEEE, 2005. v.2, p.986-989.

BUCK, J. T. et al. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. **International Journal of Computer Simulation**, [S.l.], v. 4, p. 155-182, 2000.

BURCH, J. R.; PASSERONE, R.; SANGIOVANNI-VICENTELLI, A. L. Using Multiple Levels of Abstractions in Embedded Software Design. In: INTERNATIONAL WORKSHOP ON EMBEDDED SOFTWARE, EMSOFT, 2001. **Proceedings...** Berlin: Springer, 2001, p.324-343.

CESARIO, W. et al. Multiprocessor SoC Platforms: A Component-Based Design Approach. **IEEE Design & Test of Computers**, [S.l.], v. 19, n. 6, Nov.-Dec., 2002.

CHEN, R. et al. Embedded System Design Using UML and Platforms. In: VILLAR, E.; MERMET, J. (Ed.). **System Specification & Design Languages**. US: Springer, 2004. p. 119-128.

DAMM, W.; HAREL, D. LSCs: Breathing Life into Message Sequence Charts. **Formal Methods in System Design**, Dordrecht, v.19, n. 1, p. 45-80.

DENSMORE, D.; PASSERONE, R.; SANGIOVANNI-VINCENTELLI, A. A Platform-Based Taxonomy for ESL Design. **IEEE Design and Test of Computers**, [S.l.], v.23, n.5, p.359-374, Sept. 2006.

DOUGLASS, B. **Real-Time UML:** Developing Efficient Objects for Embedded Systems. Boston: Addison-Wesley, 1998.

DSPACE. **Real-time Interface for Multiprocessor systems (RTI-MP)**. Available at: <http://www.dspaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm>. Visited on: Oct. 2005.

DSPACE. **TargetLink**. Available at: <http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm>. Visited on: Oct. 2005.

ECLIPSE DEVELOPMENT TEAM. **ATLAS Transformation Language (ATL)**. Available at: <http://www.eclipse.org/m2m/atl/>. Visited on: Apr. 2007.

ECLIPSE DEVELOPMENT TEAM. **EMF (Eclipse Modeling Framework)**. Available at: <http://www.eclipse.org>. Visited on: May 2006.

ECLIPSE DEVELOPMENT TEAM. **Introduction to JET (Java Emitter Templates)**. Available at: <http://eclipse.org/articles/Article-JET/jet_tutorial1.html>. Visited on: June 2005.

EDWARDS, S. et al. Design of Embedded Systems: Formal Models, Validation, and Synthesis. **Proceedings of IEEE**, Piscataway, v. 85, n.3, p. 366-390, 1997.

ESTEREL TECHNOLOGIES. **SCADE tool**. Available at: <http://www.esterel-technologies.com/products/scade-suite/>. Visited on: Mar. 2007.

FLANAGAN, C. et al. Extended Static Checking for Java. **ACM SIGPLAN Notices**, New York, v.37, n.5, p. 234-245, 2002.

GERY, E.; HAREL, D.; PALACHI, E. Rhapsody: A Complete Life-Cycle Model-Based Development System. In: INTERNATIONAL CONFERENCE ON INTEGRATED FORMAL METHODS, IFM, 3., 2002, Turku, Finland. **Proceedings…** Berlin: Springer. 2002. p.1-10.

GOMAA, H. **Designing Concurrent Distributed, and Real-Time Applications with UML.** Boston: Addison-Wesley, 2000.

GRAAF, B.; LORMANS, M.; TOETENEL, H. Embedded Software Engineering: The State of the Practice. **IEEE Software**, Los Alamitos, v. 20, n. 6, p. 61-69, 2003.

GREEN, P. UML as a Framework for Combining Different Models of Computation. In: MARTIN, G.; MUELLER, W. (Ed.). **UML for SoC Design**. Berlin: Springer-Verlag, 2005. p. 37-62.

GROSE, T. J.; DONEY, G. C.; BRODSKEY, S. A. **Mastering XMI:** Java Programming with XMI, XML, and UML. New York, NY, USA: John Wiley & Sons, 2002.

HAN, S.-I.; CHAE, S.-I., JERRAYA, A. Functional modeling techniques for efficient SW code generation of video codec applications. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, ASP-DAC, 2006. **Proceedings…** New York, NY, USA: ACM Press, 2006a. p. 935 - 940.

HAN, S.-I. et al. Buffer memory optimization for video codec application modeled in Simulink. In: DESIGN AUTOMATION CONFERENCE, DAC, 2006, San Francisco, USA. **Proceedings…** New York, NY, USA: ACM Press, 2006b. p. 689-694.

HIRANANDANI, S.; KENNEDY, K.; TSENG, C. Compiling Fortran D for MIMD Distributed Memory Machines. **Communications of the ACM**, New York, v.35, n.8, p.66-80, 1992.

HONEKAMP, U. et al. Component-node-network: three levels of optimized code generation with ASCET-SD. In: IEEE INTERNATIONAL SYMPOSIUM ON COMPUTER AIDED CONTROL SYSTEM DESIGN, 1999. **Proceedings…** [S.l.:s.n.], 1999. p. 243-248.

HUANG, K.; HAN, S.-I.; POPOVICI, K.; BRISOLARA, L.; GUERIN, X.; LI, L.; YAN, X.; CHAE, S.-I.; CARRO, L.; JERRAYA, A. A. Simulink-Based MPSoC Design Flow: Case Study of Motion-JPEG and H.264. In: DESIGN AUTOMATION CONFERENCE, DAC, 2007, San Diego, California, USA. **Proceedings…** New York, NY, USA: ACM Press, 2007. p. 39-42.

HUBBERS, E.; OOSTDIJK, M. Generating JML Specifications From UML State Diagrams. In: FORUM ON SPECIFICATION AND DESIGN LANGUAGES, FDL, 2003. **Proceedings…**[S.l.:s.n.], 2003. p. 263-273.

ITO, S. A.; CARRO, L.; JACOBI, R. Making Java Work for Microcontroller Applications. **IEEE Design & Test**, Los Alamitos, v. 18, n. 5, p. 100-110, Set-Oct. 2001.

JACOBSON, I. et al. **Object-Oriented Software Engineering:** A Use Case Driven Approach. Boston: Addison-Wesley, 1992.

JERRAYA, A. A.; WOLF, W.; TENHUNEN, H. Guest Editors. **Computer**, [S.l.], v.38, n.7, p. 36-40, July 2005. Special Issue on MPSoC.

JOHN, K.; TIEGELKAMP, M. **IEC61131-3:** Programming Industrial Automation Systems: Concepts and programming languages, Requirements for programming systems, Aids to decision-making. Berlin: Springer-Verlag, 2001.

KAHN, G.; MACQUEEN, D.B. Coroutines and Networks of Parallel Processes. In: IFIP CONGRESS, 1977. **Information Processing 77**. Amsterdam: North-Holland, 1977. p. 993-998.

KANGAS, T. et al. 2006. UML-Based Multiprocessor SoC Design Framework. **ACM Transactions on Embedded Computing Systems**, New York, v.5, n.2, p.281-320, 2006.

KENNEDY CARTER. **iUML**: Intelligent UML. Available at: <http://www.kc.com>. Visited on: May 2005.

KEUTZER, K. et al. System-level design: Orthogonalization of concerns and platform-based design. **IEEE Transactions on CAD of Integrated Circuits and Systems**, New York, v.19, n.12, 2000.

KUMAR, R. et al. Heterogeneous Chip Multiprocessors. **Computer,** Los Alamitos, v.38, n.11, p. 32-38, 2005.

LAVAGNO, L.; MARTIN, G.; SELIC, B. **UML for Real:** Design of Embedded Real-Time Systems. Dordrecht: Kluwer Academic, 2003.

LEDECZI, A. et al. The Generic Modeling Environment. In: IEEE INTERNATIONAL WORKSHOP ON INTELLIGENT SIGNAL PROCESSING, 2001, Budapest, Hungary. **Proceedings…** [S.l.:s.n.], 2001.

LIEVERSE, P. et al. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. **Journal of VLSI Signal Processing for Signal, Image, and Video Technology**, Boston, v.29, n.3, p. 197-207, Nov. 2001.

MADISETTI, V. K.; ARPIKANONDT, C. **A Platform-Centric Approach to System-on-Chip (SOC) Design**. Netherlands: Springer, 2005.

MARTIN, G.; MUELLER, W. **UML for SoC Design**. Dordrecht, Netherlands: Springer, 2005. v.1.

MATHAIKUTTY, D. et al. UMoC++: Modeling environment for heterogeneous systems based on generic MoCs. In: VACHOUX, A. (Ed.). **Applications of Specification and Design Languages for SoCs**. Netherlands: Springer, 2006. p. 115-130.

MATTOS, J. C. B.; BRISOLARA, L. B.; HENTSCHKE, R.; CARRO, L.; WAGNER, F. R. Design Space Exploration with Automatic Generation of IP-based Embedded Software. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS, DIPES, 2004, Toulouse, France. **Proceedings...** Boston: Kluwer Academic, 2004. p.237-246.

MATHWORKS. **Real-Time Workshop (RTW)**. Available at: <http://www.mathworks.com>. Visited on: Nov. 2004.

MATHWORKS. **Simulink**. Available at: <http://www.mathworks.com>. Visited on: July 2003a.

MATHWORKS. **Stateflow**. Available at: <http://www.mathworks.com/products/stateflow/>. Visited on: Sept. 2003b.

MELLOR, S.; BALCER, M. **Executable UML:** A Foundation for Model Driven Architecture. Boston: Addison-Wesley, 2002.

MENTOR GRAPHICS. **BridgePoint UML Suite**. Available at: <http://www.projtech.com>. Visited on: Jan. 2005.

MOHANTY, S. et al. Rapid Design Space Exploration of Heterogeneous Embedded Systems using Symbolic Search and Multi-Granular Simulation. **ACM SIGPLAN Notices**, New York, v.37, n.7, p. 18-27, 2002.

MOSER, E.; NEBEL, W. Case Study: System Model of Crane and Embedded Control. In: DESIGN, AUTOMATION DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 1999, Munich, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999.

NATIONAL INSTRUMENTS. **Labview**. Available at: <http://www.ni.com/labview/>. Visited on: Jan. 2006.

NEEMA, S. et al. Constraint-Based Design-Space Exploration and Model Synthesis. In: NEEMA, S. et al. **Embedded Software**. Berlin: Springer, 2003. p. 290-305. (Lecture Notes in Computer Science, v. 2855).

NETBEANS DEVELOPMENT TEAM. **MDR:** Netbeans Metadata Repository. Available at: <http://mdr.netbeans.org>. Visited on: July 2005.

NO MAGIC. **MagicDraw.** Available at: <http://www.magicdraw.com/>. Visited on: April 2007.

OH, H.; HA, S. Memory-optimized Software Synthesis from Dataflow Program Graphs with Large Size Data Samples. **EURASIP Journal on Applied Signal Processing**, Akron, Ohio, v. 2003, p. 514-529, May 2003.

OMG (Object Management Group). **Unified Modeling Language (UML), version 2.1.1.** Available at: <http://www.omg.org/technology/documents/formal/uml.htm>. Visited on: June 2007a.

OMG (Object Management Group). **QoS&FT:** UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. 2004. (OMG document ptc/04-09-01). Available at: <http://www.omg.org/>. Visited on: June 2007b.

OMG (Object Management Group). **SysML:** Systems Modeling Language. Available at: <http://www.omgsysml.org/>. Visited on: in July 2006.

OMG (Object Management Group). **UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)**, RFP. [S.l.], 2005. (OMG document: realtime/05-02-06).

OMG (Object Management Group). **UML Profile for Schedulability, Performance, and Time**, 2002. (OMG document n. ptc/02-03-02). Available at: <http://www.omg.org>. Visited on: Jan. 2003.

OMG (Object Management Group). **XMI**: XML Model Interchange. (OMG document formal/2002-01-01). Available at: <http://www.omg.org>. Visited on: Jan. 2002.

OMG (Object Management Group). **MOF**: Meta-object Facility (MOF). [S.l.], 2001. (OMG document formal/2001-11-02). Available at: <http://www.omg.org>. Visited on: June 2005.

OMG (Object Management Group). **Unified Modeling Language Specification. v. 1.3.** [S.l.], 1999.

PIMENTEL A. D. et al. Exploring Embedded-Systems Architectures with Artemis. **Computer**, Los Alamitos, v.34, n.11, p. 57-63, 2001.

PIMENTEL, A. D.; ERBAS, C.; POLSTRA, S. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. **IEEE Transactions on Computers**, [S.l.], v.55, n.2, Feb. 2006.

PINO, J. L.; BHATTACHARYYA, S. S.; LEE, E. A. A hierarchical multiprocessor scheduling system for DSP applications. In: IEEE ASILOMAR CONFERENCE ON SIGNALS, SYSTEMS, AND COMPUTERS, 1995. **Proceedings…**[S.l.: s.n], 1995.

POPOVICI, K.; GUERIN, X.; BRISOLARA, L.; JERRAYA, A. Mixed Hardware Software Multilevel Modeling and Simulation for Multithread Heterogeneous MPSoC. In: INTERNATIONAL SYMPOSIUM ON VLSI DESIGN, AUTOMATION & TEST, VLSI-DAT, Taiwan, 2007. **Proceedings...** [S.l.: s.n], 2007.

PORRES, I. A toolkit for model manipulation. **Software and Systems Modeling,** Berlin, v. 2, n. 4, p. 262-277, Dec. 2003.

PTOLEMY. Available at: <http://ptolemy.eecs.berkeley.edu/>. Visited on: Mar. 2004.

REICHMANN, C. et al. GeneralStore - a CASE-tool integration platform enabling model level coupling of heterogeneous designs for embedded electronic systems. In: IEEE INTERNATIONAL CONFERENCE AND WORKSHOP ON THE ENGINEERING OF COMPUTER-BASED SYSTEMS, ECBS, 11., 2004. **Proceedings…** [S.l.: s.n], 2004. p. 225- 232.

RIOUX, L. et al. MARTE: A new profile RFP for the modeling and analysis of real-time embedded systems. In: UML-SOC WORKSHOP, UML-SoC, 2005. **Proceedings...** [S.l.: s.n], 2005.

RUMBAUGH, J. et al. **Object-Oriented Modeling and Design**. [S.l.]: Prentice Hall, 1991.

SANDER, I.; JANTSCH, A. System Modeling and Transformational Design Refinement in ForSyDe. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, New York, v.23, n.1, p. 17-32, 2004.

SANGIOVANNI-VINCENTELLI, A. et al. Benefits and Challenges for Platform Based Design. In: DESIGN AUTOMATION CONFERENCE, DAC, 41., 2004, San Diego, USA. **Proceedings…** New York: ACM Press, 2004. p. 409 - 414.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. A Vision for Embedded Software. Invited Talk. In: INTERNATIONAL CONFERENCE ON COMPILERS, ARCHITECTURE, AND SYNTHESIS FOR EMBEDDED SYSTEMS, CASES, 2001, Atlanta, USA. **Proceedings…** New York: ACM Press, 2001. p.1-7.

SCHMIDT, D. C. Guest Editor's Introduction: Model-Driven Engineering. **Computer**, Los Alamitos, v. 39, n. 2, p. 25-31, Feb. 2006.

SELIC, B. UML 2: A model-driven development tool. Model-Driven Software Development. **IBM Systems Journal**, Riverton, v. 45, n. 3, p. 607-620, 2006.

SELIC, B. Models, Software Models and UML. In: LAVAGNO, L.; MARTIN, G.; SELIC, B. **UML for Real:** Design of Embedded Real-Time Systems. Norwell, MA, USA: Kluwer Academic, 2003. p. 1-16.

SMARTQVT. **SmartQVT:** Open Source Transformation Tool Implementing the MOF 2.0 QVT-Operational Language. Available at: <http://smartqvt.elibel.tm.fr/>. Visited on: May 2007.

SZTIPANOVITS, J.; KARSAI, G. Embedded Software: Challenges and Opportunities. In: INTERNATIONAL WORKSHOP ON EMBEDDED SOFTWARE, EMSOFT, 1., 2001. **Embedded Software:** proceedings. Berlin: Springer, 2001. p. 403-415. (Lecture Notes in Computer Science. v. 2211.)

TELELOGIC. **Rhapsody**. Available at: <http://modeling.telelogic.com/products/rhapsody/index.cfm>. Visited on: Mar. 2007.

TELELOGIC. **Statemate**. Available at: <http://modeling.telelogic.com/products/statemate/index.cfm>. Visited on: Oct. 2003.

TELELOGIC. **Telelogic Tau Architecture/Development**. Available at: <http://www.telelogic.com/>. Visited on: Oct. 2004.

TENSILICA. **Xtensa V**. Available at: <http://www.tensilica.com>. Visited on: Jan. 2006.

TEMMERMAN, M. et al. Moving Up to the Conceptual Modeling Level for the Transformation of Dynamic Data Structures in Embedded Multimedia Applications. In: INTERNATIONAL CONFERENCE ON EMBEDDED COMPUTER SYSTEMS: ARCHITECTURES, MODELING, AND SIMULATION, SAMOS, 2005. Greece. **Proceedings...** [S.l.: s.n.], 2005.

TIDWELL, D. **XSLT**. [S.l.]: O'Reilly. 2001.

VERKEST, D.; KUNKEL, J.; SCHIRRMEISTER, F. System Level Design Using C++. In: DESIGN, AUTOMATION AND TEST IN EUROPE, DATE, 2000. **Proceedings...** [S.l.]: IEEE Computer Society, 2000.

ZHOU, G.; LEUNG, M.; LEE, E. A. **Code Generation Framework for Actor-Oriented Models with Partial Evaluation**. 2007. Technical Report. Available at: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-29.pdf>. Visited on: May 2007.

WEHRMEISTER, M. A.; BECKER, L.B.; PEREIRA, C. E. Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API. In: WORKSHOP ON THE MOVE TO MEANINGFUL INTERNET SYSTEMS, OTM, 2004. **Proceedings…** Berlin: Springer, 2004. p. 292-302. (Lecture Notes in Computer Science 3292).

WIEGAND, T. et al. Overview of the H.264/AVC Video Coding Standard. **IEEE Transactions on Circuits and Systems for Video Technology**, New York, v.13, n.8, p. 560-570, July 2003.

APPENDIX A ESTRATÉGIAS PARA DESENVOLVIMENTO DE SOFWARE EMBARCADO BASEADAS EM MODELOS DE ALTO NÍVEL

O desenvolvimento tecnológico expôs uma nova realidade, o uso intensivo pelo ser humano de sistemas computacionais. Esses sistemas computacionais, quando embutidos em um produto, são chamados de sistemas embarcados, pois constituem parte de um todo e desenvolvem tarefas específicas. Os sistemas embarcados estão presentes em diversos setores tais como: automotivo, aeronáutico, telecomunicações, eletrônica de consumo e de dispositivos medicinais. Geralmente, os sistemas embarcados complexos são implementados como system-on-chip (SoC) heterogêneos compostos de componentes de hardware dedicado, processadores programáveis, memória, controladores de interface e outros módulos de hardware.

Muitos sistemas embarcados têm requerimentos que os diferem dos tradicionais sistemas desenvolvidos para PCs. Muitas vezes esses são inseridos em equipamentos para os quais a portabilidade é um fator importante, nestes casos, tamanho, peso e dissipação de potência são requisitos críticos. Muitos sistemas embarcados possuem restrições de tempo de resposta e de confiabilidade, além das restrições tradicionais de consumo de energia, área de memória e desempenho. Além disso, o tempo para lançamento do produto no mercado é crucial para o sucesso do projeto. Portanto, produtividade e qualidade são simultaneamente requeridas no projeto de sistemas embarcados a fim de lançar um produto competitivo no mercado.

Projeto baseado em plataformas (PBD) (SANGIOVANNI-VINCENTELLI et al., 2001; SANGIOVANNI-VINCENTELLI, 2004; VERKEST, 2000) é uma metodologia de projeto que visa maximizar o reuso de componentes e consequentemente melhorar a produtividade dos projetos. Com o reuso de plataformas de hardware, o software embarcado é o que diferencia os produtos.

Segundo Burch (2001), o interesse por implementações baseadas em software cresceu principalmente motivado pelo aumento no poder computacional das plataformas de hardware que possibilitou mover mais funcionalidade para o software. Outro fator motivacional foi o aumento dos custos de desenvolvimento de hardware que motivou o reuse de uma mesma plataforma em diferentes produtos. A utilização desta abordagem de projeto baseado em software proporciona flexibilidade e portabilidade, enquanto diminui o tempo de projeto. Além disso, quando se desloca maior funcionalidade para o software, o custo do sistema pode ser reduzido, assim como o tempo para colocá-lo no mercado já que uma plataforma pré-definida será reusada. Porém, alguns aspectos tais como consumo de potência e desempenho podem ser prejudicados.

Atualmente, com o uso de abordagens baseadas em plataformas, o gargalo para a implementação de sistemas embarcados vem sendo considerado o desenvolvimento de software, a sua depuração e a sua integração com os componentes de hardware. Deste modo, o software está se tornando cada vez mais o principal fator de custo nos dispositivos embarcados (GRAFF, 2003). Este cenário motiva a investigação de estratégias para acelerar o desenvolvimento de software embarcado através de ferramentas de automação.

Na área de engenharia de software, ferramentas CASEs (Computer Aided Software Enginnering) são largamente utilizadas para automatizar o processo de desenvolvimento. Como softwares convencionais são geralmente homogêneos, ou seja, dedicados a um único domínio, as ferramentas de automação de software focam na gerencia do desenvolvimento de grandes sistemas, sem lidar com aspectos como a heterogeneidade. Porém, sistemas embarcados complexos abrangem uma grande variedade de aplicações e possuem muitas funcionalidades agregadas em um único sistema, devido a isso, existem diferentes necessidades de computação requeridas em um único produto. Por exemplo, a especificação de um telefone celular não requer somente processamento digital de sinais para o domínio de telecomunicações, que segue o modelo de computação tempo-discreto. Ela também requer lógica seqüencial para descrever várias outras aplicações embarcadas no celular (agenda, alarme, etc.). Assim, pode-se afirmar que os sistemas embarcados são naturalmente heterogêneos e, portanto, as ferramentas de automação devem suportar diferentes modelos de computação. Porém, as ferramentas existentes para automação de desenvolvimento de software não oferecem este recurso.

Além disso, o desenvolvimento de software embarcado difere do software tradicional quanto às exigências impostas ao projeto de software embarcado. Por exemplo, restrições de tamanho de memória e consumo de potência são muito mais rígidas nestes sistemas do que em sistemas tradicionais, o que é um outro fator que inviabiliza o uso de ferramentas CASE tradicionais para o projeto de software embarcado. Embora, consideremos estes aspectos de qualidade do software muito importantes para o domínio de embarcados, isto não faz parte do escopo deste trabalho.

Além do projeto baseado em plataformas, o uso de abstrações de alto nível também tem sido adotado para lidar com a crescente complexidade dos sistemas embarcados e aumentar a produtividade do projeto. Selic (2003) e Gomma (2000) ressaltam que o uso de técnicas de projeto começando por níveis de abstração mais altos é a única maneira viável para lidar complexidade das novas gerações de sistemas embarcados, sendo considerada uma prática essencial para o sucesso do projeto.

O uso de abstrações de mais alto nível permite abstrair detalhes de implementação na linguagem alvo, facilitando a especificação do sistema que é realizada através da construção de modelos, as invés de escrita de código. Usando esta abordagem, modelos de sistemas embarcados podem evoluir de abstrações de alto nível até implementações, assegurando um processo muito mais suave e confiável que o provido pelas práticas de engenharia de software tradicionais. A tradução automática do modelo de alto nível em código executável é altamente desejável, mas dependendo da notação de modelagem usada, diferentes graus de interação com o projetista podem ser requeridos. A linguagem de modelagem deve prover mecanismos para expressar não só a funcionalidade como também os requisitos da aplicação, alem de suportar a validação e mecanismos que facilitem a obtenção de uma implementação do modelo. Muitas abordagens de modelagem e linguagens têm sido propostas para a especificação de sistemas embarcados, mas não há um consenso, já que nenhuma linguagem é considerada boa para modelar todas às aplicações encontradas neste domínio.

Dentre as abordagens propostas, duas abordagens se ressaltam, uma que é a baseada em blocos funcionais e é provida pelo Simulink e a outra que é baseada em orientação a objetos e provida pela UML. Tradicionalmente, abordagens baseadas em blocos funcionais têm sido usadas nas comunidades de processamento de sinais e de engenharia de controle para desenvolvimento de sistemas embarcados. Esta abordagem tem sido largamente aceita pela indústria, principalmente, devido ao grande número de ferramentas disponíveis como, por exemplo, Simulink (MATHWORK, 2003a) e Labview (NATIONAL INSTRUMENTS, 2006).

Por outro lado, a linguagem UML é considerada a linguagem de fato para a modelagem de sistemas orientados a objetos e tem crescido em popularidade também na área de projeto e especificação de sistemas embarcados de tempo real. Em (LAVAGNO, 2003), esforços que descrevem o uso de UML em diferentes fases do projeto de sistema embarcados são apresentados.

No contexto deste trabalho, as duas abordagens baseadas em UML e Simulink são avaliadas quanto à modelagem, geração de código e mecanismos de exploração do espaço de projeto. Os resultados de análise foram publicados em (BRISOLARA, 2004; BRISOLARA, 2005b) e são apresentados e discutidos no capítulo 2. A partir desta análise, observou-se que as abordagens de geração de software embarcado baseado em UML e Simulink possuem limitações, e esta tese propõe estratégias para resolver as principais limitações encontradas nas duas abordagens.

Apesar dos esforços e propostas para extensão da linguagem, UML continua não sendo adequada para modelagem de sistemas dataflow, pois ela é uma linguagem baseada em eventos e, portanto, control-flow. Quanto à geração de código, a maioria das ferramentas UML geram somente esqueletos de código a partir de modelos estáticos. Poucas ferramentas são capazes de gerar código a partir de diagramas comportamentais. Porém, para geração de código completo, as ferramentas exigem que o projetista insira fragmentos de código junto aos diagramas. Todas as ferramentas comerciais encontradas geram código somente a partir de diagramas de estado e para gerar código completo, exigem que o projetista descreva as ações referentes a cada estado. Muitas vezes, o projetista usa a linguagem de programação alvo para fazer isso, o que além de tornar o modelo dependente da linguagem alvo. Nós propomos aqui o uso de abstrações junto aos modelos comportamentais UML para reduzir o esforço requerido ao projetista, reduzindo o número de linhas de código, enquanto, suportando a geração de código completo a partir de modelos UML. Esta proposta é discutida no capítulo 3 desta tese.

Por outro lado, Simulink suporta modelos do tipo dataflow de tempo-discreto e tempo-contínuo frequentemente encontrados em aplicações embarcadas. Além disso, completo código pode ser gerado usando Real-time workshop (MATHWORKS, 2004). Porém, o código gerado é voltado para uma arquitetura mono-processada. Observando esta limitação, propomos uma estratégia para geração de código multithread voltado para plataformas multi-processadas (MPSoC) heterogêneas (BRISOLARA, 2007a), que é apresentada no capítulo 4. Nesta estratégia, código multithread é gerado a partir de um modelo denominado Simulink CAAM (*combined algorithm architecture model*). O modelo Simulink CAAM combina algoritmo (funcionalidade) e arquitetura, contendo informações sobre o particionamento do sistema em *threads* e também sobre o mapeamento das *threads* para processadores. A abordagem de geração de código a partir de modelos Simulink CAAM proposta aqui faz parte de um fluxo de projeto de sistemas MPSoC baseado em Simulink proposto em (HUANG et al., 2007).

A comparação entre UML e Simulink mostra também que as duas abordagens de modelagem apresentam pros e contras, o que motiva pesquisadores a encontrar uma maneira de explorar simultaneamente benefícios providos pelas duas linguagens em um único fluxo de projeto. Recentes esforços mostram que tanto UML como Simulink são consideradas atrativas para o projeto de sistemas embarcados. Boldt (2007) propõe a integração de modelos Simulink em modelos UML na ferramenta Rhapsody. Reichmann (2004) também propôs a integração de modelos desenvolvidos em diferentes ferramentas incluindo UML, Simulink e Statemate (TELELOGIC, 2003). Usando esta abordagem, módulos do sistema podem ser modelados usando a ferramenta mais apropriada e geradores de código de domínio específico são usados para gerar código para cada módulo. SysML (OMG, 2006) foi proposta como uma extensão de UML para ser usadas por engenheiros de sistemas, provê um alto grau de integração com o paradigma de blocos funcionais. Porém, a primeira especificação desta linguagem ainda é muito próxima da UML, não apresentando melhorias significativas. Além disso, devido a ser ainda uma novidade, as ferramentas de modelagem que suportam a linguagem não tiveram ainda suas capacidades devidamente avaliadas.

Nesta tese (capítulo 5), propomos uma maneira de integrar UML e Simulink em um único fluxo de projeto, permitindo que UML seja usada como a linguagem de especificação e *front-end* para diferentes abordagens de geração de código (BRISOLARA, 2007b). Diferentemente das abordagens propostas por Boldt e Reichmann, nossa abordagem propõe o uso de UML como a linguagem única para especificação inicial. O fluxo proposto baseia-se na tradução de modelos UML para outras notações mais adequadas para a geração de código, por exemplo, modelos Simulink para dataflow ou máquina de estados (*finite state machines*, FSM) para control-flow. Além disso, o fluxo proposto permite que um modelo UML possa ser reusado para diferentes abordagens de geração de código, sejam abordagens tradicionais baseadas em UML ou abordagens baseada em Simulink, visando diferentes plataformas.

Uma das principais motivações para a definição deste fluxo de projeto integrador foi usar UML como *front-end* para a ferramenta de geração de código *multithread* baseada em Simulink proposta em (BRISOLARA, 2007a). Portanto, nós propomos aqui um mapeamento entre UML e Simulink CAAM. O proposto mapeamento permite a exploração dos benefícios de UML para especificação de requisitos funcionais e não funcionais, enquanto provê um caminho para a obtenção de código executável, para rodar em uma arquitetura composta de múltiplos processadores heterogêneos, a partir de um modelo de alto nível de abstração. O Simulink CAAM gerado a partir do modelo UML pode ser usado como entrada para um fluxo completo de projeto de sistemas MPSoCs, podendo ser usado também na geração da especificação do HW para a plataforma MPSoC. O emprego da abordagem proposta evita que projetistas construam ou modifiquem modelos Simulink diretamente, o que significa maior abstração e evita que projetistas lidem com detalhes de baixo nível como sinais e portas.

O proposto mapeamento de UML para Simulink CAAM baseia-se principalmente em informações extraídas de diagramas de seqüência e diagrama de distribuição (*deployment*). O Diagrama de distribuição é usado para indicar o mapeamento de *threads* para processadores. O diagrama de seqüência é o principal diagrama usado neste mapeamento, sendo assim um diagrama de seqüência deve ser definido para cada *thread* que compõe o sistema. A partir do diagrama de seqüência captura-se um diagrama composto de blocos Simulink, compondo um modelo dataflow e que define o comportamento da *thread*. A invocação de métodos de objetos passivos no diagrama de seqüência é mapeada para blocos funcionais (pré-definidos ou definidos pelo projetista). A invocação de métodos entre diferentes *threads* indica a comunicação entre elas e são mapeadas para blocos de comunicação no modelo Simulink CAAM e a invocação de métodos a partir de objetos decorados com o estereótipo <<*IO*>> são mapeados para portas de entrada e saída no modelo Simulink.

Não existe um mapeamento 1 para 1 entre as duas notações. Portanto, além do mapeamento, propomos três tipos de otimizações, que são a inferência de canais de comunicação, a inserção de barreiras temporais e o agrupamento de *threads*. A inferência de canais de comunicação e o agrupamento de *threads* são necessários para a construção do modelo Simulink CAAM, pois tratam de aspectos como comunicação entre *threads* e definição do mapeamento de *threads* para processadores. Estes são aspectos importantes na definição de um modelo multithread e muti-processado. Além disso, a fim de evitar deadlocks, barreiras temporais são inseridas automaticamente quando caminhos cíclicos são encontrados na geração do modelo dataflow Simulink.

Quando o agrupamento de *threads* é usado, ao invés do projetista definir a alocação de *threads* para processadores através do diagrama de distribuição, um algoritmo baseado no *linear clustering* é usado para definir o melhor mapeamento de *threads* para processadores com base no volume de comunicação entre as *threads*. A inferência de canais de comunicação instancia blocos de comunicação para representar a comunicação entre *threads* explicitamente no modelo UML. Esta etapa seta também o protocolo de comunicação dependendo do tipo de comunicação requerida, se é entre *threads* alocadas a uma mesma CPU ou em diferentes CPUs.

To show the usefulness of the proposed design flow, we developed a prototype, which is able to generate Simulink CAAM from an UML model. Using the developed prototype, we conduct experiments to show the benefits of our proposed mapping. At present, the designer applies the mapping from UML to Simulink for whole system, but in the future, an analysis tool could be used to determine which fragments of the system are dataflow and control-flow, thus the mapping is applied only for the dataflow part.

Embora o fluxo de mapeamento proposto suporte outros mapeamentos além do Simulink, este trabalho endereça somente o mapeamento de UML para Simulink e Simulink CAAM. Para completamente suportar o fluxo proposto, o mapeamento de UML para modelos do tipo máquina de estados (FSM) também deveria ser provido. Desta maneira, além de um caminho para geração de código baseado em modelos Simulink, o fluxo suportaria o uso de diferentes ferramentas para geração de código para módulos do sistema que sejam do tipo control-flow. A tradução de UML para FSM será considerada como trabalho futuro.

Para mostrar a utilidade de nossa proposta, um protótipo foi desenvolvido no contexto desta tese, o qual implementa o mapeamento de UML para Simulink CAAM. Usando este protótipo, experimentos foram realizados. Atualmente, o mapeamento é aplicado para todo o modelo UML, porém, futuramente ele deve ser aplicado apenas a parte dataflow do sistema. A fim de automatizar ainda mais o processo de desenvolvimento de software embarcado e o suporte a modelos heterogêneos, planejamos usar uma ferramenta de análise para particionar o sistema em módulos dataflow e control-flow. Após o particionamento, cada fragmentos do modelo pode ser mapeado para a notação mais adequada a seu tipo e após o mapeamento, a abordagem de geração de código apropriada pode ser usada para obter uma implementação para cada módulo do sistema.

APPENDIX B EXPANDED FIGURES

This section presents expanded figures used in chapter 4 and 5. Figures of the MJPEG Simulink CAAM, used as case study in chapter 4, are presented here in detail. In addition, it includes the Simulink CAAM metamodel, all sequence diagrams used in the Crane case study in chapter 5. For the synthetic example, whole sequence diagram (partially presented in section 5.3.2) and the hierarchical levels of the Simulink CAAM generated for our tool are presented here.

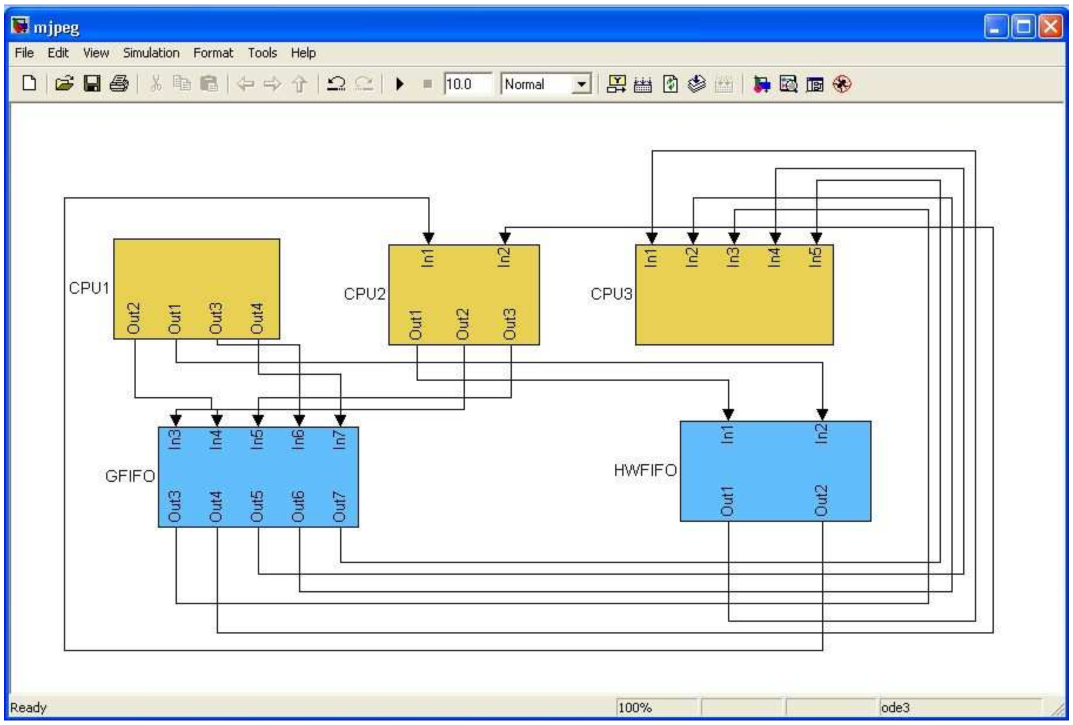


Fig. 1: MJPEG decoder Simulink CAAM: top level (Fig. 4.15a)

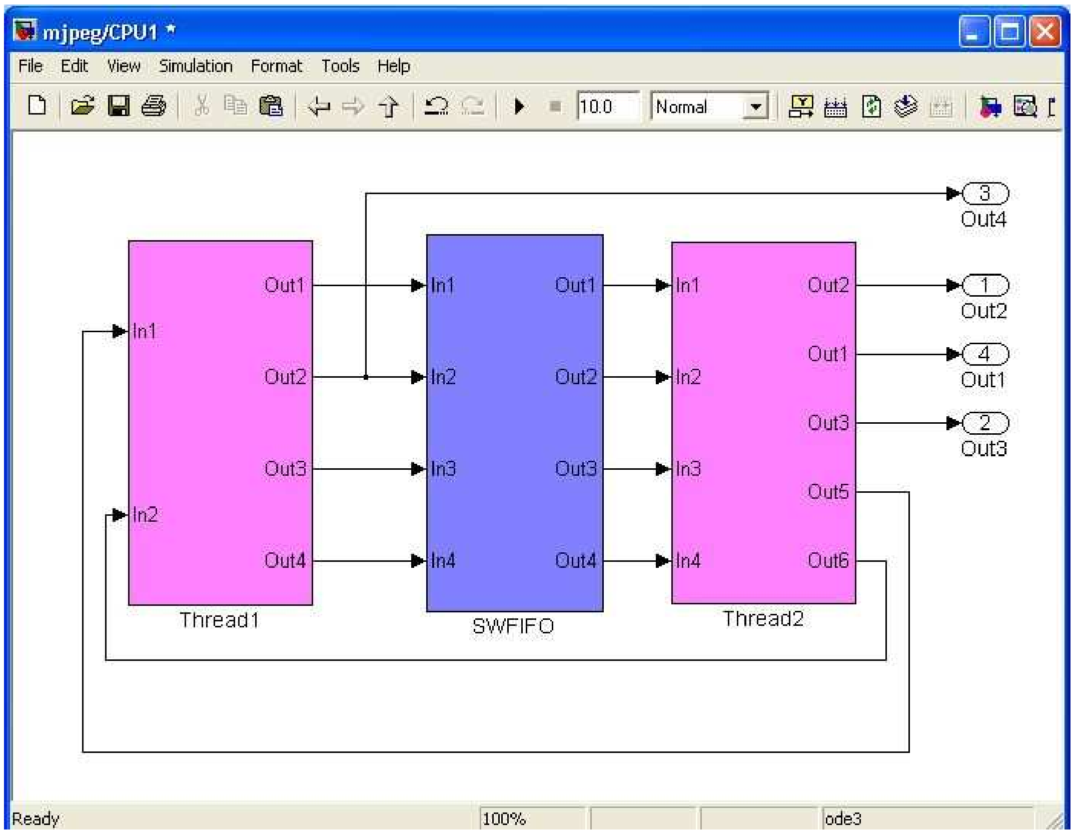


Fig. 2: MJPEG decoder Simulink CAAM: CPU1 subsystem (Fig. 4.15b)

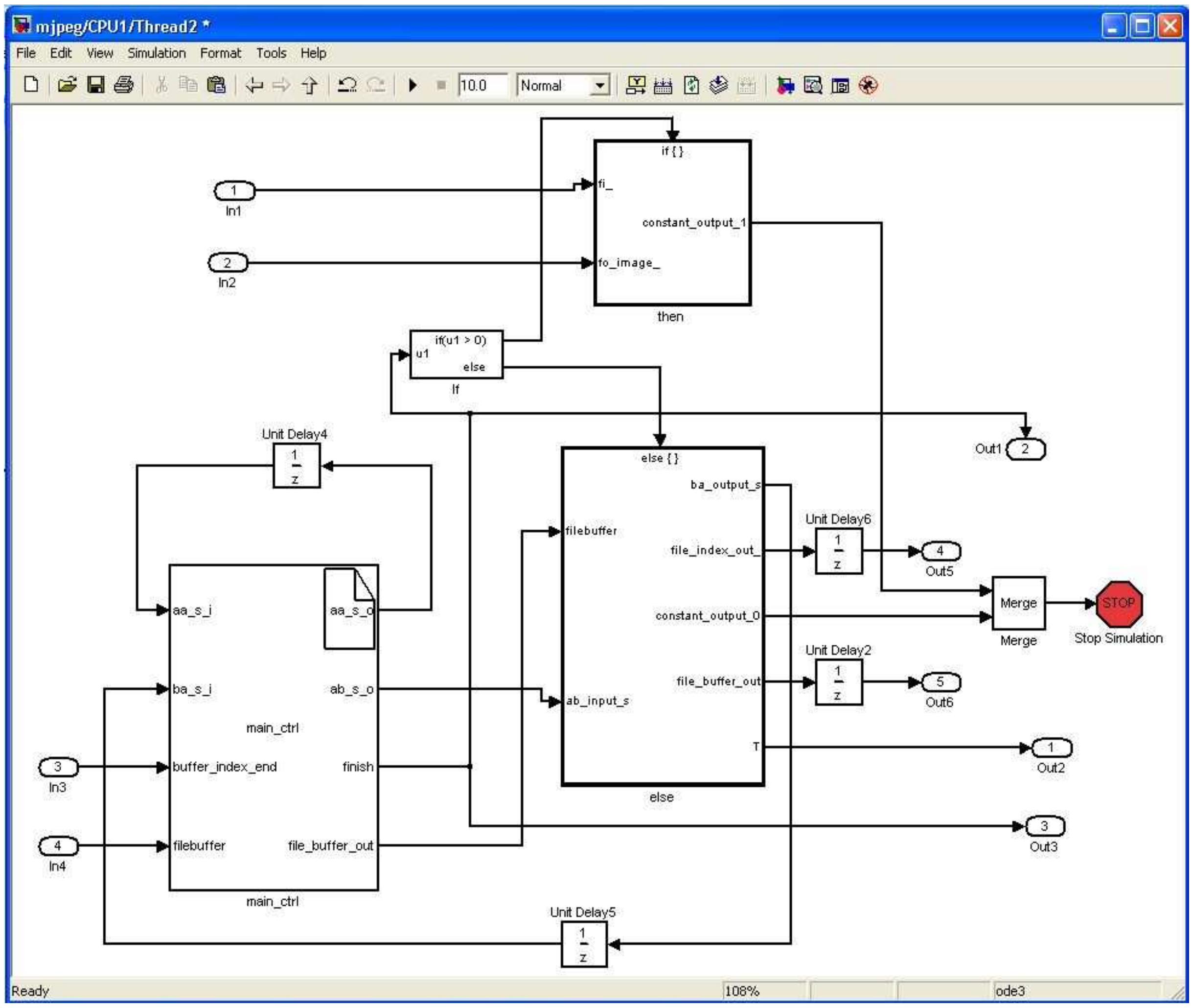


Fig. 3: MJPEG decoder Simulink CAAM: Thread2 subsystem (Fig. 4.15c)

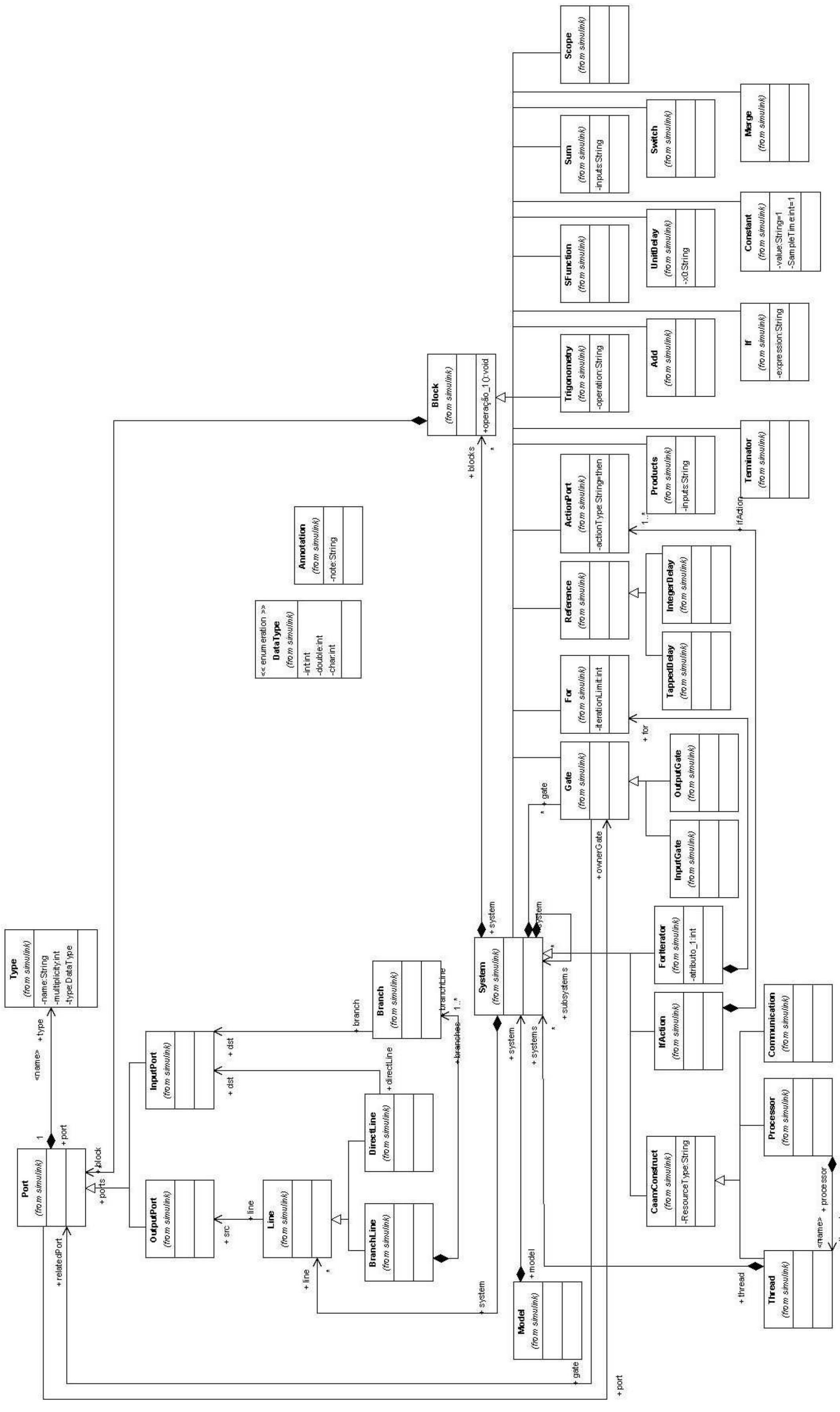
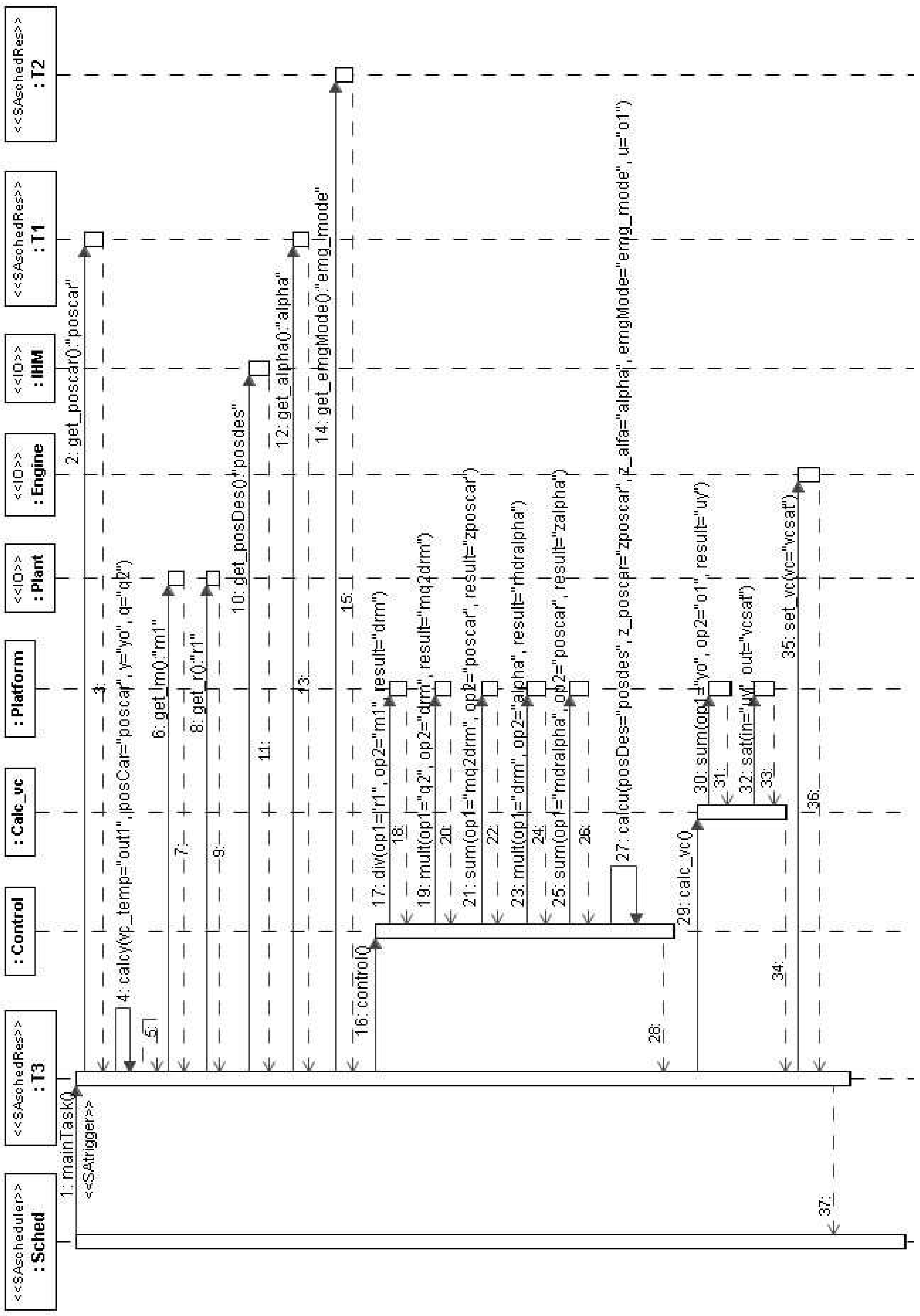


Fig 4: Simulink CAAM meta-model



<<SAscheduler>> : Sched
<<SAschedRes>> : T3
: Control
: Calc_vc
: Platform
<<IO>> : Plant
<<IO>> : Engine
<<IO>> : IHM
<<SAschedRes>> : T1
<<SAschedRes>> : T2
1: mainTask()
<<SAtrigger>>
2: get_poscar():"poscar"
3:
4: calcy(vp_temp="out1",posCar="poscar",y="yo",q="q2")
5:
6: get_lm():"m1"
7:
8: get_r():"r1"
9:
10: get_posDes():"posdes"
11:
12: get_alpha():"alpha"
13:
14: get_emgMode():"emg_mode"
15:
16: control()
17: div(op1="r1",op2="m1",result="drm")
18:
19: mult(op1="q2",op2="drm",result="mq2drm")
20:
21: sum(op1="mq2drm",op2="poscar",result="zposcar")
22:
23: mult(op1="drm",op2="alpha",result="mdralpha")
24:
25: sum(op1="mdralpha",op2="poscar",result="zalpha")
26:
27: calcu(posDes="posdes",z_poscar="zposcar",z_alfa="alpha",emgMode="emg_mode",u="o1")
28:
29: calc_vc()
30: sum(op1="yo",op2="o1",result="uy")
31:
32: sat(in="uy",out="vcsat")
33:
34:
35: set_vc(vc="vcsat")
36:
37:

Fig. 5: Crane control system: Sequence diagram of Thread T3 (Fig. 5.13)

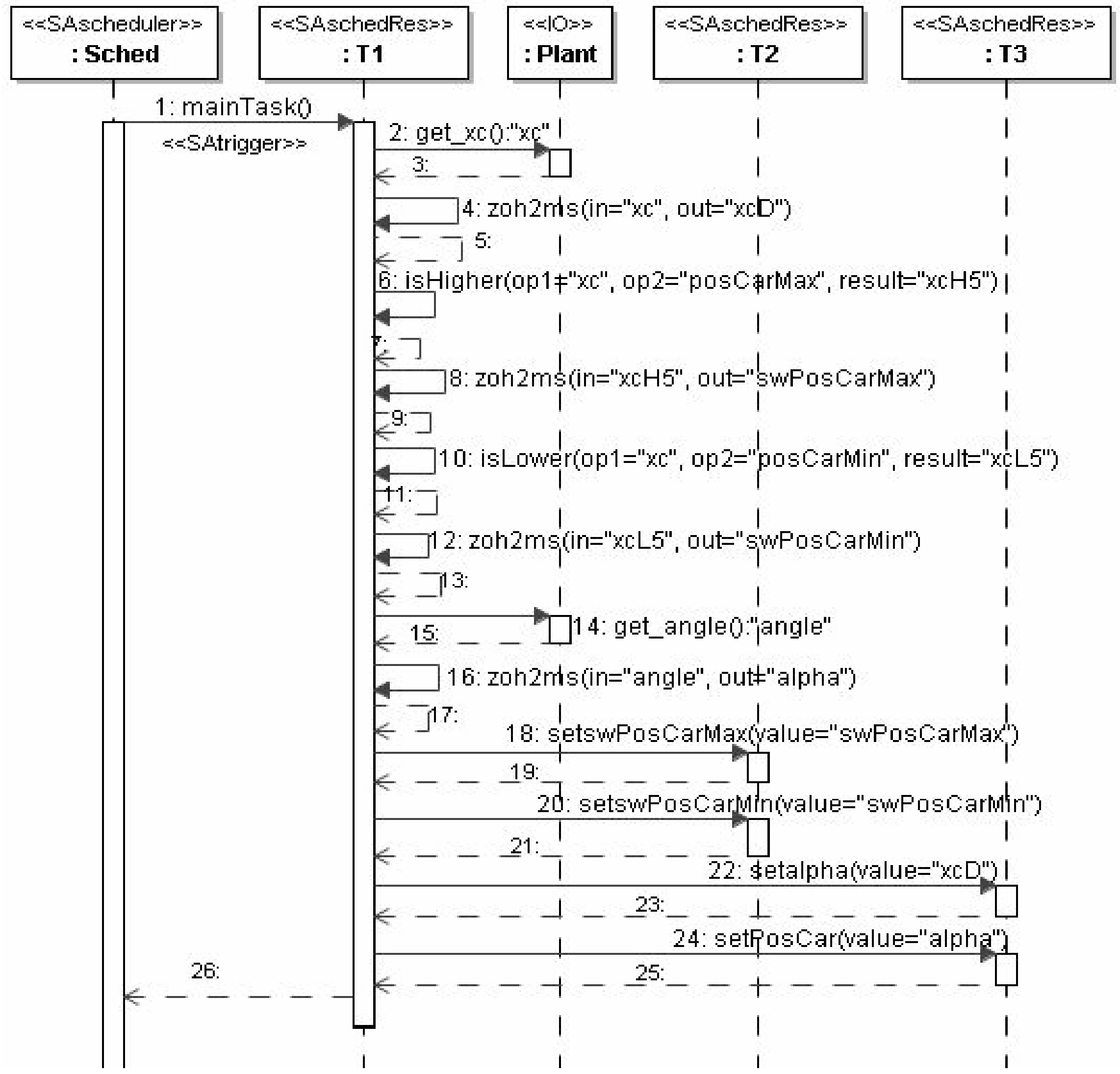
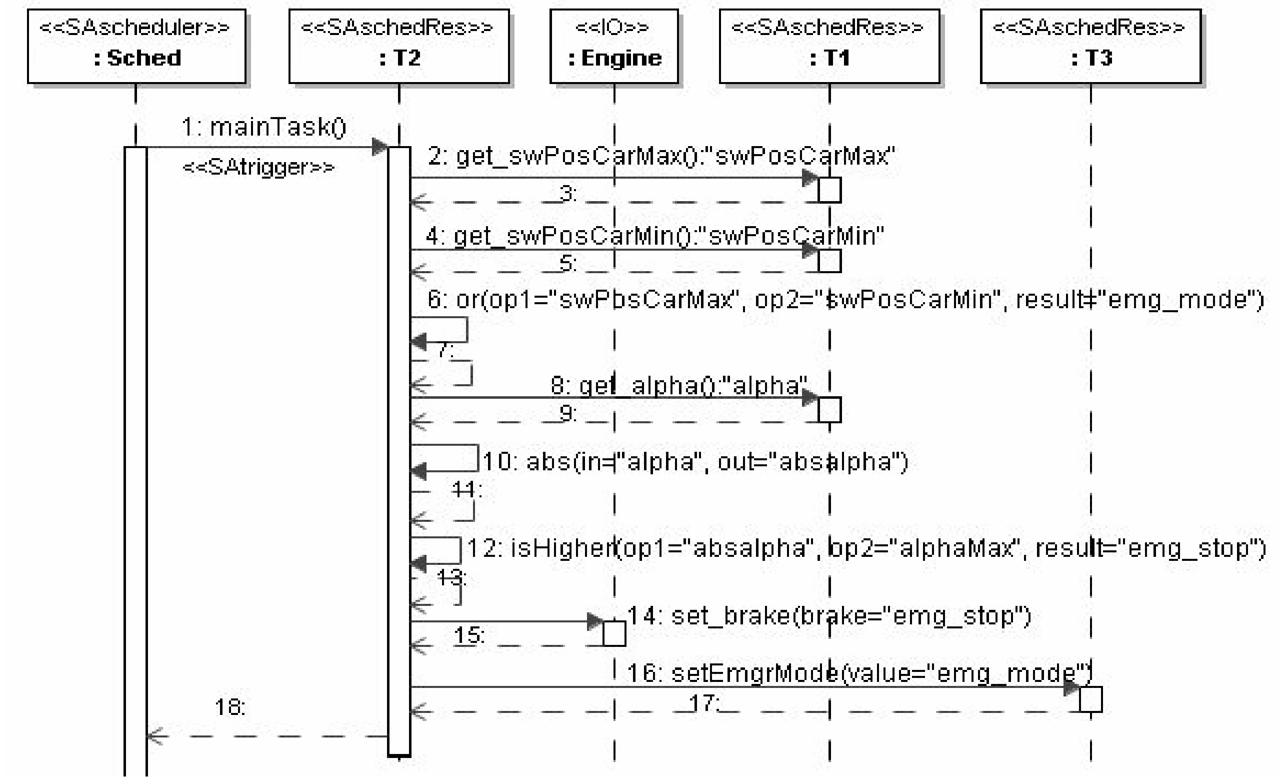

<<SAscheduler>>
: Sched
<<SAschedRes>>
: T1
<<IO>>
: Plant
<<SAschedRes>>
: T2
<<SAschedRes>>
: T3
1: mainTask()
<<SAtrigger>>
2: get_xc():"xc"
3:
4: zoh2ms(in="xc", out="xcD")
5:
6: isHigher(op1="xc", op2="posCarMax", result="xcH5")
7:
8: zoh2ms(in="xcH5", out="swPosCarMax")
9:
10: isLower(op1="xc", op2="posCarMin", result="xcL5")
11:
12: zoh2ms(in="xcL5", out="swPosCarMin")
13:
14: get_angle():"angle"
15:
16: zoh2ms(in="angle", out="alpha")
17:
18: setswPosCarMax(value="swPosCarMax")
19:
20: setswPosCarMin(value="swPosCarMin")
21:
22: setalpha(value="xcD")
23:
24: setPosCar(value="alpha")
25:
26:

Fig. 6: Crane control system: Sequence diagram of Thread T1 (Fig. 5.11)



<<SAscheduler>> : Sched
<<SAschedRes>> : T2
<<IO>> : Engine
<<SAschedRes>> : T1
<<SAschedRes>> : T3
1: mainTask()
<<SAtrigger>>
2: get_swPosCarMax():"swPosCarMax"
3:
4: get_swPosCarMin():"swPosCarMin"
5:
6: or(op1="swPosCarMax", op2="swPosCarMin", result="emg_mode")
7:
8: get_alpha():"alpha"
9:
10: abs(in="alpha", out="absalpha")
11:
12: isHigher(op1="absalpha", op2="alphaMax", result="emg_stop")
13:
14: set_brake(brake="emg_stop")
15:
16: setEmgrMode(value="emg_mode")
17:
18:

Fig. 7: Crane control system: Sequence diagram of Thread T2 (Fig 5.12)

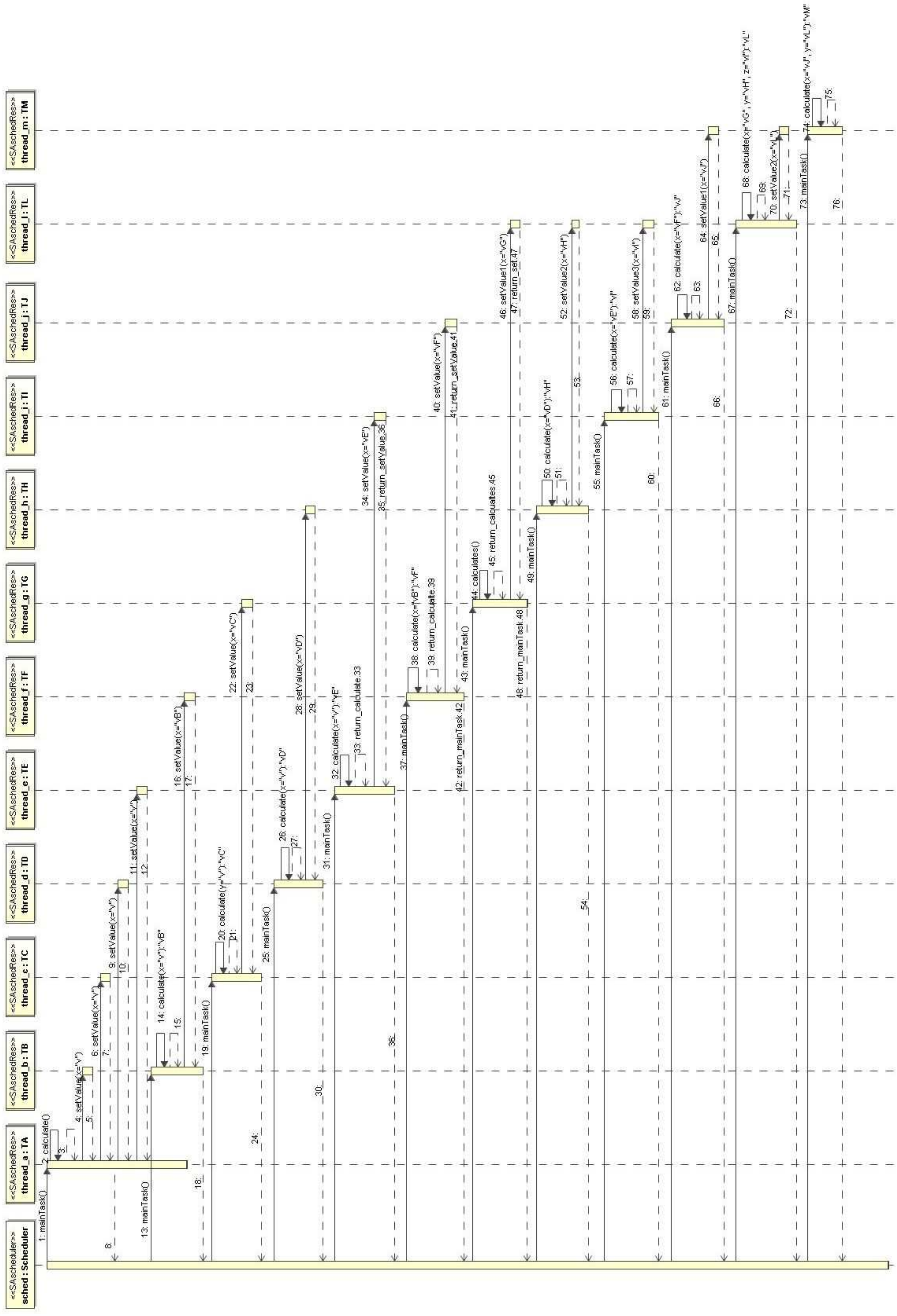


Fig. 8: Synthetic example: Sequence diagram of whole application (Fig. 5.17)

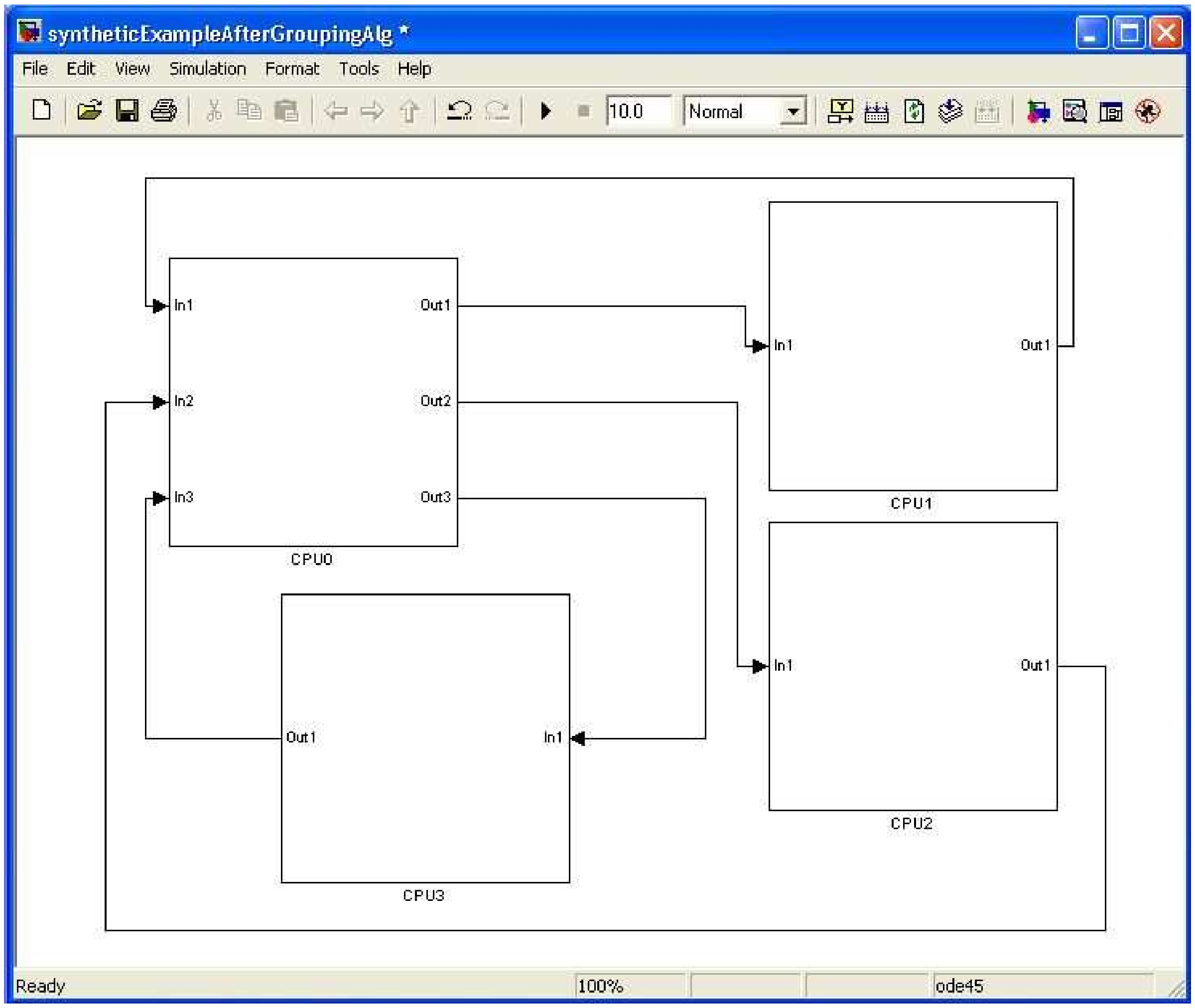


Fig. 9: Synthetic example: generated Simulink CAAM – top level

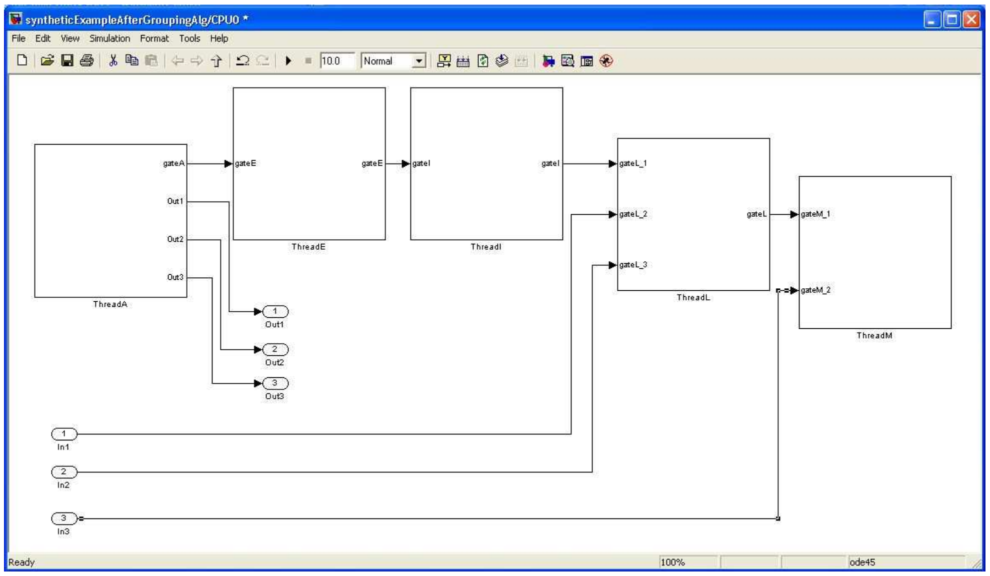


Fig. 10: Synthetic example: generated Simulink CAAM – CPU0 subsystem

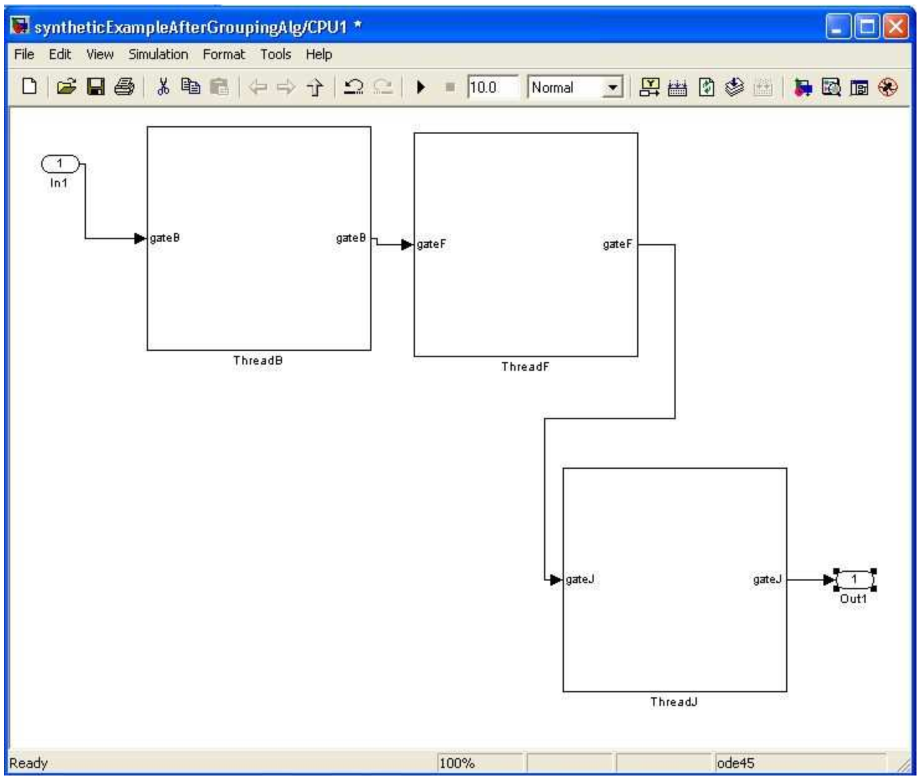


Fig.11: Synthetic example: generated Simulink CAAM – CPU1 subsystem

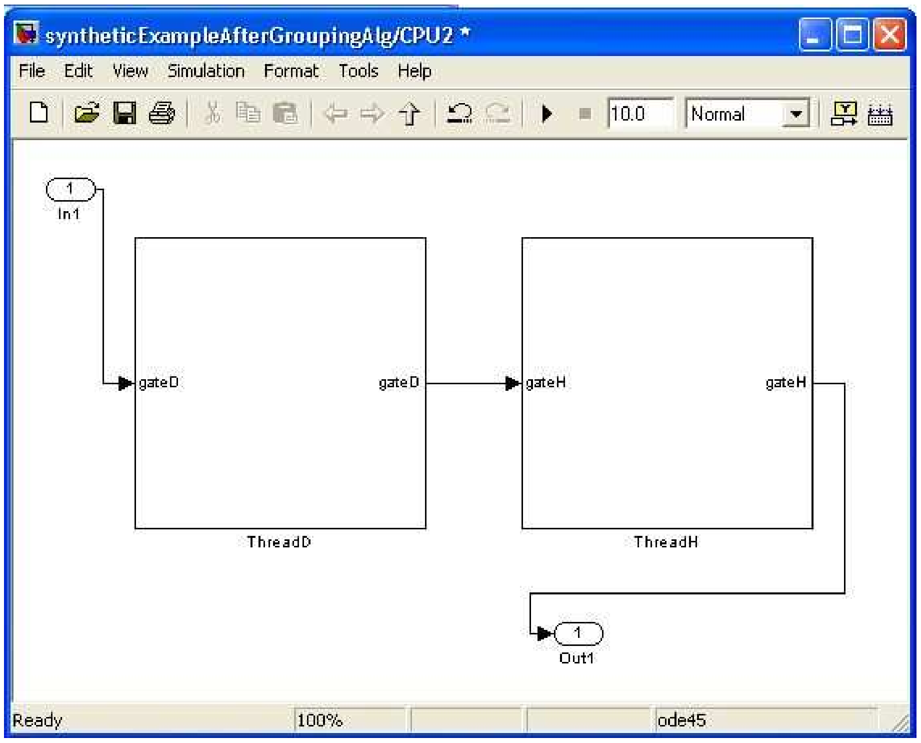

syntheticExampleAfterGroupingAlg/CPU2 *
File Edit View Simulation Format Tools Help
10.0
Normal
1
In1
gateD
gateD
ThreadD
gateH
gateH
ThreadH
1
Out1
Ready
100%
ode45

Fig. 12: Synthetic example: Simulink CAAM – CPU2 subsystem

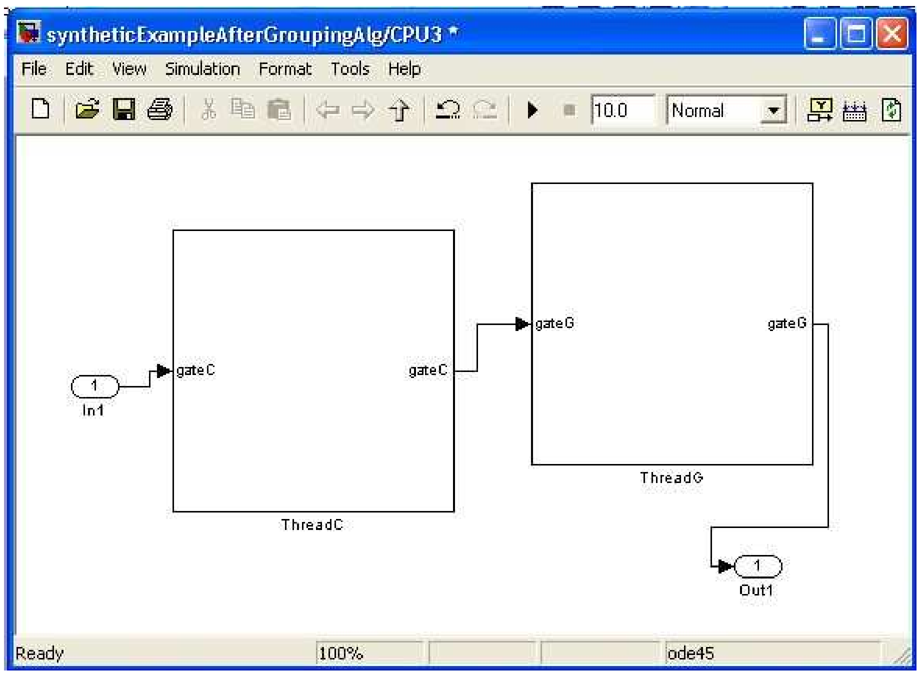

syntheticExampleAfterGroupingAlg/CPU3 *
File Edit View Simulation Format Tools Help
10.0
Normal
gateC
gateC
gateG
gateG
1
In1
ThreadC
ThreadG
1
Out1
Ready
100%
ode45

Fig.13: Synthetic example: generated Simulink CAAM – CPU3 subsystem