Downloaded from orbit.dtu.dk on: Dec 20, 2017

Technical University of Denmark

DTU

Programming Models and Tools for Intelligent Embedded Systems

Sørensen, Peter Verner Bojsen ; Madsen, Jan

Publication date:
2010

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

Link back to DTU Orbit

Citation (APA):
Sørensen, P. V. B., & Madsen, J. (2010). Programming Models and Tools for Intelligent Embedded Systems. Kgs. Lyngby, Denmark: Technical University of Denmark (DTU). (IMM-PHD-2010-232).

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Programming Models and Tools for Intelligent Embedded Systems

Peter Verner Bojsen Sørensen

Kongens Lyngby 2010
IMM-PHD-2010-70

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

Design automation and analysis tools targeting embedded platforms, developed using a component-based design approach, must be able to reason about the capabilities of the platforms. In the general case where nothing is assumed about the components comprising a platform or the platform topology, analysis must be employed to determine its capabilities. This kind of analysis is the subject of this dissertation.

The main contribution of this work is the Service Relation Model used to describe and analyze the flow of service in models of platforms and systems composed of re-usable components. Fundamental to the service relation model is the novel concept of service aggregation that simply states that one service is accessible through another.

The usefulness and versatility of the Service Relation Model is demonstrated by means of three different applications. In the first application, the model is used for checking the consistency of a design with respect to the availablity of services and resources. In the second application, a tool for automatically implementing the communication infrastructure of a process network application, the Service Relation Model is used for analyzing the capabilities of a platform and as a basis for efficient code generation. In the third application, the Service Relation Model and the concept of consistency are used to guide an automated procedure for designing systems composed of components.

Resumé

Værktøjer til automatisering af design opgaver målrettet indlejrede platforme, der er udviklet ved brug af en komponent-baseret design metode, skal have adgang til information vedrørende platformenes funktionelle egenskaber. I det generelle tilfælde hvor intet kan antages vedrørende en platforms topologi eller komponenterne, der indgår i platformen, må analyse benyttes til at bestemme platformens egenskaber. Denne form for analyse er emnet for denne afhandling.

Hovedbidraget i dette arbejde er Service Relations Modellen, som benyttes til at beskrive og analysere tilgængeligheden af tjenester i modeller af platforme og systemer bestående af gen-brugbare komponenter. Service Relations Modellen er baseret tjeneste-opsamlings konceptet, der beskriver hvorledes tjenester kan være tilrådighed igennem andre tjenester.

Brugbarheden og alsidigheden af service relations modellen bliver demonsteret igennem tre eksempeler på forskellige anvendelser. I den første anvendelse benyttes modellen til at kontrollere konsistensen af et design med hensyn til tilrådigheden af tjenester og resourcer. I den anden anvendelse, et værktøj til automatisk at implementere kommunikations infrastrukturen i et process netværk, benyttes Service Relations Modellen til at analysere en platforms funktionelle egenskaber og som grundlag for effektiv kode generering. I den tredje anvendelse, benyttes Service Relations Modellen sammen med konceptet om konsistens som grundlag for en automatiseret procedure til at designe systemer bestående af komponenter.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

Acknowledgment I would like to thank my supervisor Jan Madsen for his support and help during my studies and the preparation of this thesis. I also want to thank my colleague through many years Per Larsen for his relentless criticism of my work. Finally, I would like to thank my family and my girlfriend Ane Skak for their patience and support during the writing of this thesis.

Lyngby, February 2010

Peter Verner Bojsen Sørensen

Papers contributed

[88] Peter Verner Bojsen Sørensen and Jan Madsen. Consistency Check for Component-Based Design of Embedded Systems using SAT-Solving. *Proceedings of the 20th Nordic Workshop on Programming Theory*, 2008. Published.

[87] Peter Verner Bojsen Sørensen and Jan Madsen. Component-based Service Availablity Checking. *Nordic Workshop and Doctoral Symposium on Dependability and Security*, 2008. Published.

[89] Peter Verner Bojsen Sørensen and Jan Madsen. Generating Process Network Communication Infrastructure for Custom Multi-Core Platforms. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 2010. Accepted (to appear 2010).

Contents

Summary i

Resumé iii

Preface v

Papers contributed vii

1 Introduction 1
- 1.1 Dealing with Complexity . . . 2
- 1.2 Design Methodologies . . . 4
- 1.3 Scope of Work . . . 7
- 1.4 Problem Statement . . . 11
- 1.5 Contributions . . . 17
- 1.6 Thesis Outline . . . 18

2 The Service Relation Model 19
- 2.1 Informal Presentation . . . 19
- 2.2 Basic Concepts . . . 29
- 2.3 In-depth Presentation . . . 35
- 2.4 Analysis . . . 50
- 2.5 Discussion & Summary . . . 52

3 Consistency Checking 55
- 3.1 Service Classes and Hierarchies . . . 56
- 3.2 Assertions . . . 59
- 3.3 Resources and Resource Claims . . . 62
- 3.4 The `xSRM` Framework . . . 80
- 3.5 Consistency Checking . . . 84

3.6 Related Work 95
3.7 Discussion & Summary 101

4 Automated Programming 103
4.1 Introduction 103
4.2 The Procedure 105
4.3 Code Generation 112
4.4 The PAL Generation Tool 119
4.5 Case Study: MJPEG 139
4.6 Related Work 145
4.7 Discussion & Summary 148

5 Automated Design Generation 149
5.1 Introduction 149
5.2 Procedure Overview 152
5.3 Preparation 155
5.4 Encoding 158
5.5 Decoding 169
5.6 Experiments 169
5.7 Optimizations 182
5.8 Discussion & Summary 184

6 Conclusion 185
6.1 Contributions 185
6.2 Discussion 187
6.3 Future Work 187

A Alternative Analysis Algorithm 189

B Evaluation Functions 191

C Implementation Scheme C Templates 195

D Automated Programming Timing Measurements 199

E Design Generation Timing Measurements 201

Assumption is the mother of all fuck up's!

Travis Dane, Under Siege 2: Dark Territory

Chapter 1

Introduction

Embedded systems are specialized computing platforms and an integral part of a device or a machine and typically used for control and data processing. In contrast to general purpose computers, such as personal computers (PC's), embedded systems are characterized by only having a single purpose that never changes. More often than not, an embedded system is associated with strict constraints on metrics such as performance, power consumption, reliability and, naturally, cost. Embedded systems can be found in a very broad range of products from cellular phones to household appliances. In 2006, the number of embedded systems outnumbered the number of PC's by a factor of 10 [11]. In 2009 the total revenue from embedded systems was 88 billion dollars, up from 46 billion dollars in 2004 which yields an average annual growth of 14%, [4].

For high volume products it is imperative to keep the cost per unit to a minimum which often complicates the design significantly resulting in a relatively high non-recurring engineering (NRE) cost. This is acceptable because the NRE cost can be amortized across a large number of units and, thus, contributes marginally to the unit cost. For such products, custom hardware or application specific integrated circuits (ASIC's) are often used since they offers the best performance and the lowest power usage. For low volume products, on the other hand, the NRE cost must be kept low at the expense of a higher cost per unit which implies that the design must be kept relatively simple. For such products more general purpose common-of-the-shelf (COTS) platforms, such as micro-

controllers, are employed.

In some markets, such as the market for electronic consumer products, the time from a product is conceived until it is available for sale, called time-to-market, is of critical concern.

Ever since the dawn of the semi-conductor era, the density of transistors in integrated circuits has increased according to Moore's law [66] and it is likely to continue to do so in the foreseeable future. The increase is based on the use of smaller wires and transistors. The technological progress has enabled the construction of more complex chips. Today, it is entirely possible to integrate complete computing platforms with processors, networks, memories, sensors and actuators in a single chip. The ability to cram a historical amount of transistors onto a silicon wafer means that we are able to produce very advanced systems. The setup costs associated with the manufacturing of such hardware is, however, overwhelming and is only feasible for parts that can be guaranteed a high-volume production. As a consequence, the general trend in embedded systems design is shifting from the use of ASIC's to programmable general purpose or domain specific platforms and from hardware design to embedded software design.

The increase in complexity of modern embedded systems in general and embedded software in particular means that the design costs has risen exponentially. Unfortunately, the methodologies used to design embedded systems have not been progressing accordingly which has led to a situation where the complexity of many designs are pushing the limits of what can reasonably be handled with current design methodologies. This gap between what we can build and what we can design is sometimes referred to as the productivity gap [51] or the implementation gap [70].

1.1 Dealing with Complexity

The design of an embedded system, or any kind of system for that matter, can be seen as a series of design decisions taking the system from idea to implementation. These decisions are capture using models – abstract representations of the design. Associated with a model is a "level of abstraction" that specifies the modeling concepts and the semantics of the model. Exactly what these concepts and associated semantics are depends on what one considers the implementation. For example, for a hardware design one might take transistors as the basic concept and for a software design the central concept could be the instructions of some processors instruction set. The number of design decisions

captured by such a model is astronomical even for the smallest designs of practical interest. The lack of efficient ways of making these decisions is the cause of the productivity gap. Within the field of embedded systems design complexity has traditionally been addressed by a combination of abstraction, hierarchy and domains [53]:

Abstraction. An abstraction is a representation of some concept containing only relevant information with respect to some task. An abstraction level is a definition of the modeling concepts and their semantics used to represent a system. The term "level of abstraction" implies that abstraction levels can be ordered. A level of abstraction x is said to be higher than another abstraction level y if x has less information than y for some model and if x is at least as expressive as y. In other words, abstraction levels can be partially ordered. A characterizing feature of an abstraction is that it allows for multiple realizations at a lower level of abstraction. Moving from a high-level of abstraction to a lower level thus usually involves a number of design decisions. Design automation tools are based on abstractions. They take as input a model of a system at a given abstraction level and produces as output another model at a lower level of abstraction.

Hierarchy Hierarchy is a recursive partitioning of a model such that the details of each part are hidden in a lower hierarchical level. Hierarchy reduces complexity by simply reducing the number of entities that needs to be considered.

In some cases it may be difficult to distinguish between hierarchy and abstraction. As an example consider the 32-bit adder box in figure 1.1. Besides the 32-bit adder box, the figure also shows two possible internal models. The behavior of the adder may be given as a network of full adders in which case the adder box is a mere substitution for a network of full adders at a different hierarchical level. Alternatively, the behavior of the adder box could be given as a pair of equations. Expressing the behavior of the adder box as a network of full adders is an example of the use of hierarchy whereas expressing the behavior using simple equations is an example of applying abstraction. From the point of view of the designer both hierarchy and abstraction serves him equally well since only the behavior of the adder is of interest – how the adder is implemented is irrelevant.

Domains (Separation of Concerns) A design can be split into a number of different domains each of which emphasizes a certain aspect of a design. Such

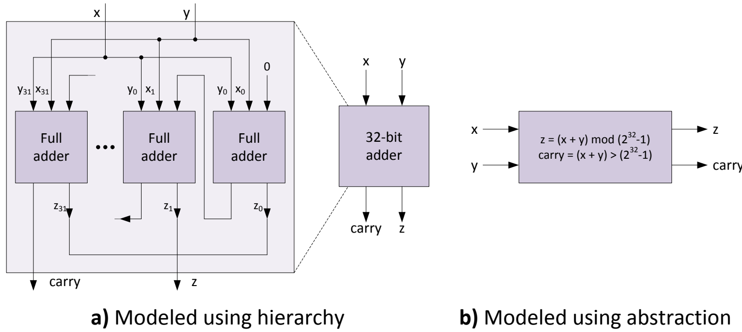


Figure 1.1 – Hierarchy vs. Abstraction. Example showing the difference between hierarchy and abstraction [53]. The figure shows two different meanings of the simple 32-bit adder box. To the left we have a meaning expressed using hierarchy as a network of full adders. To the right the meaning of the box is expressed using abstraction. Here the complex full adder network has been replaced by a simpler model based on algebraic formulae.

domains can be logically analyzed in isolation from other domains [58]. A prime example of separation of concerns is the separation of computation and communication, exemplified by the concept of transaction level modeling (TLM) [18], which has received a lot of attention from both industry and academia. Another example is the separation of time and computation in the synchronous data flow models.

It may be difficult to distinguish between abstraction and domain separation. The reason for this is that in most practical models these two are intimately linked together in the sense that a domain focus is often realized as an abstraction layer. For example, in the context of transaction level modeling two level of abstraction are of interest: the TLM level and the pin-accurate level. The TLM level focuses on the behavior of the design abstracting the details of communication whereas the pin-accurate level includes a detailed description of the communication in addition to the behavior. In principle, however, the domains of computation and communication are independent of these abstraction layers.

1.2 Design Methodologies

A design methodology, or a design process, is a sequence of steps necessary to build something. Besides the obvious goal to build something that is function-

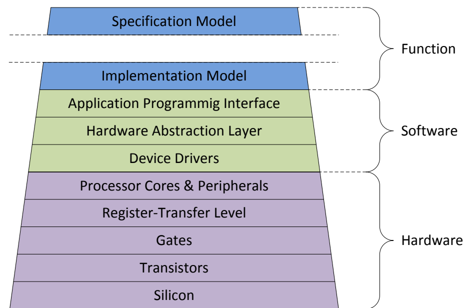


Figure 1.2 – A stack of abstraction layers taking a design from specification to silicon implementation [82]. The gap between specification and implementation represents the manual step of going from an informal specification to a (high-level) formal description.

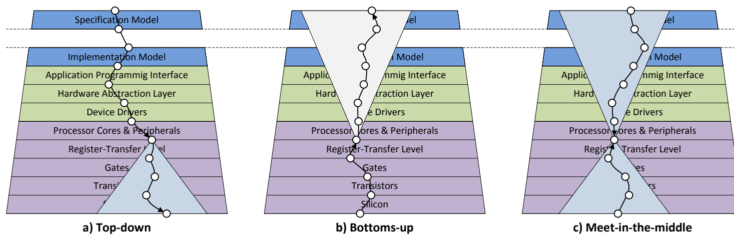


Figure 1.3 – Top-down, bottoms-up and meet-in-the-middle methodologies.

ally correct, a methodology may have other equally important goals: time-to-market, design cost and/or quality. Being explicit about the process is important to ensure that the design team pulls in the same direction. More importantly, a methodology provides a context for understanding the models, methods and tools that are an indispensable part of modern embedded systems development.

Methodologies for embedded systems development are often described relative to a set of abstraction levels linking the specification with the implementation. These are ordered in a stack according to their level of abstraction with the most abstract representation (the specification) at top and the least abstract (the implementation) at the bottom. The number and types of abstraction levels in a stack depends on the concrete methodology. Figure 1.2 shows an example of a typical stack. Relative to such a stack, a design methodology can be classified as a top-down, bottoms-up or a meet-in-the-middle approach:

Top-down (Figure 1.3.a) The starting point of a top-down approach is a specification given at the highest level of abstraction. The specification is gradually refined into an implementation at the lowest level of abstraction by a process called stepwise refinement. Top-down approaches can potentially yield the most optimal implementation because it allows for fine-tuning at all levels of abstraction and does not exclude any portions of the design space. For the same reasons, top-down approaches are also the most expensive and time-consuming.

Bottoms-up (Figure 1.3.b) The starting point of a bottoms-up approach is a set of entities at the lowest level of abstraction. These entities are assembled into richer entities that are abstracted to the second level of abstraction. The resulting entities are again assembled into even richer entities that are abstracted at the third level of abstraction. This process continues until a single entity that, hopefully, implements the specification, emerges at the highest level of abstraction. Bottoms-up approaches inevitably leads to over design because the choices made at each level of abstraction must be conservative enough to ensure that the resulting system will satisfy the specification. In favor of bottoms-up approaches is the fact that the decisions made at the lower levels of abstraction can be re-used in other designs.

Meet-in-the-middle (Figure 1.3.c) A meet-in-the-middle approach is, as the name suggests, a combination of the top-down and bottoms-up approaches and is often referred to as *platform-based design* [83, 58]. In platform-based design, a platform is constructed bottoms-up and combined with a specification of the intended system behavior refined top-down. This enables the trade-off between design space and other design objectives such as time-to-market or cost. This is so because the platform can be designed independently of any actual application which means that the associated design time and cost does not (necessarily) contribute to the cost and time-to-market of the final system. The overwhelming design and setup costs associated with the development of hardware has been one of the primary motivating factors behind the concept of platform-based design.

In practice, all methodologies are meet-in-the-middle or bottoms-up since the entities of the lowest abstraction layer necessarily must be given in order for the methodology to make sense. A number of other conceptual frameworks have been developed for understanding and comparing methodologies for embedded systems design [54, 33, 38] that will not be discussed here.

1.2.1 System-level Platform-based Design

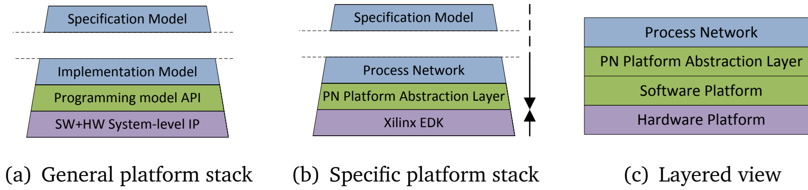
A particularly interesting class of platform-based design methodologies are those where the refined specification, called the application hence forth, meets the platform at the so called *system-level of abstraction*. The system-level of abstraction has traditionally been defined as an abstraction layer above the register-transfer level that comprises software in addition to hardware [37]. At this level, a hardware platform is described as a netlist of high-level components such as processors, memories and interconnects. In addition to the hardware, a platform may also consist of software such as operating systems and other middleware components, executing on the programmable processors of the hardware platform.

The task of choosing the right platform and mapping of the application onto the chosen platform is central to platform-based design. Ideally, the application is a pure description of behavior containing no implementation details meaning that the performance characteristics of the final system are determined by the choice of platform and mapping alone. Much research has been devoted to the problem of exploring the design space to determining an optimal mapping and/or platform using both formal (e.g. [46, 45, 79]) and simulation-based approaches (e.g. [64, 22]).

Many of these approaches uses the concept of abstract services to link abstract representations of applications and platforms [43, 92]. In this view, an application is seen as a service consumer whose function or behavior is defined by the temporal ordering of a set of service requests and platforms as the provider of services. Different platforms may provide different implementations of the same services. Using simulation or formal methods the performance of an application with respect to a particular platform and mapping can be evaluated and quantitatively compared to alternative platforms and mappings.

1.3 Scope of Work

In this thesis, we will focus on a class of system-level platform-based design methodologies that can be described using three different level of abstraction as shown in Figure 1.4(a). The lowest of level of abstraction is the system-level where a platform is designed using a component-based design approach. On top of the system-level we have an API level that implements a specific programming model, to the highest abstraction level, the implementation model.



(a) General platform stack (b) Specific platform stack (c) Layered view

Figure 1.4 – Organization of the design methodology used in this thesis.

Most of the experiments and case studies presented in this thesis will be given relative to a methodology where platforms are created using the Xilinx Embedded Development Kit design flow and applications are specified using the process networks model of computation (MoC). In this methodology, an application is tied to a platform by means of a platform abstraction layer providing an implementation of the process network MoC. As shown in Figure 1.4(b) the platform is created bottoms-up where as the application and the platform abstraction layer is created top-down. For reference, Figure 1.4(c) shows a layered view of systems designed using the methodology. In the rest of this thesis we will use the term "system" for a complete system comprising both a platform and an application.

Even though most examples will be given relative to this specific methodology most of the principles and methods presented in this work are independent of the actual methodology and may be applied to other methodologies as well.

1.3.0.1 Component-Based Design

Component-based design is based on the idea that system can be decomposed into components. A component is a self-contained part or sub-system that can be used as a building block in the design of a larger and more complex system. It provides specific services to its environment across well-specified interfaces. In the context of embedded systems design, a component is a piece of functionality implemented as software, hardware or a combination of the two. Components are units of composition that can be composed into new components with richer functionality using hierarchy. Ideally, a component should be re-usable and may provide some degree of customizability.

In a component-based design approach, the task of developing components and the task of assembling systems are independent of each other. This separation is what makes component-based design appealing for embedded systems

development because the development of the components is not part of the critical time-to-market. Obviously, the truth of this depends on the availability of components which implies that component-based design disfavors design with very specialized needs [52]. In general, component-based design is appropriate when the market is unable to support expensive design costs or when ones competitive advantage is not in the design of platforms [85]. The strength of the component-based design is the natural focus on component re-use.

Given is a library of (re-usable) components and the objective is to assemble these into a network that implements the desired behavior (given in the specification). The process of integrating a component into a design can be divided into a series of steps:

1. The first of these is *component matching*. Here, the functional requirements of the component is matched against the functions offered by the available components. The result of this step is a set of components that may satisfy the functional requirements.
2. The second step is *component selection* and involves selecting the most appropriate component with respect to any non-functional constraints placed on the design.
3. The third step is *component integration* where the component is integrated (i.e. connected) into the design.

The resulting design is characterized by its *allocation*, *configuration* and *topology* where the allocation specifies the number and types of components in the design, the configuration specifies, for each allocated component, the value of any required parameters and the topology defines the communication infrastructure of the design.

1.3.0.2 Programming Models & Abstraction Layers

Compared to single processor systems, heterogeneous multi-processor systems are difficult to program. In the absence of a platform-wide programming solution, shielding the application developer from the details of the platform, the developer must program each processor individually. This is problematic for several reasons:

- First, the application becomes dependent on the particular platform making re-use hard and changes in the platform, application and/or mapping difficult.

- Second, a partitioning and mapping of the application must be decided upon before any code is written which leaves little room for errors and may lead to resource waste due to over-engineering.
- Finally, in addition to implementing the application, the application developer must also worry about low-level processor-to-processor interfacing that in turn will require intimate knowledge of the platform in particular and low-level programming in general.

Through the use of a parallel programming model (e.g. TTL [94], YAPI [28], OpenMP [1]) it is possible to abstract away different aspects of the platform, such as communication and synchronization, and thus effectively de-couple the application from any actual platform [55, 74, 36]. A programming model can be provided as an integral part of a programming language or as an add-on in the form of an API. In most cases, a given platform does not directly implement a given programming model and thus an abstraction layer, realizing the abstractions of the programming model, is needed. An abstraction layer can be application-specific so that it only provides support for the functionality that is actually used by a particular application. The opposite of an application-specific layer is a general purpose layer where functionality is provided to support a broader range of applications.

A general purpose layers are constructed bottoms-up whereas special purpose layers are constructed top-down.

1.3.0.3 Process Networks

For embedded systems, programming models based on the Kahn Process Network (KPN) [57, 62] model of computation have been studied intensively. An application modeled as a KPN consists of processes communicating and synchronizing using unidirectional FIFO channels. In theory, the capacity of the channels is unbounded and writing to them is a non-blocking operation. In this work, we will consider a more practical version with bounded channel capacity and blocking write that we will refer to simply as a *process network*. The process network MoC is especially well suited for capturing streaming (e.g. image processing) applications and has previously been shown to be a suitable representation for efficiently mapping applications onto multi-processor platforms [5, 91, 77].

The process network model of computation is favored because interactions between processes are explicitly modelled by means of channels. In other models of computation, interactions are often implicit which means that program

analysis is required to determine dependencies amongst the behavioral entities of an application.

1.4 Problem Statement

Tools supporting exploration and synthesis must be able to reason about the capabilities of the target platform. Having knowledge of what can be done with a particular platform and how to do it is fundamental for refinement and, similarly, knowing what a platform can do and the cost associated with doing it is fundamental for design space exploration. If the target platform is known in advance then this knowledge can be embedded within the algorithm used. If, on the other hand, the target platform is not known in advance then the algorithm must be independent of any particular platform and a method for analyzing the actual capabilities of a given platform must be employed. In the case of tools targeting platforms built from re-usable components little or nothing can be assumed about capabilities of the actual platforms. This is so because the components are essentially black boxes.

By cleverly choosing and imposing a set of restrictions on the supported platforms it is often possible to significantly reduce the required analysis effort for a given task. This is so because, given an appropriate set of restrictions, it is possible to make assumptions about *all* possible target platforms. For example, a tool for evaluating the mapping of an application onto an execution platform might assume that all processing elements are connected to the same interconnect so that communication between the parts of the application mapped to different processing elements is always possible. The tool will thus be limited to platforms where the processing elements are fully connected. Similarly, a tool for compiling a high-level specification of an application onto a heterogeneous multi-processor platform might assume that standard C compilers exists for each of the different processors thus limiting the use of the tool to platforms with processors for which C compilers are available. Exactly what kind of assumptions, and thus restrictions, are useful depends on the task at hand and, of course, on whether or not the restrictions are justifiable. In many cases, it is even possible to completely trivialize the analysis so that it becomes an implicit part of the tool or the algorithm. A compiler for a class of very similar processors (e.g. ARM) that may generate different results depending on exactly which processor is the target is an example of this.

The point here is that whenever assumptions are required in order for a tool to avoid having to automatically discover the capabilities of the actual target platform (which might not be trivial) then these assumptions impose restric-

tions on the targeted platforms which in turn limits the design space. The alternative to imposing restrictions is to allow errors in the tools. For example, the mapping tool might come up with a mapping that is infeasible because two dependent parts of the application are mapped so that their dependency is violated. If one is unable, or unwilling, to accept imposing restrictions on the platform then nothing can be assumed about the platform and, consequently, any information required for decision making must be extracted from the platforms themselves. Extracting this information is, in general, not trivial and the subject of this thesis. In other words: there exists a trade of between how easy it is to design a tool and the usefulness of the tool.

We believe that a number of tools for key problems, primarily within the platform-based design, could benefit from improved analysis techniques for analyzing the capabilities of platforms and systems. By improving these techniques it is possible to develop more generally applicable tools supporting a broader range of architectures and design styles. In this work, we will focus on the task of extracting information about the capabilities of platforms that either is, or can be seen as, a network of components without imposing restrictions on the type of the components or the topology of the network.

1.4.1 Platform Capability Analysis Challenges

In the following, we will give two examples of some of the problems that makes analyzing the capabilities of a platform a challenge.

1.4.1.1 Localization of Services

Since modern embedded platforms are increasingly based on heterogeneous multicore architectures, the services offered by the different programmable elements are very likely to differ. As a consequence, talking about the services provided by the platform as a whole is too simplistic, instead we have to focus on the services provided at the different *points-of-contact* between the platform and its application. Example 1 illustrates this.

Example 1.1 Figure 1.5 shows a relatively simple multi-processor platform created using the *Xilinx Embedded Development Kit* tool. The platform is taken from an official application note published by Xilinx for demonstrating multi-processor designs using the tool. The platform consist of two processors, a Mi-

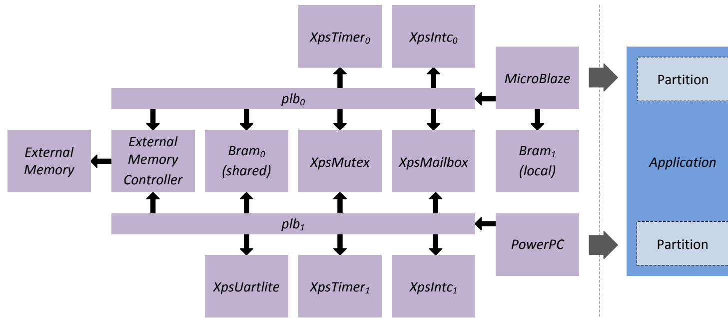


Figure 1.5 – Multi-processor platform created using the *Xilinx Embedded Development Kit*. Source: Xilinx Application Note, [12]

croblaze and a PowerPC, and a number peripheral cores. Each of the processors are connected to a private bus through which each of them may access a subset of the peripheral in the platform. The sets of peripherals accessible from each of the two processors are given below:

From Microblaze	From PowerPC
`Microblaze`	`PowerPC`
`Plb`$_0$	`Plb`$_1$
`XpsTimer`$_0$	`XpsTimer`$_1$
`XpsIntc`$_0$	`XpsIntc`$_1$
`External Memory Controller`	`External Memory Controller`
`External Memory`	`External Memory`
`Bram`$_0$	`Bram`$_0$
`XpsMutex`	`XpsMutex`
`XpsMailbox`	`XpsMailbox`
`Bram`$_1$	`XpsUartlite`

It is obvious that the two sets are different. For example, only software executing on the PowerPC will be able to use the UART (XpsUartlite) and although the platform features two timer peripherals only one can be accessed from each processor. ■

Peripherals accessible from several points-of-contact may be accessed differently depending on the point of contact. A dual-port memory, for example, can be mapped to different segments of the address spaces of two different processors.

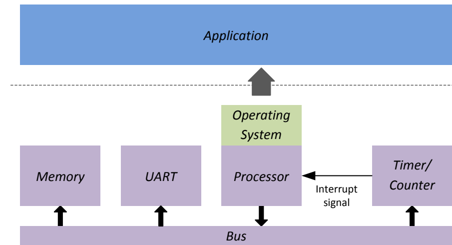


Figure 1.6 – A simple single processor platform. The Timer/Counter peripheral is not part of the capabilities of the platform since it is used internally by the operating system.

1.4.1.2 Resources

The services offered by a network of components comprising a platform at a given point-of-contact is not simply the sum of the services accessible from that point. Some services are only available in finite quantities and may be used internally by the platform as is illustrated by the next example.

Example 1.2 Consider the simple platform of figure 1.6. The hardware of platform consists of a single processor, a bus and a handful of peripheral cores including a timer/counter. Besides the hardware the platform also includes an operating system featuring time-sliced multiprocessing executing on the processor. In order to provide multiprocessing, the operating system requires (exclusive) access to the timer/counter and, consequently, the services provided by the timer/counter peripheral is not part of the capabilities the platform. ■

1.4.2 Case Study: Xilinx Embedded Development Kit

Field Programmable Gate Arrays (FPGA) is an example of a class of platforms that provides a high degree of customizability. The use of FPGA's have traditionally been associated with a long time-to-market compared to the use of microcontrollers. The reason for this is that FPGA's must be "programmed" using hardware description languages such as VHDL or Verilog. Today, however, tools exists that can generate synthesizable designs for FPGA's from higher-level models eliminating the need for the hardware description languages. Two such tools are the *Embedded Development Kit* [97] from Xilinx and the *System-on-a-Programmable-Chip Builder* [9] from Altera. These tools allows for an

FPGA to be configured as a customized computing platform complete with programmable processors, peripherals and sophisticated interconnects. Another tool is the System Generator [98] that can be used as a backend to Simulink for realizing DSP algorithms in programmable logic. In the terms of platform-based design, these tools may represent an abstraction layer above the FPGA platform.

Of special interest to this thesis are the before mentioned tools for configuring an FPGA as a more general purpose computing platform. Next, we will present a few shortcomings of the Xilinx Embedded Development Kit tool that are representative for the more general issues addressed in this thesis.

1.4.3 Xilinx The Embedded Development Kit

The Xilinx Embedded Development Kit (EDK) consists of a number of integrated tools for developing embedded processor systems based on the Microblaze [101] and PowerPC [99] processors targeting the Xilinx line of FPGA's.

The *Platform Studio* tool is a graphical system editor used for assembling computing platforms using a rich library of existing IP-cores. Among the provided cores are a number of different buses and point-to-point links that can be mixed in a platform to provide a custom communication infrastructure. Additional cores, written in VHDL or Verilog, can easily be added to the library if they implement an interface to one of the supported buses or point-to-point links. A core in the library is organized as a set of parameterized VHDL or Verilog files and a high-level description of its interfaces and possible configurations. Platform Studio generates a hardware specification, in the form of a Microprocessor Hardware Specification (MHS) file, specifying a set of IP core instances, their configurations and the connections between them.

A deployable bitfile is generated on the basis of an MHS file using the *Platgen* tool. This tool first generates a description of the system in a hardware description language and then it uses the Xilinx ISE implementation flow to synthesize the bitfile. Because of the hardware synthesis involved, running Platgen can be quite time consuming. It can easily take up to 15 to 20 minutes to synthesize even relatively small designs. Before attempting to generate a bitfile, the MHS file is subjected to a verification/consistency check using TCL scripts.

The EDK also comprises a number of tools supporting the development of software for hardware platforms generated using the Platform Studio and Platgen tools. The most interesting of these tools is the *Libgen* tool – a tool for generating libraries and drivers for the processors of a platform. This tool takes as input a MHS file specifying a hardware platform and a Microprocessor Software

Specification (MSS) file created by hand or using the Platform Studio graphical IDE. The MSS file specifies the drivers associated with peripherals, which peripherals should be used for standard input/output and other software related features. Libgen can also be used to configure an operating system for each processor in the platform. The EDK is shipped with a single operating system called *Xilkernel* [102]. The Xilkernel operating system can be configured to include a number of optional features such as semaphore and message queue support.

1.4.3.1 Issues

Using the tools of the EDK to create sophisticated systems is surprisingly easy. There are, however, some limitations in the tools that indirectly restricts the freedom of the designer. More specifically, the analysis capabilities of the Libgen tool are too limited and based on assumptions about the IP cores used as well as on how the communication infrastructure is realized. In order to make use of the Libgen tool these assumptions must be justified which in practice limits the freedom of the designer.

Based on our experience with the EDK, we believe that the identification of the set of peripherals reachable from a processor is done in the following way: First the processor in question is located. Next, the buses connected to the processor are enumerated. Finally, the slave attachments of these buses are added to set of reachable peripherals. For this to be possible, the analysis back-end must know which components represent buses and processors. It should be noted that this is speculation as the documentation [100] does not describe the analysis capabilities of the tools in any kind of detail. Our experiments shows that the analysis back-end of the EDK fails to locate peripherals that are indirectly connected the address space of a processor through a bridge. More specifically, the Libgen tool fails with a somewhat arcane error message stating that there was an error in the address specification.

Furthermore, the component-model underlying the EDK tool does not model resources. For example, the Xilkernel operating system requires a timer of the type XPS Timer/Counter or *fit timer* to provide it with the ticks necessary to implement time-sliced multitasking. The XPS Timer/Counter is a standard bus-mounted peripheral accessible via a set of memory mapped registers. In order to work, the timer/counter must be connected so that its memory mapped registers are accessible from the processor running the operating system. Also, the interrupt port of the timer/counter must be connected to the processor either directly or indirectly via an interrupt controller. Our experiments has shown that it is possible to create a dual processor design with two Xilkernel operating systems using the *same* timer/counter instance. This error is not caught at design

time. One could argue that, in theory, two identical operating systems could use the same timer for generating ticks. This, however, is most likely not the case with the Xilkernel as the time between ticks is a customizable parameter.

From the point of view of the end-user, the real problem is not so much the shortcoming of the analysis back-end but rather the lack of proper error detection and reporting. Debugging a complex system can be a very time consuming process. This is especially true if the error does not show up at design time so that hardware synthesis must be re-run for each iteration of the debug cycle.

1.5 Contributions

We have developed a formalism called the *Service Relation Model* for describing systems composed of components based on the novel concepts of service aggregation and service exchange relations. The formalism can be used to create re-usable abstract descriptions of hardware and software components that may be combined into models of platforms and systems. The resulting models may be analyzed to obtain information about service availability and flow. We consider the Service Relation Model and the associated analysis method the main contribution of this work.

To demonstrate the usefulness and versatility of the proposed model we have developed a number of procedures and associated proof-of-concept tools for assisting the designer in tasks related to component integration and low-level programming:

- **Consistency Checking.** A procedure for checking the consistency of a platform or system modeled as a network of re-usable components with respect to service and resource availablity. Using our consistency checking procedure, we are able to capture some inconsistencies that will show up as run-time errors in industry standard component-based design tools.
- **Automated Programming.** A procedure for automatically generating an application-specific abstraction layer implementing the communication infrastructure of an application given as a process network. The procedure uses a Service Relation Model representation of a platform for determining different implementation alternatives and for code generation purposes.
- **Automated Design Generation.** A procedure for automatically transforming an inconsistent model of a system into a consistent model by allocating, configuring and inserting new components into the model. This

procedure has a number of different uses in the context of "automated design generation" and generalizes the analysis and decision making parts of the procedure for automated programming.

All of the three procedures are central contributions to this work. The proof-of-concept tools supporting the different procedures are all based on a common framework, called the `xSRM` framework, for working with service relation models that we also consider an important, albeit theoretically less interesting, contribution.

1.6 Thesis Outline

This thesis is organized as follows: In chapter 2, the basic concepts of the Service Relation Model and its associated analysis method are presented. In chapter 3, a number of additional concepts are added to the basic Service Relation Model that forms the basis of the procedure for consistency checking presented in the same chapter. This chapter also contains an overview of related work. The procedure for automated programming is presented in chapter 4 followed by the presentation of the procedure for automated design generation in chapter 5. Chapter 6 contains the concluding remarks.

Chapter 2

The Service Relation Model

The Service Relation Model, presented in this chapter, is an abstract component-centric model used to extract information about the capabilities of a system given as a network of components. The model does not impose any restrictions on what a component is and may be used to describe platforms and systems comprised of both hardware and software. The Service Relation Model is an analysis model, as opposed to a composition model, meaning that it is primarily intended to analyze systems created using other tools and models. A key feature of the model is that it allows for re-use of component descriptions by separating the definition of components from any actual model.

This chapter is organized as follows: The first section contains an informal introduction to the basic concepts of the Service Relation Model. After the informal introduction the basic concepts are formalized with the intent of providing a precise and unambiguous presentation.

2.1 Informal Presentation

Most models supporting component-based design of execution platforms (e.g. [97, 9]) employs the basic concepts of components, interfaces and connections.

In these models, a component represents a physical entity such as a processor, bus or a block of memory. Likewise, an interface represents a physical connection point where a component may be connected to another component with a matching interface. The *Service Relation Model* extends this line of thought to also include conceptual components and conceptual connections (called *relations*). In other words, the components of the service relation model need not necessarily have a physical manifestation and their relationships (connections) can be conceptual as well as physical.

The components of the Service Relation Model are defined by the service or services they offer to other components. A service can be any kind of work that a component carry out on behalf-of another component. As an example, consider a bus: the ability of a bus master attachment to issue read and write requests to slave attachments can be seen as a service provided by the bus to the master attachment. A component representing a function in the C programming language might be considered to provide other components with the service that it describes. The service relation model explicitly separates the concept of a service and its provider (a component).

The Service Relation Model supports only one kind of binary relations called *service exchange relations*. A service exchange relation is characterized by facilitating some kind of exchange of services between two components. The relation between a bus master attachment and a bus is an example of such a relation since the bus master attachment may request service from the bus. Similarly, a software module may have a service exchange relationship with its processor because it, conceptually, is serviced by the processor. On the other hand, a relation stating that two components are identical is not a service exchange relation because no exchange of service can be attributed to it. A service exchange relation consists of three parts: the relation itself and two roles (the subject and the object). A component in the Service Relation Model exposes its services through a set of interfaces each of which is associated with exactly one role of a predefined service exchange relation. Two components may be connected to each other, using a service exchange relation, only if one of the components define an interface with the subject role of the relation and the other define an interface with the object role of the relation (see Figure 2.1). The flow of service through service exchange relations is unidirectional and service always flow from the object interface to the subject interface.

Besides the service exchange relation the model supports two additional relations: *service import relations* and *service export relations*. In contrast to service exchange relations, the service import and export relations are used inside components to relate services and interfaces to each other. An export relation states that a given service is exported to a given interface. Recall that we said that the services of a bus are provided *to* its master attachments. In the Service Relation

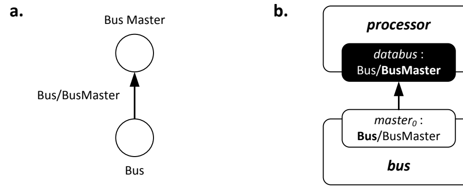


Figure 2.1 – **a.** A service exchange relation modeling the relationship between a bus master and the bus. **b.** The same relation used to relate the data bus interface of a processor with a bus (note: role names are not included but suggested with the coloring of the boxes representing interfaces).

Model, this is modeled by an export relation relating the bus services with the interfaces representing the bus role of a bus-master/bus service exchange relation. A service exported to an interface is said to be *available* at that interface. Services available at an interface are also considered to be available at the interface of another component if the two interfaces are (properly) connected by means of a service exchange relation.

The import relation facilitates *service aggregation* which essentially is an "available through" relationship between any services available at an interface and a service defined in the component. A service representing the ability to execute software on a RISC processor can be said to aggregate services of its attached data bus since the services of the bus may be invoked using the memory LOAD and STORE instructions of the processor. A driver is another example of service aggregation. The service offered by a driver aggregates the service offered by the underlying device. A service aggregated by another service is considered available at that service and at any interfaces and services where the aggregating service is considered available.

2.1.1 Example: Hardware Platform

As an example, we will consider a simple hardware platform consisting of a memory, a bus, a processor core and a floating point co-processor (FPU). These four components are organized in the following way: The data bus interface of the processor is attached to the bus as a master attachment and the memory as a slave attachment. The FPU is an optional part of the processor core and thus the instruction set of the processor includes instructions for operations on floating point numbers. If the FPU is not included these instructions are undefined.

Figure 2.2 shows a graphical representation of a service relation model of

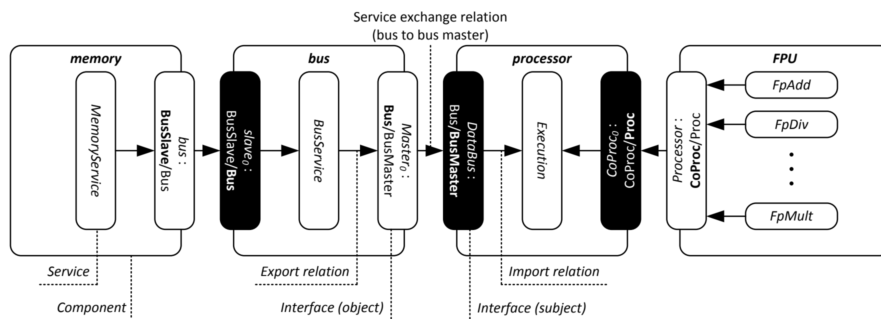


Figure 2.2 – A simple system described in the Service Relation Model.

the platform. In the graphical notation, a component is represented as a box with its name printed on it. An interface is represented as a box placed on the border of a component. Object interfaces are represented using white boxes and subject interfaces using black boxes. The text printed in the boxes representing interfaces (*name* : *relation*) gives the name of the interface as well as the relation to which it belongs. In this thesis, all relations names follows the same generic naming convention based on the types of the components that it relates. For example, the relation relating a bus slave to a bus is named *BusSlave/Bus* and, similarly, the relation relating a co-processor to the processor is named *CoProc/Proc*. Notice that the component with the role of the object is always mentioned first. Service exchange relations are represented as arrows connecting an object interface with a subject interface. Services are represented as boxes completely contained within the bounds of a component. A service is associated with a name printed inside it. Import and export relations are represented using arrows connecting interfaces to services and services to interfaces respectively.

Next, the meaning of the different parts of the model will be explained:

- **Memory:** The memory has a single object interface of type `BusSlave/Bus` meaning that it may participate in a single `BusSlave/Bus` relation in the role of the bus slave. The memory provides other parts of the system with the ability to read and write its data through its bus interface. The ability of the memory to be read and written is modeled using a service called `MemoryService` that is exported through the bus slave interface.

- **Bus:** The bus component has two interfaces – a subject interface of type `BusSlave/Bus` and an object interface of type `Bus/BusMaster`. Consequently, the bus may participate in a maximum of two relations (in the

role of the bus). The function of the bus is to provide the bus master with (read/write) access to the bus slave. This is modeled using the service `BusService` that aggregates services available on $\texttt{slave}_0$ interface and is exported to the $\texttt{Master}_0$ interface.

- **Processor:** The processor provides the capability of executing software. This capability is modeled by means of a single service, called `Execution`, that represents the possible execution of one instruction of the processors instruction set. The service can import services from the data bus interface (where the bus is connected) because the data bus is accessible by means of the processors LOAD and STORE instructions. Similarly, the `Execution` service may import services from the co-processor interface because these can be accessed through the special FPU instructions of the processor. At this time, `Execution` service itself is not exported anywhere.

- **FPU:** The FPU component provides the processor with the ability to execute a set of predefined floating point instructions. Each of these instructions is modeled using an appropriately named service exported to the connected processor through the `CoProc/Proc` interface of the FPU component.

The components are related to each other using three different service exchange relations:

- `Bus/BusMaster` This relation represents a bus masters ability access services provided by a bus by issuing read and write requests to it.

- `BusSlave/Bus` This relation represents the ability of a bus to access services provided by slave peripherals by forwarding read and write requests.

- `CoProc/Proc` This relation represents a processors ability to use a coprocessor. The understanding here is that only services representing a predefined set of FPU instructions can be propagated through the relation.

It is important to note that even though the names of the components, services and service exchange relations are fairly generic the intention is that they refer to actual components rather than abstract classes. For example, the processor component represents a concrete processor rather an unspecified instance of a generic processor type and, similarly, the `BusSlave/Bus` service exchange relations represents a relation between a specific type of bus and components implementing a compatible slave interface.

Using the information asserted in the model and our knowledge of the meaning of the different concepts we can determine the availablity (or accessability) of the 11 services in the model. Below the services available at the different interfaces and service of the model is given:

$$
\begin{aligned}
\mathrm{SA}[\texttt{memory.MemoryService}] &= \{\ \texttt{memory.MemoryService}\ \}\\
\mathrm{SA}[\texttt{memory.bus}] &= \{\ \texttt{memory.MemoryService}\ \}\\
\mathrm{SA}[\texttt{bus.slave}_0] &= \{\ \texttt{memory.MemoryService}\ \}\\
\mathrm{SA}[\texttt{bus.BusService}] &= \{\ \texttt{memory.MemoryService},\ \texttt{memory.BusService}\ \}\\
\mathrm{SA}[\texttt{bus.Master}_0] &= \{\ \texttt{memory.MemoryService},\ \texttt{memory.BusService}\ \}\\
\mathrm{SA}[\texttt{processor.DataBus}] &= \{\ \texttt{memory.MemoryService},\ \texttt{memory.BusService}\ \}\\
\mathrm{SA}[\texttt{processor.Execution}] &= \{\ \texttt{memory.MemoryService},\ \texttt{memory.BusService},\\
&\qquad \texttt{processor.Execution},\ \texttt{FPU.FpAdd}\ ,\\
&\qquad \texttt{FPU.FpDiv},\ \texttt{FPU.FpMul},\}\\
\mathrm{SA}[\texttt{FPU.Processor}] &= \{\ \texttt{FPU.FpAdd},\ \texttt{FPU.FpDiv},\ \texttt{FPU.FpMul}\}\\
\mathrm{SA}[\texttt{FPU.FpAdd}] &= \{\ \texttt{FPU.FpAdd}\ \}\\
\mathrm{SA}[\texttt{FPU.FpDiv}] &= \{\ \texttt{FPU.FpDiv}\ \}\\
\mathrm{SA}[\texttt{FPU.FpMul}] &= \{\ \texttt{FPU.FpMul}\ \}
\end{aligned}
$$

The sets above describe the availablity of the services in the model and this can be considered an abstract representation of the capabilities of the platform. From the information contained in the sets we may, for example, infer that software executing on the processor can access the memory and it may use the floating point instructions provided by the attached co-processor because the services representing these capabilities are available through the `Execution` service representing the instruction set of the processor.

2.1.2 Example: Software

The next example will show how the Service Relation Model can be used to model software and how such a model can be combined with a model of a hardware platform.

As an example of an application, we will use a simple program for solving quadric equations. A quadric equation, $ax^2 + bx + c$, is defined by means of three real values (a, b and c) and has two solutions (r_0, r_1)) given by:

$$r_0, r_1 = \frac{-b \pm \sqrt{d}}{2a}, \qquad D = b^2 - 4ac$$

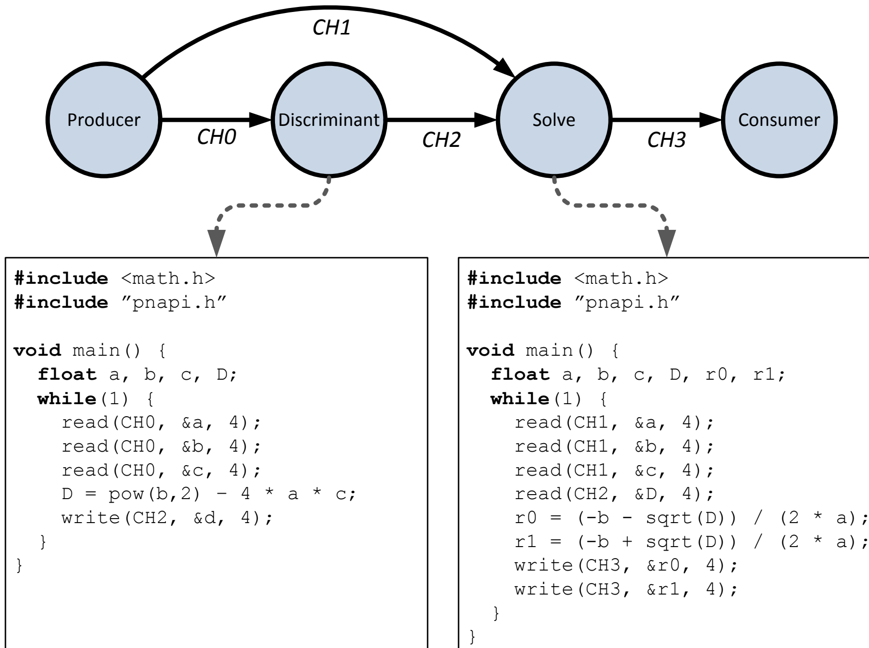


Figure 2.3 – Process network for solving quadric equations.

where D is the discriminant.

Figure 2.3 shows a process network for a simple application for solving quadric equations. The network has a four processes (`Producer`, `Discriminant`, `Solve` and `Consumer`) and four channels (`CH0`, `CH1`, `CH2` and `CH3`). The `Producer` process produces data for the process network by continuously writing three values (a, b and c) to both of the channels `CH0` and `CH1`. The `Discriminant` process reads the values of the variables a, b and c from channel `CH0`, computes the discriminant and writes the result to channel `CH2`. The `Solve` process reads the values of a, b and c from channel `CH1` and the value D of the discriminant from channel `CH2`. The process then computes the two solutions to the equation and writes the result to channel `CH3`. The last process, `Consumer`, continuously reads pairs of values representing solutions to quadric equations from the channel `CH3`.

A C code implementation of the two processes `Discriminant` and `Solve` is also shown in figure 2.3. Each of the two processes are given as a standalone C program. Both processes uses functions from the `math` library and from a li-

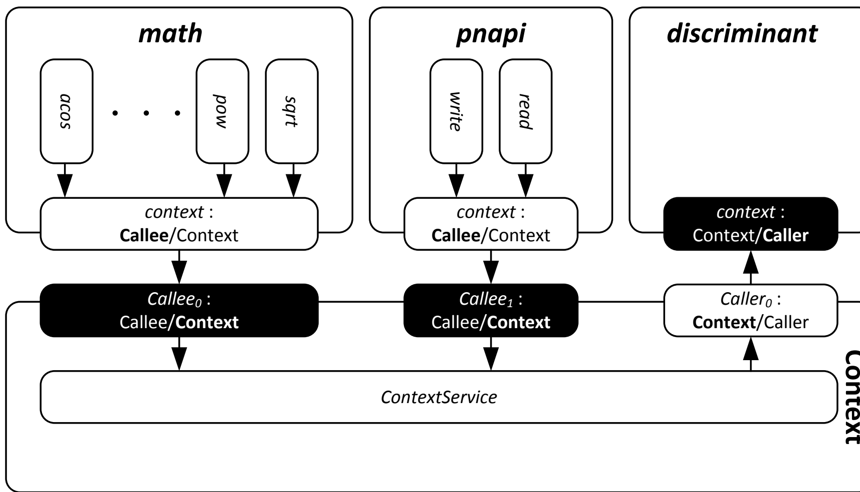


Figure 2.4 – Service relation model of the C program implementing the discriminant process.

brary `pnapi` that implements a simple API for accessing channels in a process network. The channels of the process network are accessed by means of two function `read` and `write`. Figure 2.4 shows a service relation model of the C program implementing the discriminant process. The model consists of four components representing the `math` library, process network API, the discriminant main function and a context. The context components represents the "execution context" of a program – the conceptual infrastructure through which the different entities of the program communicates. The components of Figure 2.4 are related to each other by means of two different service exchange relations:

- **Callee/Context.** The Callee/Context service exchange relation models a relation between two components where the object provide C functions that are made accessible through a context (the subject).
- **Context/Caller.** The Context/Caller service exchange relation models a relation where the subject can access C functions available in a context (the object).

The context component and the two relations allows us to model the accessibility of functions in a C program. From Figure 2.4 we may infer that the discriminant component has access to the services of both the `math` library and the process network API.

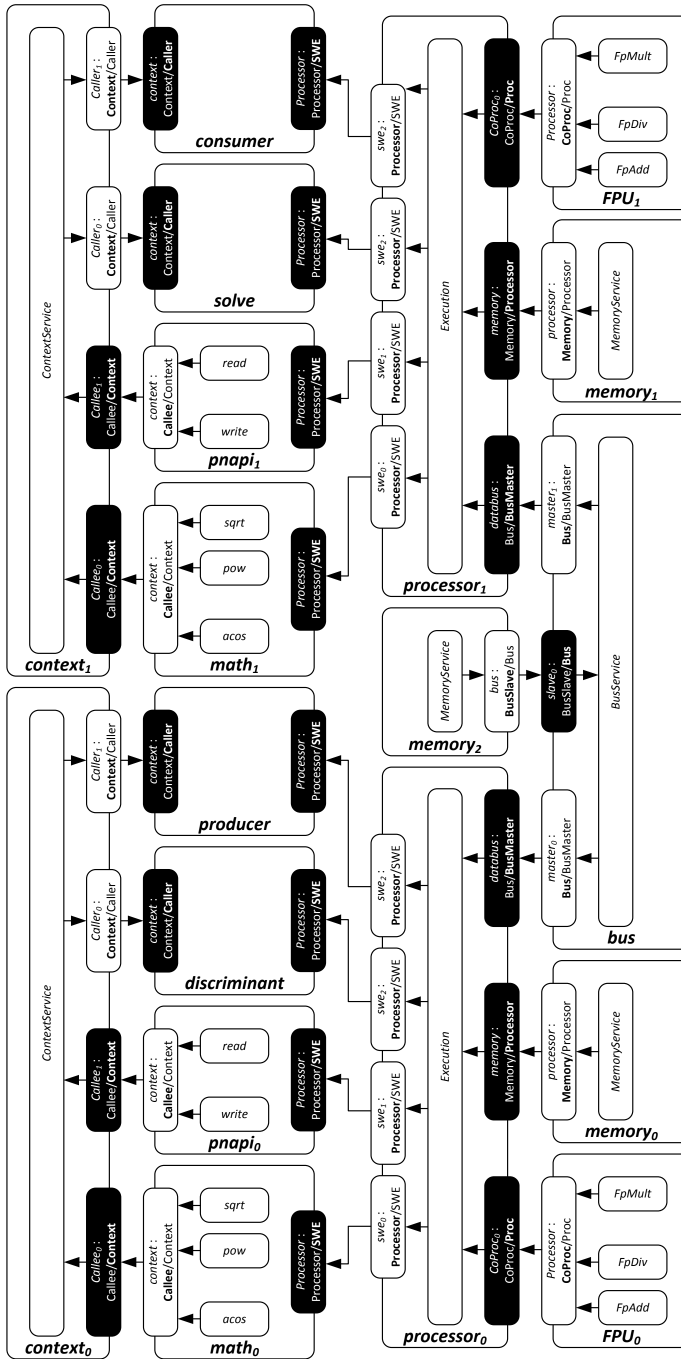


Figure 2.5 – Service relation model of the process network of Figure 2.3 mapped to a simple dual-processor platform.

Figure 2.5 shows a service relation model of the process network from Figure 2.3 mapped to a simple dual-processor platform. The platform consists of two processors ($\texttt{Processor}_0$ and $\texttt{Processor}_1$) connected to a shared bus (Bus). Besides the bus, each of the processors are also connected to a local memory ($\texttt{Memory}_0$ and $\texttt{Memory}_1$) and a floating point co-processor ($\texttt{FPU}_0$ and $\texttt{FPU}_1$). A third memory, $\texttt{Memory}_2$, is connected to the shared bus and is used for inter-processor communication.

In the model, the Producer and Discriminant processes are mapped to $\texttt{Processor}_0$ and the other two processes, Solve and Consumer, to $\texttt{Processor}_1$. The programs executing on the two processors of the platform are modeled using two context components to which the components representing the four processes are mapped. Each context is related to a total of four other components – two processes, a component representing the math library and a component representing the process network API. Besides being related to a shared context component, the four components are also related to the processor that they execute on. This is done by means of the service exchange relation Processor/SWE where SWE is short for "software entity". These relations provides the means for relating the capabilities (services) offered by a processor through its instruction set to the software entities in the program executing on the processor.

Using the information embedded in the model of Figure 2.5 we may infer a number of interesting things about the system. First, we can check if the processes are properly mapped meaning that they have access any services they may require to function properly. Below the sets of services available at the Context and Processor interfaces of the Discriminant component are given:

$$
\begin{aligned}
\mathrm{SA}[\texttt{discriminant.Context}] = \{\, & \texttt{context}_0\texttt{.ContextService},\\
& \texttt{math}_0\texttt{.acos},\ \texttt{math}_0\texttt{.pow},\ \texttt{math}_0\texttt{.sqrt},\\
& \texttt{pnapi}_0\texttt{.write},\ \texttt{pnapi}_0\texttt{.read};\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{SA}[\texttt{discriminant.Processor}] = \{\, & \texttt{processor}_0\texttt{.Execution},\ \texttt{memory}_0\texttt{.MemoryService},\\
& \texttt{FPU}_0\texttt{.FpAdd},\ \texttt{FPU}_0\texttt{.FpDiv},\ \texttt{FPU}_0\texttt{.FpMult},\\
& \texttt{bus.BusService},\ \texttt{memory}_2\texttt{.MemoryService}\ \}
\end{aligned}
$$

Looking at these sets we can conclude that the process is indeed properly mapped because it has access to an Execution service needed for basic execution, the pow service needed to compute the value of the discriminant, the read and write functions for accessing channels and, finally, a set of services needed for doing computations on floating point values. In the model, we assume that the

process network API implements the communication infrastructure of the process network. This means that all processes that can access a pair of `write` and `read` functions can somehow, magically, communicate. Obviously, this imposes some requirements on the two components representing the process network API's. Assuming that communication between two process network API's can be accomplished by means of shared memory we can check if communication is in fact possible. To do this, we compute the intersection of the sets of services available at the `Processor` interfaces of the two process network API components. If the result contains one of the `MemoryServices` in the system then a memory exists that can be accessed from both components and, consequently, communication is possible.

This example shows one way to model software using the Service Relation Model. Depending on the purpose of the model other ways may be more appropriate.

2.2 Basic Concepts

The Service Relation Model is based on a number of novel concepts that will be explained in this section.

Fundamental for this thesis and the Service Relation Model is the concept of *service aggregation* which was conceived during the early phases of the project ending in the writing of this thesis. Service aggregation is a relation between two services stating that one is accessible through the other. These kinds of relations is the basic building block of the Service Relation Model. Service aggregation is, per definition, a transitive relation meaning that if we know that service s_0 is aggregated by service s_1 and that service s_1 is aggregated by service s_2 then we may infer that s_0 is also aggregated by (accessible from) s_1. Using service aggregation relations, the services of some system can be arranged in a graph representing the flow of service in the system. Using such a graph, the service flow of a model can be analyzed with the aim of determine which services are accessible where.

When dealing with platforms or systems given as a network of re-usable components information about service aggregation is not readily available. This is so because the (re-usable) components are defined in isolation of each other and, in general, are not based on assumptions about the context in which they are used. For example, the description of a component representing a hardware bus consisting of a single service cannot contain a list of the services of other components that are aggregated by its service simply because the set of compo-

nents to which the bus is connected is not known. When defining a component one usually knows something about the types of components that it may be immediately connected to. For example, the hardware bus component will always be connected to components implementing a compatible slave and/or master interface. In the Service Relation Model, service aggregation relations between services of different components are established in-directly through interfaces and service exchange relations. Service exchange relations defines relations between two classes of components (e.g a specific bus and its compatible slaves) and interfaces acts as proxies when defining the service flow.

In order to more clearly understand service aggregation and, more importantly, what needs to be true in order for an import or export relation to be justified we first need a way to more formally describe the phenomenon represented by services and service exchange relations. In the following, a characterization of services and service exchange relation based on the concept of functions will be presented. This characterization can be used to more formally define what is service aggregation is – and what it is not.

2.2.1 Service Characterization

Most people have an intuitive idea about what a service is and will agree that a service is "some work" that an individual does on behalf-of another individual. Here, an individual can be anything ranging from a person to a conceptual component. A service in the Service Relation Model represents the potential for service rather than the use of it. When we say that "a provides b with service c" it means that the service c is at the disposal of the component a. A given component can provide the same service to several other components and it may provide different services to different groups of other components. Knowing what services a given component provides to other components is the same as knowing the capabilities of the component. In other words, a service can be considered a capability – a possibility of performing some function.

The Service Relation Model is exclusively concerned with services that can, and must be, actively invoked by a service consumer in order for them to carry out their function. This is as opposed to services that are randomly or continuously being provided to the consumer without the consumers explicit consent. A service can be described as a function taking n-inputs and returning m-outputs:

$$\langle o_0, \ldots, o_m \rangle = ServiceName(i_0, \ldots, i_n)$$

Intuitively, such a service can be invoked by a service consumer by providing a proper valuation of its inputs. The invoked service may produce zero or more

outputs that are passed back to the service consumer. In keeping with the terminology of functions, the set of possible inputs to a service is called the *domain* of the service and the set of possible outputs the *range* of the service. The domain $D(s)$ of a service s is defined as the Cartesian product of the domains of the inputs:

$$D(s) = I_0 \times \ldots \times I_n$$

and, similarly, the range $R(s)$ as the Cartesian product of the domains of the outputs:

$$R(s) = O_0 \times \ldots \times O_m$$

The domain of a service is allowed to contain elements that does not represent a proper service invocation (i.e. invalid valuations of its inputs) as well as multiple elements that will result in the invocation of the same functionality. This provides an additional degree of freedom in choosing the domain used to describe a service which helps to make the description more intuitive.

Note that a service is not a proper mathematical function because it represents a phenomenon that is allowed to have side effects and whose outputs may depend on the state of the system in addition to its inputs. Also, in the Service Relation Model, services are abstractions meaning that we do not concern ourselves with the definition of the behavior or function implemented by the service.

Example 2.1 (Memory Access Services) Services representing access to memory are quite common. Such services can be represented using four inputs and one output:

$$\langle val \rangle = MemoryAccessService(op, w, addr, val)$$

where $op \in \{\texttt{rd}, \texttt{wr}\}$ specifies the type of operation (read or write), $w \in \{1, 2, 4\}$ specifies the width of the access (1, 2 or 4 bytes), $addr \in \mathbb{N}_0$ specifies the address to be accessed and $val \in \mathbb{Z}$ the value to be written in case of $op = \texttt{wr}$. The `MemoryService` and `BusService` from the previous examples are examples of such services. ■

Example 2.2 (Functions) The service representation of C functions is straightforward: one output representing the function return value and a single input for each function argument:

$$\langle result \rangle = FunctionName(arg_0, arg_1, \ldots, arg_n)$$

For example, the power function of the C math library can be modeled as $\langle result \rangle = pow(a, b)$ where $result, a, b \in \mathbb{R}$. ■

Example 2.3 (Execution) Another interesting, and often used, class of services is the monolithic `Execution` service of processors. The inputs and outputs of this service is highly dependent upon the actual processor and, in general, not trivial to describe. Fortunately, we can get around this by assuming that the services of a processor are accessed through a high-level language such as C.

For a given processor only a selection of instructions will be used to access services provided by the connected hardware. Common examples of such instructions are the LOAD and STORE instructions of a RISC machine. Assuming that a high-level function wrapper (e.g. `void load(int* addr)`) is available for each of these instructions, the execution service can be described as:

$$\langle result \rangle = Execution(name, arg_0, arg_1, \ldots, arg_{max})$$

where *result* is the return value, *name* is the name of the wrapper function, arg_x is an argument and *max* is the maximum number of arguments any function can take. Notice that many of the combinations of inputs for this function does not correspond to a proper invocation of the execution service. For example, the valuation $load, void, 10, 10, \ldots void$ does not make sense. ■

2.2.2 Service Exchange Relation Characterization

Like services, the phenomenon represented by service exchange relations may also be described using functions. The purpose of such a function is to define the possible service requests (inputs) and results (outputs) that may pass through a particular service exchange relation. In comparison with the functions describing services, the functions describing service exchange relations does not represent any ”work” to be done at run-time. As was the case for services, the Cartesian product of the domains of the inputs of a service exchange relation is called the *domain* of the relation and, similarly, the Cartesian product of the domains of the outputs the *range* of the relation.

Example 2.4 (Memory Interfaces) A common way of interfacing hardware components is through memory interfaces where a master component may access a slave by issuing read and write requests. The function signature describing a service exchange relation representing a memory interface can be given as:

$$\langle val \rangle = BusToBusSlave(op, w, addr, val)$$

where $op \in \{\texttt{rd}, \texttt{wr}\}$ specifies the type of operation (read or write), $w \in \{1, 2, 4\}$ specifies the width of the access (1, 2 or 4 bytes), $addr \in \mathbb{N}_0$ specifies the address

to be accessed and $val \in \mathbb{Z}$ the value to be written in case of $op = \texttt{wr}$. Notice that the function has exactly the same inputs and outputs as the function used for describing memory access services. ■

Example 2.5 (Function Invocation) The service exchange relations the `Context/Caller` and `Callee/Context` representing the exchange of service between a context and a software entity using function invocation can be described as:

$$\langle result \rangle = FunctionSxr(name, arg_0, arg_1, \ldots, arg_n)$$

where *name* is the domain of valid function names, n is the maximum number of allowed arguments a function can take and arg_x is an input representing the value of an argument. arg_x is in the domain $TYPE \cup \perp$ where $TYPE$ is the domain of values for the corresponding type and $\perp$ represents "not used". ■

In earlier versions of the Service Relation Model, service exchange relations were bi-directional – as the name "exchange relation" suggests. Both the object and subject components of a relation were allowed to export services to it. While this simplified some models slightly it also complicated the task of encoding the service flow because each interface would introduce a circular dependency in the flow graph. Eventually, the bidirectional relations were replaced with the unidirectional relations. There might be good reasons to allow bidirectional flow of service since it can simplify some models. Bidirectional flow, however, is difficult to handle since it introduces cycles in the model. An example of a problem caused by such cycles will be presented later in section 5.4.6.

2.2.3 Service/Interface Aggregation

Using the presented characterization of services and service exchange relations we may now more formally define the concept of service aggregation. As previously mentioned, the service aggregation relations between services are not explicit in the Service Relation Model. Instead, all service aggregation relations are established in-directly through interfaces and service exchange relations. A service may participate in two different relations with interfaces: import and export relations.

In the following, let $a \xrightarrow{i} b$ be an infix boolean-valued operator between a specification of a service invocation a and a specification of a service exchange relation access b or a specification of a service exchange relation access a and a specification of a service invocation b. The operator returns true if the invocation or access specified by a causes the invocation or access specified by b and

otherwise false. For example,

$$r_0 = S(i_0) \xrightarrow{i} r_1 = I(i_1)$$

is true if invoking the service S with input i_0 causes the interface I to be accessed with input i_1. Furthermore, let $x \lhd y$ be another boolean-valued operator between two sets of input/output values returning true if the information contained in x is somehow part of the information contained in y. This will, for example, be the case for the following pair of values:

$$\langle q, t \rangle \lhd \langle \texttt{SomeFunction}, q, t \rangle$$

if the meaning of the values q and t is the same in both tuples. In other words, $x \lhd y$ is true only if the value x can be extracted from y.

Definition 2.0 (Import) An interface i associated with the service exchange relation *sxr* can be *imported* into a service s only if the following is true:

$$\forall x \in D(s) : \exists y \in D(sxr) : r_0 = s(x) \xrightarrow{i} r_1 = sxr(y) \land x \lhd y \land r_1 \lhd r_0$$

meaning that it must be possible to invoke s so that i is accessed with every possible input and that any information returned by the access of i is part of the output returned by the invocation of s. □

Definition 2.1 (Export) A service s can be *exported* to an interface i associated with the service exchange relations *sxr* only if the following is true:

$$\forall x \in D(sxr) : \exists y \in D(s) : r_0 = sxr(x) \xrightarrow{i} r_1 = s(y) \land x \lhd y \land r_1 \lhd r_0$$

meaning that it must be possible to access i so that s is invoked with every possible input and that any information returned by the invocation of s is part of the output returned by the access of i. □

Collectively, the two definitions ensures that when we say a service s is available at some point x in a model it will be possible to access s from that point. It should be noted that the definition of services and service exchange relations as functions is not required by the Service Relation Model. In the Service Relation Model, both services and service exchange relations are abstract entities and, as such, are not associated with a definition of their structure or behavior. Later, in section 4.3, an approach to code generation based on the Service Relation Model is presented. This approaches is based on the definition of services and service exchange relations as functions as presented in this section.

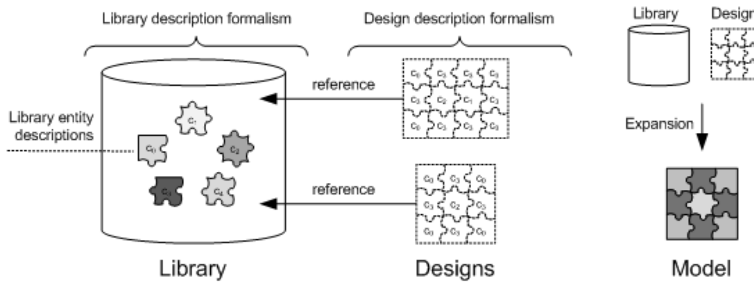


Figure 2.6 – The concepts of the Service Relation Model are split into three domains: the *library* domain, the *design* domain and the *model* domain. The relationship between library entity descriptions of the library and two (different) designs. The description of a design does not contain descriptions of its constituent components and instead references the descriptions of the library.

2.3 In-depth Presentation

The concepts of the Service Relation Model are divided into three different domains called the *library domain*, the *design domain* and the *model domain*. The model domain contains concepts that can be used to create models like those presented in the examples of section 2.1.1 and 2.1.2. In such models, however, components are not readily re-usable because of their static structure. For example, the bus component of the example in section 2.1.1 represents a bus instance with exactly one slave and one master attachment. The library and design domains contains additional concepts that acts as a front-end for specifying models and, collectively, enables re-use of components by separating component definition from component assembly. Figure 2.6 illustrates this separation. This emphasis on re-use is mandatory for the Service Relation Model to efficiently support component-based design approaches.

Finally, a note on notation. To simplify the presentation a "." notation will be used to refer to the members of tuples. For example, given the tuple $a = \langle b, c \rangle$ we will write $a.b$ when referring to the first member and $a.c$ when referring to the second. For reference, table 2.7 contains a list of symbols used in this thesis.

2.3.1 Models

It is the purpose of this section to provide a detailed description of service relation models ignoring the details of how these models come into existence. The first part of the presentation will focus solely on the structure of models. The

m, M	Models	RI, ri	Import relations
δ, Δ	Designs	RE, re	Export relations
ℓ	Library	S, s	Services
T, t	Templates	I, i	Interfaces
C, c	Components	RES, res	Resources
O, o	Options	CL, cl	Resource claims
R, r	Service exchange relations	P, p	Parameters
RS, rs	Resource shares	A, a	Assertions
G, g	Interface groups		

Figure 2.7 – Symbols and their meaning

meaning and purpose of models will be explained afterwards. Formally, a model is defined as:

Definition 2.2 (Model, m**)** A model is a pair $m = \langle C, R \rangle$ where C is the set of components in the design and R is the set of service exchange relations between the elements of C. □

Internally, a component consists of services, interfaces and import/export relations. Services are used to describe the capabilities of components. A service, in the context of the Service Relation Model, shares many similarities with the concept of functions in programming. An interface is a connection point through which the component may be connected to another component via a service exchange relation. The interfaces and services of a component are related to each other using import and export relations. Formally, a component is defined as:

Definition 2.3 (Component, c**)** A component is a quadruple $c = \langle S, I, RE, RI \rangle$ where S is a set of services, I is a set of interfaces, RE is a set of export relations and RI a set of import relations. □

Definition 2.4 (Import Relation, ri**)** An import relation is a triple $ri = \langle c, i, s \rangle$ where c is the parent component of the relation, $i \in c.I$ an interface and $s \in c.S$ a service. □

Definition 2.5 (Export Relation, re**)** An import relation is a triple $re = \langle c, s, i \rangle$ where c is the parent component of the relation, $s \in c.S$ a service and $i \in c.I$ an interface. □

The components of a model are related to each other using a set of service exchange relations. A service exchange relation is a binary relation representing

the possible flow of service from one component to another. Service exchange relations are unidirectional meaning that service can only flow in one direction. Two entities related to each other using binary relations are often referred to as the *object* and the *subject* of the relation. In the Service Relation Model, the object of the relation is per definition always the service provider and the subject the service consumer. Components are indirectly connected to service exchange relations through their interfaces:

Definition 2.6 (Interface, i) An interface is a triple $i = \langle c, rel, role\rangle$ where c is the parent component of the interface, $rel \in R$ is the relation associated with the interface and $role \in \{\text{object}, \text{subject}\}$ is the role associated with the interface. □

Definition 2.7 (Service Exchange Relation, r) A service exchange relation is a pair of pairs $r = \langle\langle c_1, i_1\rangle, \langle c_2, i_2\rangle\rangle$ where $c_1, c_2 \in C$, $i_1 \in c_1.I$ and $i_2 \in c_2.I$. The pair $\langle c_1, i_1\rangle$ identifies the object of the relation and the pair $\langle c_2, i_2\rangle$ the subject. □

An interface can at most be connected to one service exchange relation and may be unconnected. A model with one or more unconnected interfaces is called a *partial* model. A component has a finite set of interfaces and, consequently, can only participate in a finite number of service exchange relations with other components.

Well-formed Model. For a *well-formed* service relation model $m = \langle C, R\rangle$ the following must be true. Each interface is either properly connected or not connected:

$$\forall \langle\langle c_0, i_0\rangle, \langle c_1, i_1\rangle\rangle \in R : i_0.rel = i_1.rel \wedge i_0.role \neq i_1.role \tag{2.1}$$

Each interface is connected at most once:

$$\begin{gathered}\forall \langle\langle c_0, i_0\rangle, \langle c_1, i_1\rangle\rangle, \langle\langle c_2, i_2\rangle, \langle c_3, i_3\rangle\rangle \in R \times R : \\ (c_0 = c_2 \wedge i_0 = i_2) \leftrightarrow (c_1 = c_3 \wedge i_1 = i_3)\end{gathered} \tag{2.2}$$

The target interface of an export relation is always an object interface:

$$\forall c \in C : (\forall re \in c.RE : re.I.rel = \text{object}) \tag{2.3}$$

and the source of an import relation is always a subject interface:

$$\forall c \in C : (\forall ri \in c.RI : ri.I.rel = \text{subject}) \tag{2.4}$$

Collectively, 2.1, 2.3 and 2.4 ensures that service only flow one way through interfaces.

Concept	Constraint
Interface, i	*nothing*
Service, s	$\{s\} \subseteq SA[s]$
Import relation, $ri = \langle c, s, i \rangle$	$SA[s] \subseteq SA[i]$
Export relation, $re = \langle c, i, s \rangle$	$SA[i] \subseteq SA[s]$
Service exchange relation, $r = \langle \langle c_1, i_1 \rangle, \langle c_1, i_2 \rangle \rangle$	$SA[i_0] = SA[i_1]$

Table 2.1 – Service flow, concepts and implied constraints

2.3.2 Service Flow

The purpose of the Service Relation Model is to determine the sets of services available for each component in a model. More specifically, we are interested in the services available at each interface and service of each component. Collectively, this information is called service availability information:

Definition 2.8 (Service Availablity Information, SA) Let I_* be the set of interfaces and S_* be the services in a model m. The set of services available at an interface $i \in I_*$ or a service $s \in S_*$ is called the service availablity information of i or s. The service availability information of m is a map:

$$SA : I_* \cup S_* \rightarrow \mathcal{P}(S_*)$$

mapping interfaces and services to the sets of available services. □

To more formally describe the semantics of service flow in the Service Relation Model, we will show how the service availablity information of a model is given relative to a set of constraints derived from the various concepts of the Service Relation Model. Table 2.1 shows the constraints associated with the concepts. Notice that components are not mentioned as they do not directly influence the service availablity information. The set of constraints associated with each occurrence of the concepts in a model m comprises a constraint system describing the flow of service in m. The constraint system may have several feasible solutions. One of these solutions, called the *least solution*, exactly describes the service availablity information of m. Informally, the least solution to a constraint system can be thought of as the ”smallest” solution in terms of the number of services available. For a more exact definition of the least solution see [68]. An algorithm for computing the service availablity information of a model will be presented later in section 2.4.

Example 2.6 For the hardware platform of figure 2.2, we can extract the following constraints using table 2.1:

$$
\begin{aligned}
\{\,\texttt{memory.MemoryService}\,\} &\subseteq \mathrm{SA}[\texttt{memory.MemoryService}] \\
\mathrm{SA}[\texttt{memory.MemoryService}] &\subseteq \mathrm{SA}[\texttt{memory.bus}] \\
\mathrm{SA}[\texttt{bus.slave}_0] &\subseteq \mathrm{SA}[\texttt{bus.BusService}] \\
\{\,\texttt{bus.BusService}\,\} &\subseteq \mathrm{SA}[\texttt{bus.BusService}] \\
\mathrm{SA}[\texttt{bus.MemoryService}] &\subseteq \mathrm{SA}[\texttt{bus.Master}_0] \\
\mathrm{SA}[\texttt{processor.DataBus}] &\subseteq \mathrm{SA}[\texttt{processor.Execution}] \\
\mathrm{SA}[\texttt{processor.CoProc}_0] &\subseteq \mathrm{SA}[\texttt{processor.Execution}] \\
\{\,\texttt{processor.Execution}\} &\subseteq \mathrm{SA}[\texttt{processor.Execution}] \\
\{\,\texttt{FPU.FpAdd}\,\} &\subseteq \mathrm{SA}[\texttt{FPU.FpAdd}] \\
\{\,\texttt{FPU.FpDiv}\,\} &\subseteq \mathrm{SA}[\texttt{FPU.FpDiv}] \\
\{\,\texttt{FPU.FpMul}\,\} &\subseteq \mathrm{SA}[\texttt{FPU.FpMul}] \\
\mathrm{SA}[\texttt{FPU.FpAdd}] &\subseteq \mathrm{SA}[\texttt{FPU.Processor}] \\
\mathrm{SA}[\texttt{FPU.FpDiv}] &\subseteq \mathrm{SA}[\texttt{FPU.Processor}] \\
\mathrm{SA}[\texttt{FPU.FpMul}] &\subseteq \mathrm{SA}[\texttt{FPU.Processor}] \\
\mathrm{SA}[\texttt{bus.Master}_0] &= \mathrm{SA}[\texttt{processor.DataBus}] \\
\mathrm{SA}[\texttt{FPU.Processor}] &= \mathrm{SA}[\texttt{processor.CoProc}_0] \\
\mathrm{SA}[\texttt{memory.bus}] &= \mathrm{SA}[\texttt{memory.bus}]
\end{aligned}
$$

The least solution to this constraint system – the service availablity information of the model – was presented earlier as part of the example in section 2.1.1. ■

Another kind of interesting information, which may be derived from the service availability information, is called available at information and records for each service in a model the set of interfaces and services where the service is available:

Definition 2.9 (Available At Information, $\overline{AA}$) Let I_* be the set of interfaces and S_* be the services in a model m. The set of services and interfaces $x \in I_* \cup S_*$ where a service $s \in S_*$ is available is called the available at information of s. The available at information of m is a map:

$$\overline{AA} : S_* \to \mathcal{P}(I_* \cup S_*)$$

mapping services to sets of interfaces and services. □

In some rare cases, we are also interested in the accessibility of interfaces. Intuitively, an interface is accessible from an other interface or service if it is

possible request a service through it. This has uses when dealing with partial models because if we know that an unconnected interface i is available at some point x then we may infer that any services later made available at i are also available at x.

Definition 2.10 (Service/Interface Availablity Information, *SIA***)** Let I_* be the set of interfaces and S_* be the services in a model m. The set of services available at and the set of interfaces accessible from an interface $i \in I_*$ or a service $s \in S_*$ is called the service/interface availablity information of i or s. The service/interface availability information of m is a map:

$$SIA : I_* \cup S_* \to \mathcal{P}(S_* \cup I_*)$$

mapping interfaces and services to the sets of services and interfaces. □

Note that the service/interface availablity information of a model contains the service availablity information. The service/interface availablity information of models will not be used much in this work but has been included primarily because the analysis back-end used for experimentation purposes computes this instead of the less detailed service availablity information.

2.3.3 Configurability

The examples presented thus far have all focused on models. The components of a model are concrete instances that does not provide much flexibility. For example, the bus component of the example in figure 2.2 represents a concrete bus with exactly one master attachment and one slave attachment. While it is entirely possible to formulate models directly on the basis of some input, say a platform specification, it is often more convenient to employ a scheme where models are generated on the basic of re-usable component descriptions (templates). In this section, configurability in the Service Relation Model will be presented.

As was mentioned earlier, a specification of a model consists of a design description and a library. A library is a repository for re-usable information such as component templates and types of service exchange relations. A design description specifies a model by specifying an allocation, a configuration and a topology of components relative to a library.

The front-end of the Service Relation Model consists of two different (but related) formalisms. The first of these formalisms, called the *library description*

formalism, is used for describing re-usable entities such as components and the possible relations between the them. These descriptions, called *library entity descriptions*, are conceptually grouped together in a *library*. Formally, a library is defined as:

Definition 2.11 (Library, ℓ) A library is a pair $\ell = \left\langle T^{\ell}, R^{\ell} \right\rangle$ where T^{ℓ} is a set of component templates and R^{ℓ} is a set of service exchange relations. □

Notice that the term *component template* is used to denote a re-usable description of a component in the library domain.

The second formalism, called the *design description formalism*, is used for describing a concrete allocation, configuration and topology of a system based on the library entity descriptions of a given library. A system described using the design description formalism is referred to simply as a *design*:

Definition 2.12 (Design, δ) A design δ is an triple $\delta = \left\langle \ell, C^{\delta}, R^{\delta} \right\rangle$ where ℓ is the library containing the library entities referenced by the design, C^{δ} is a set of component instances and R^{δ} a set of service exchange relation instances . □

Through a process called *design expansion* the description of a design and a library containing the necessary library entities are merged into a ”standalone” service relation model of the design. Informally, a model can be thought of as a design where the references to library entities have been replaced by their content (i.e. descriptions in the library domain).

Since several of the similarly named concepts in the Service Relation Model have slightly different meaning depending on whether the context is the library domain, design domain or the model domain we will use a special notation to set them apart: Entities in the library domain will be decorated with the superscript ℓ, entities in the design domain will be decorated with the superscript δ and entities in the model domain context will not be decorated. Thus, s^{ℓ} denotes a service in the library domain and c^{δ} a component of the design domain where as s and c denotes entities of the model domain.

2.3.3.1 Textual Representation, The `xSRM` Language

All of the example of models presented thus far have been given using diagrams using the graphical notation introduced in section 2.1.1. While the graphical

```
model SimplePlatform {
  component memory {
    interface bus : obj(BusSlaveToBus);
    service MemoryService { export(bus); }
  }
  component bus {
    interface slave0 : sub(BusSlaveToBus);
    interface master0 : obj(BusToBusMaster);
    service BusService { import(slave0); export(master0); }
  }
  component processor {
    interface DataBus : sub(BusToBusMaster);
    interface CoProc0 : sub(CoProcToProc);
    service Computation { import(DataBus, CoProc0); }
  }
  component FPU {
    interface Processor : obj(CoProcToProc);
    service FpAdd  { export(Processor); }
    service FpDiv  { export(Processor); }
    service FpMult { export(Processor); }
  }

  connect(memory.bus, bus.slave0);
  connect(bus.master0, processor.DataBus);
  connect(processor.CoProc0, FPU.Processor);
}
```

Listing 2.1 – `sSRM` Descriptions of the service relation model of the platform from the example of section 2.1.1.

notation is excellent for depicting models it is less well suited for depicting entities in the library and design domains. The reason for this being that many of the concepts that has yet to be presented does not have an intuitive graphical representation. This section contains a brief presentation of a textual representation, named the *Simple Service Relation Model* `sSRM` language, that will be used in the remainder of this thesis.

The `sSRM` language is associated with three different views – one for each of the three domains of the Service Relation Model. The model view corresponds to the graphical notation and are used to describe models. The library view is used to describe the entities of the library domains such as service exchange relation and component template definitions. Finally, the design view is used to specify a model relative to a library. Hopefully, the semantics of the `sSRM` language should be intuitive enough to excuse the fact that a formal presentation of the language is not provided.

Listing 2.1 shows an `sSRM` description of the service relation model of the hardware platform of example 2.1.1. The description is enclosed in a `model` tag showing that it contains a description of a model. Listing 2.2 shows the `sSRM` descriptions of the library and design specifying the model of the platform.

```
library {
  namespace platform {
    sxr BusSlaveToBus;
    sxr BusToBusMaster;
    sxr CoProcToProc;

    template memory {
      interface bus : obj(BusSlaveToBus);
      service MemoryService { export(bus); }
    }
    template bus {
      interface slave0  : sub(BusSlaveToBus);
      interface master0 : obj(BusToBusMaster);
      service BusService { import(slave0); export(master0); }
    }
    template processor {
      interface DataBus : sub(BusToBusMaster);
      interface CoProc0 : sub(CoProcToProc);
      service Computation { import(DataBus, CoProc0); }
    }
    template FPU {
      interface Processor : obj(CoProcToProc);
      service FpAdd  { export(Processor); }
      service FpDiv  { export(Processor); }
      service FpMult { export(Processor); }
    }
  }
}

design SimplePlatform {
  component processor : platform.processor {}
  component memory    : platform.memory {}
  component FPU       : platform.FPU {}
  component bus       : platform.bus {}

  connect(memory.bus, bus.slave0);
  connect(bus.master0, processor.DataBus);
  connect(processor.CoProc0, FPU.Processor);
}
```

Listing 2.2 – sSRM Descriptions of the library and design used for specifying the model of listing 2.1.

The word *"simple"* in the name "Simple Service Relation Model" is due to the fact that it is a syntactically simpler version of another XML-based language called the `xSRM` language supported by the `xSRM` framework that will be presented later in section 3.4.

2.3.3.2 Options

A component template is configurable by means of *options*. An option is a partial description of service flow that can optionally be included in the service flow description of a component instance depending on the needs of that particular instance. An option can be included zero or more times in the same component instance. In the case that an option is included several times then its service flow description is duplicated accordingly.

Example 2.7 Listing 2.3 shows a `sSRM` description of a hardware bus illustrating the use of options. Here, the number of master and slave attachments have been made customizable by means of two options called `master` and `slave`. The master option adds an interface for connecting the bus component to a master attachment and, similarly, the slave option adds an interface for connecting it to a slave attachment. The description also shows a design containing an instance of the bus with one master and two slave options included and the resulting model. ■

The concept of an option exists in both the library and the design domain. In the library domain, an option is defined as:

Definition 2.13 (Option, o^ℓ) An option is a quintuple $o = \left\langle t^\ell, S^\ell, I^\ell, RE^\ell, RI^\ell \right\rangle$ where t^ℓ is the template to which the option belongs, S^ℓ is a set of services, I^ℓ a set of interfaces, RE^ℓ a set of export relations and RI^ℓ a set of import relations. □

In the design domain, an option refers to an inclusion or instantiation of an option from the library domain in some component. An option o^δ in the design domain is defined as:

Definition 2.14 (Option, o^δ) An option o^δ is a pair $o^\delta = \left\langle c^\delta, o^\ell \right\rangle$ where c^δ is the parent component of the option (i.e. the component in which the option is included) and o^ℓ is the re-usable service flow description in the library. □

```
library {
  namespace platform {
    ...
    template bus {
      option master { interface master : obj(BusToBusMaster); }
      option slave  { interface slave  : sub(BusSlaveToBus);  }
      service BusService { import(slave.slave); export(master.master); }
    }
  }
}

design BusExample {
  component MyBus : platform.bus {
    include master0 : master;
    include slave0  : slave;
    include slave1  : slave;
  }
}

model BusExample {
  component MyBus  {
    interface master.master0 : obj(BusToBusMaster);
    interface slave.slave0   : sub(BusSlaveToBus);
    interface slave.slave1   : sub(BusSlaveToBus);
    service BusService {
      import(slave.slave0, slave.slave1);
      export(master.master0);
    }
  }
}
```

Listing 2.3 – An sSRM description of a hardware bus using options.

For obvious reasons, only options where $o^{\delta}.o^{\ell} \in o^{\delta}.c^{\delta}.t.O^{\ell}$ are valid.

The service flow description of a component template consists of a mandatory and an optional part. The mandatory part contains the part of the service flow description that must be part of all instances of a given template. Conceptually, this mandatory description is not an option itself but in order to ease the formalization of the concepts we will adopt the view that the partial description of service flow shared by all instances is an option with the restriction that it must be included exactly once in each component instance. This leads to the following definition of a template in the library domain and a component in the design domain:

Definition 2.15 (Template, t^{ℓ}) A component template is a pair $t^{\ell} = \left\langle O^{\ell}, o^{\ell}_m \right\rangle$ where O^{ℓ} a set of options and $o^{\ell}_m \in O^{\ell}$ is the mandatory option. □

Definition 2.16 (Component, c^δ) A component c^δ is a triple $c^\delta = \left\langle t^\ell, O^\delta, o^\delta_m \right\rangle$ where t^ℓ is the template upon which the component is based, O^δ is a set of included options and $o^\delta_m \in O^\delta$ is the mandatory option (as dictated by the associated template of c). We require that the mandatory option is included exactly once:

$$\exists o^\delta \in O^\delta : o^\delta.o^\ell = o^\delta_m.o^\ell$$

$$\forall o^\delta_1, o^\delta_2 \in O^\delta : o^\delta_1.o^\ell = o^\delta_m.o^\ell \wedge o^\delta_1.o^\ell = o^\delta_m.o^\ell \Rightarrow o^\delta_1 = o^\delta_2$$

This implies that O^δ is never the empty set. □

It should be noted that the `sSRM` and `xSRM` languages does not incorporate the concept of the mandatory option directly. In these languages, the description of service flow otherwise contained in a mandatory option is part of the component itself.

Libraries. Besides the concepts already mentioned, the following concepts also belongs to the library domain:

Definition 2.17 (Interface, i^ℓ, I^ℓ) An interface is a pair $i^\ell = \left\langle o^\ell, rel, role \right\rangle$ where o^ℓ is the parent option to which the interface belongs, rel is the service exchange relation associated with the interface and $role \in \{\text{object}, \text{subject}\}$ is the role associated with the interface. □

Definition 2.18 (Import Relation, ri^ℓ, RI^ℓ) An import relation is a triple $ri^\ell = \left\langle o^\ell, i^\ell, s^\ell \right\rangle$ where o^ℓ is the parent option of the relation, $i^\ell \in o^\ell.I^\ell$ an interface and $s^\ell \in o^\ell.S^\ell$ a service. □

Definition 2.19 (Export Relation, re^ℓ, RE^ℓ) An export relation is a triple $re^\ell = \left\langle o^\ell, s^\ell, i^\ell \right\rangle$ where o^ℓ is the parent option of the relation, $s^\ell \in o^\ell.S^\ell$ a service and $i^\ell \in o^\ell.I^\ell$ an interface. □

As can be seen, the definition of templates and interfaces in the library domain resembles those of the model domain. The entities of the library domain, however, represent classes whereas the entities of the model domain represents instances of such classes.

Well-formed. The service flow of all possible components instantiated on the basis of a given template must be well-formed meaning that the source interface of an import relation is always associated with the subject role of a service exchange relation and, similarly, that the target interface of an export relation is always associated with the object role of a service exchange relation. Recall that in a service exchange relation, service flows from the object to the subject. Let RE^ℓ_* be the set of all export relations and RI^ℓ_* the set of all import relations in a template. The following must hold for the template to be well-formed:

$$\forall re^\ell \in RE^\ell_* : re^\ell.i^\ell.rel = \text{object} \tag{2.5}$$

and

$$\forall ri^\ell \in RI^\ell_* : ri^\ell.i^\ell.rel = \text{subject} \tag{2.6}$$

Designs. Besides the concepts already introduced, the design domain also consists of service exchange relations:

Definition 2.20 (Service Exchange Relation, r^δ, R^δ) A service exchange relation $r^\delta \in R^\delta$ is a pair of triples $r = \left\langle \left\langle c^\delta_0, o^\delta_0, i^\ell_0 \right\rangle, \left\langle c^\delta_1, o^\delta_1, i^\ell_1 \right\rangle \right\rangle$ where $c_0, c_1 \in C^\delta$, $o^\delta_0 \in c^\delta_0.O^\delta$, $o^\delta_1 \in c^\delta_1.O^\delta$, $i_0 \in o^\delta_0.o^\ell.I^\ell$ and $i^\ell_1 \in o^\delta_1.O^\ell.I^\ell$ for some design $\delta = \left\langle \ell, C^\delta, R^\delta \right\rangle$ □

Well-formed. We assume that the relations of a design are all properly connected meaning that only interfaces associated with the same service exchange relation and with opposite roles can be connected. More formally:

$$\forall((c^\delta_0, o^\delta_0, i^\ell_0), (c^\delta_1, o^\delta_1, i^\ell_1)) \in R^\delta : i^\ell_0.rel = i^\ell_1.rel \land i^\ell_0.role \neq i^\ell_1.role \tag{2.7}$$

Furthermore, each interface is connected at most once:

$$\begin{array}{c} \forall \left\langle \left\langle c^\delta_0, o^\delta_0, i^\ell_0 \right\rangle, \left\langle c^\delta_1, o^\delta_1, i^\ell_1 \right\rangle \right\rangle, \left\langle \left\langle c^\delta_2, o^\delta_2, i^\ell_2 \right\rangle, \left\langle c^\delta_3, o^\delta_3, i^\ell_3 \right\rangle \right\rangle \in R^\delta \times R^\delta : \\ (c^\delta_0 = c^\delta_2 \land o^\delta_0 = o^\delta_2 \land i^\ell_0 = i^\ell_2) \leftrightarrow (c^\delta_1 = c^\delta_3 \land o^\delta_1 = o^\delta_3 \land i^\ell_1 = i^\ell_3) \end{array} \tag{2.8}$$

As can be seen, not all interface has to be connected. A design with one or more unconnected interfaces is called a *partial design*.

2.3.4 Design Expansion

Having presented the various concepts of the Service Relation Model in the three different domains we can now more formally define the design expansion process. Design expansion is formalized as a function $E(\delta) = m$ that takes a design and returns the corresponding model. The definition of E, as presented here, uses several auxiliary functions.

The first step of the formalization consists of mapping services in the library domain to services in the model domain. Because of options and the fact that multiple components may be instantiated on the basis of the same template there may be zero of more services in the model domain for each service in the library domain. To describe this relation we use an injective function:

$$S_{\delta \to m} : S^{\ell} \times C^{\delta} \times O^{\delta} \to S \tag{2.9}$$

The function formalizes the instantiation of a service which is part of an option in the library. The function requires three arguments: a component d^{δ} and an option $o^{\delta} \in d^{\delta}.O^{\delta}$ in the design domain and a service $s^{\ell} \in o^{\delta}.o^{\ell}.S^{\ell}$ in the library domain. The function returns the service in the model domain that corresponds to the instantiation of s^{ℓ} belonging to the option o^{δ} of component c^{δ}. The function $S_{\delta \to m}$ is injective:

$$S_{\delta \to m}(s_1^{\ell}, c_1^{\delta}, o_1^{\delta}) = S_{\delta \to m}(s_2^{\ell}, c_2^{\delta}, o_2^{\delta}) \Rightarrow s_1 = s_2 \wedge c_1 = c_2 \wedge o_1 = o_2 \tag{2.10}$$

This is an important property of the function since it captures the fact that a service in the model domain represents an instance of a library service or interface of a particular included option in a particular component instance.

We also have a similar injective function for mapping interfaces in the library domain to interfaces in the model domain:

$$I_{\delta \to m} : I^{\ell} \times C^{\delta} \times O^{\delta} \to I \tag{2.11}$$

Using these two functions the mapping of a component c^{δ} in the design domain into a component c in the model domain can now be described using another function:

$$E_c(c^{\delta}) = \langle S, I, RE, RI \rangle \tag{2.12}$$

where

$$
\begin{aligned}
S &= \{S_{\delta\to m}(s^\ell, c^\delta, p(s)) | s^\ell \in c^\delta.t.S^\ell\} \\
I &= \{I_{\delta\to m}(i^\ell, c^\delta, p(i)) | i^\ell \in c^\delta.t.I^\ell\} \\
RE &= \{\left\langle S_{\delta\to m}(s^\ell, c^\delta, o_1^\delta), I_{\delta\to m}(i^\ell, c^\delta, o_2^\delta)\right\rangle | \\
&\qquad \left\langle s^\ell, i^\ell\right\rangle \in c^\delta.t.RE^\ell \wedge o_1, o_2 \in c.O^\delta \wedge o_1 = p(s) \wedge o_2 = p(i)\} \\
RI &= \{\left\langle S_{\delta\to m}(s^\ell, c^\delta, o_1^\delta), S_{\delta\to m}(s^\ell, c^\delta, o_2^\delta)\right\rangle | \\
&\qquad \left\langle i^\ell, s^\ell\right\rangle \in c^\delta.t.RI^\ell \wedge o_1, o_2 \in c.O^\delta \wedge o_1 = p(i) \wedge o_2 = p(s)\}
\end{aligned}
$$

The mapping of service exchange relations in the design domain to service exchange relations in the model domain is straightforward:

$$
E_r((c_0^\delta, o_0^\delta, i_0^\ell), (c_1^\delta, o_1^\delta, i_1^\ell), C) = \left\langle (E_c(c_0^\delta), I_{\delta\to m}(c_0^\delta, o_0^\delta, i_0^\ell)), (E_c(c_0^\delta), I_{\delta\to m}(c_1^\delta, o_1^\delta, i_1^\ell)))\right\rangle
$$

The function describing the expansion of a design δ can now be given as:

$$
E(\delta) = \left\langle \{E_c(c^\delta) | c^\delta \in \delta.C^\delta\}, \{E_r(r^\delta) | r^\delta \in \delta.R^\delta\}\right\rangle
$$

Example 2.8 Figure 2.8 shows a library template (**a**) and three components based on it (**b**, **c**, **d**). The template provides two options (o_0, o_1) in addition to the mandatory option. The lines in the figure represents import/export relations. Component instance **b** does not include any options so the service flow of the component matches that of the mandatory option excluding the relations it has with the two options. In component instance **c** a single instance of the option o_1 is included. The third component instance, **d**, includes two instances of both options o_0 and o_1. Notice that the relations between the options in the template have been duplicated in the instantiated component according to the number of instances included.

■

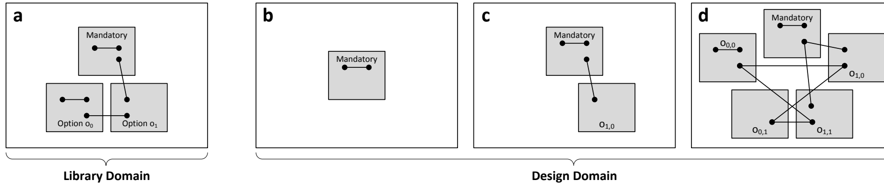


Figure 2.8 – Import/export relations and design expansion.

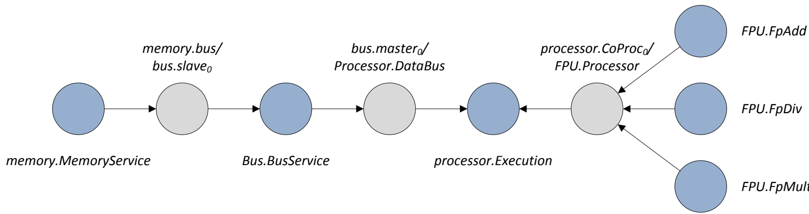


Figure 2.9 – Service flow graph (SFG) of the model of the simple hardware platform in figure 2.2

2.4 Analysis

In this section, a pair of algorithms for computing the service availablity information and the service/interface availablity information of a model are presented. Because the algorithms are almost identical emphasis will be on the algorithm for computing the service availablity information. The difference between the two algorithms will be briefly discussed at the end of the section.

Both algorithms operates on a more explicit representation of the service flow in a model called the *service flow graph* (SFG). Figure 2.9 shows the service flow graph of the model depicted in Figure 2.2. Formally, a service flow graph is defined as:

Definition 2.21 (Service Flow Graph) A service flow graph is a quadruple $sfg = \langle S, I, N, E \rangle$ where S is a set of service, I is a set of interface, $N = S \cup I$ is a set of nodes and $E \subset N \times N$ is a set of edges representing service flow. □

A service flow graph is a cyclic graph where the nodes represents the services and interfaces in the model and the edges represent import/export relations. Pairs of interfaces connected by means of a service exchange relation are collapsed and represented by means of a single node. Extracting the service flow graph from a model is fairly straightforward and will not be discussed here.

Algorithm 1 Available Services - Worklist Algorithm

```
 1: procedure SOLVE(S, I, E)
 2:     worklist := E                                   ▹ Initialization
 3:     for each i ∈ I do
 4:         available[i] := ∅
 5:     end for
 6:     for each s ∈ S do
 7:         available[s] := { s }
 8:     end for
 9:     while worklist ≠ ∅ do                           ▹ Main loop
10:         e := DEQUEUE(worklist)
11:         t := GETTARGETNODE(e)
12:         s := GETSOURCENODE(e)
13:         if ¬ (available[t] ⊆ available[s]) then
14:             available[t] := available[t] ∪ available[s]
15:             for each o ∈ GETSUCCESSOREDGES(t) do
16:                 ENQUEUE(o)
17:             end for
18:         end if
19:     end while
20: end procedure
```

Algorithm 1 is used to compute the service availability information of a model as presented previously in section 2.3.2. The algorithm is based on a generic worklist algorithm. More specifically, algorithm 1 is an adaptation of an algorithm presented in [68] used for computing data flow information in the context of program analysis.

In algorithm 1, every node in the service flow graph is associated with a set of services that are considered available at that particular node – an associative array, named *available,* is used to represent these associations. For the nodes representing interfaces these sets are initialized to the empty set and for the nodes representing services the sets are initialized to contain the service represented by the node (i.e. we adopt the idea that a service is available at itself). A queue, called the worklist, is initialized to contain all the edges of the service flow graph. The main loop of the algorithm extracts one edge at a time and tests if the set of services available at the source node is a subset of the set of services available at the target node. If this is not the case then the services available at the source node is added to the set of services available at the target node and any edges having the target node as source node is added to the worklist.

The complexity of algorithm 1 is bounded by the number of required itera-

tions of the while loop. To determine the maximum number of iterations, we note that if the test at line 13 succeeds then at least one new service will be added to the availablity set of the target of the edge being considered. For a given target node n, the update of the availablity set at line 14 can at most be executed h times where h is the number of services in S. Let M be the number of successor edges associated with the node with the most successor edges in the service flow graph, each entry into the if-statement of line 13 will cause at most M new edges to be inserted into the worklist. Each node will cause a maximum of $M \times h$ edges to be added to the worklist. This yields a complexity of $O(N \times M \times h)$ where N is the number of nodes in the service flow graph. Since $h \leq N$ and $M \leq N$ we can simplify this to $O(N^3)$. Algorithm 1 uses a worklist of edges. Another version of the algorithm uses nodes instead of edges. The complexity of such an algorithm is also known to be $O(N^3)$, [68].

Algorithm 1 cannot be used to compute the service/interface availablity information of a model because it only considers services. Algorithm 2 is an adaption of Algorithm 1 that also computes the availablity of the interfaces in the service flow graph. Pseudo code for Algorithm 2 can be found in appendix A. Algorithm 2 differs from Algorithm 1 in the size of the sets and in its initialization of the associative array *available* and, consequently, the two algorithms have the same complexity. When considering both the availablity of services and accessibility of interfaces, the problem amounts to computing the transitive closure of its service flow graph. Much research has been done in efficient algorithms for computing the transitive closure of graphs and algorithms exists that can handle very large problems quite fast [73].

2.5 Discussion & Summary

In this chapter, the basic concepts of the Service Relation Model have been presented. Readers familiar with static program analysis will notice many similarities between, on one hand, the concepts of the Service Relation Model and the service availablitity analysis and, on the other, data flow analysis. Both the Service Relation Model, the analysis method and parts of the presented formalization have been inspired by data flow analysis.

A key feature of the Service Relation Model, that has not been explicitly mentioned, is that it does not impose any constraints on what a component is. In many component models, components are given meaning by organizing them in a class hierarchy similar to the service class hierarchy presented previously. For example, many models restricts themselves to components that can be classified as being processors, interconnects or memories.

The concepts of the Service Relation Model presented in this chapter should be considered a ”minimal” set. The modeling capabilities of the model could be broadened by adding additional concepts. An obvious extension would be to incorporate the notion of hierarchy into the Service Relation Model. A hierarchy concept would allow the designer to create new components using existent components. As was mentioned in the introduction, hierarchy can be used to keep complexity in check. Another interesting extension, related to hierarchy, is embedding relations. Some service exchange relations models a form of embedding where one of two connected component can be said to be embedded within the other. The most notable example of this is the Processor/SWE service exchange relation between processors and software entities. From a modeling point of view, it may be more intuitive to represent such relations by means of embedding (e.g. boxes inside other boxes).

The Service Relation Model can only be used to model designs with a static topology. This means that it cannot be used to analyze designs that employs runtime re-configuration. It may be possible to extend the concepts of the Service Relation Model to also handle dynamic topologies. This, however, has not been explored much in this project and is left for future work.

Another potential shortcoming of the model had to do with the justification of service/interface aggregation. An import relation between an interfaces and a service states that *all* services available at the interfaces are also available at the service. In some cases, however, only a subset of the services available at an interface can be aggregated by a service. The platform of figure 2.10 illustrates this. The platform consists of two processors, two buses, two memories, a peripheral and a bus-to-bus bridge. The bridge is used to mount part of the address space of slave bus (bus_1) to the address space of the master bus (bus_0). A possible organization of the address spaces of the two buses is shown in figure 2.10. Part of the address space of bus_0 is used for mounting part of the address space of bus_1. Notice that the address range of the peripheral in the address space of bus_1 is only partially mapped through the bridge to bus_0 meaning that some addresses of the peripheral cannot be accessed from bus_0.

In these cases, an import relation cannot be justified as it may cause the service availablity analysis information to be incorrect because services accessible through the low address range of $peripheral_1$ will be considered available at the processor even though these services cannot be accessed through the bridge. In order to avoid invalidating the analysis information, the import relation must necessarily be removed. This, however, means that the analysis becomes inaccurate albeit not invalid.

The obvious solution to this problem is to allow for import relations to model aggregation of some services rather than all. One way to accomplish is to

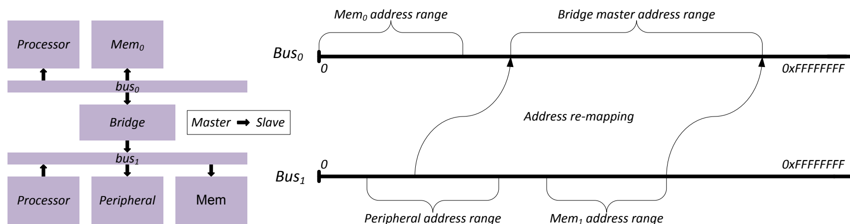


Figure 2.10 – Simple platform using a bus-to-bus bridge to map part of the address space of one bus onto the address space of the other.

associate all instances of the three relation types (import, export and service exchange relations) with a *service transformation function* that can be used to moderate which services are allowed to flow through the relation. In order for a transformation function to make a decision about whether or not a particular service should be allowed to flow through it, it must know something about the service. Exactly what kind of information is required to make this decision depends on what the relation represents.

CHAPTER 3

Consistency Checking

In a component-based design approach, a platform is constructed by selecting, configuring and connecting components from a library of existing components. A platform can consist of both hardware (e.g. processors, memories, interconnects and peripherals) and software (e.g. operating systems, middleware) components. The resulting high-level description of the platform can be fed to tools used for estimating its performance in regards to an application and/or to synthesis tools to obtain an implementation at a lower level of abstraction. Both performance estimation and synthesis can be quite slow and must be restarted if an error is detected in the design. In order for the designer to remain productive, it is imperative that errors are detected as early as possibly – preferably at design time. A common way to accomplish this is by means of consistency checking where one or more properties of the design are statically tested on the basis of the high-level description.

In this chapter, we introduce a set of additional concepts to the basic Service Relation Model presented in the previous chapter. Together these concepts can be used for checking the consistency of a platform or a system with respect to service and resource availablity.

3.1 Service Classes and Hierarchies

Using the analysis procedure presented in the previous chapter, we are able to determine the sets of services available at different points of interest in a design. In order for this information to be useful, we need a way to attach meaning to these services. This is achieved by means of the *service class* concept presented in this section.

A service class is an abstract representation of a set of services that can be considered to be similar in some way. If we know that a given service belongs to a given service class we may infer something about the service. For example, if we know that a service s is a member of the service class `SCMemoryService` we may infer that s can be used to access memory because we assume this to be the case for all members of the `SCMemoryService` service class. When we say that a service class is an abstract representation of a set of services it means that the class exists independently of any members it may have in some model. In the Service Relation Model, service classes are shared between the library and model domains but it is only makes sense to talk about the individual members of a service class in the context of a specific model.

The concept of service classes provide the means for analysis tools to reason about the services of a model without the specific (implementation) details of each service. This allows for the construction of tools with the property of being independent of the component templates available at any given time and thus designs upon which it operates. An example of such a tool will be presented later in chapter 4. Service classes can also be used by components to reference services provided by other components without having to explicitly name them. If a component c_0 includes a reference to a specific service of another component c_1 then c_0 will be come tightly coupled with c_1. This would be unfortunate because it would never be possible to replace c_1 with another, possibly more efficient, component c_2 even though it was functionally equivalent to c_1. Also, since components in the Service Relation Model are based on templates defined independently of any actual models it is not possible to reference a service of another component and, consequently, the indirection provided by the service class concept is needed to enable inter-component referencing of services.

It is important to note that the meaning of a service class is not captured by the Service Relation Model. Instead, information about what a given service class represents must be shared between component and tool designers by other means.

```
library {
  namespace platform {
    serviceclass SCMult;
    serviceclass SCTCMult : SCMult; /* two's complement multiplication */
    serviceclass SCFPMult : SCMult; /* floating point multiplication */
  }
}
```

Listing 3.1 – Declaration of service classes.

3.1.1 Hierarchies

In the Service Relation Model, the declaration of a service class belongs to the library domain. A service class is allowed to have other service classes as members in addition to the services of a concrete model and all service classes are implicitly assumed to be member of the general `SCService` service class. In the `sSRM` language such subsumption relations between service classes are a part of the declaration.

Example 3.1 Services representing operations on numbers can be organized into service classes. Three such classes could be `SCPow` representing services for computing the power function, `SCSub` representing services for doing subtraction and `SCMult` representing services for doing multiplication. Each of these classes could be decomposed into other classes representing operations on different types of numbers (e.g. two's complement integers, floating point reals and so on). Listing 3.1 shows how the `SCMult` service class and two other sub-classes representing multiplication of two's complement and floating point numbers is declared in the Service Relation Model. ■

A service in the library domain can be declared to be a member of one of more service classes. Any instances of the service in a model will be considered a member of those service classes. All services are implicitly a member of the service class `SCService`.

Example 3.2 Listing 3.2 shows a revisited description of the FPU component used in the examples of sections 2.1.1 and 2.1.2 where the individual services have been declared as members of different service classes. ■

The service classes of a library are naturally organized into a hierarchy called a *service class hierarchy*. Similarly, the service classes of a library and the services in a model based on the library are naturally organized into a hierarchy called a *service hierarchy*.

```
library {
  namespace platform {
    template FPU {
      interface Processor : obj(CoProcToProc);
      service FpAdd  : SCFPAdd  { export(Processor); }
      service FpDiv  : SCFPDiv  { export(Processor); }
      ...
      service FpMult : SCFPMult { export(Processor); }
    }
  }
}
```

Listing 3.2 – Declaration of service class membership.

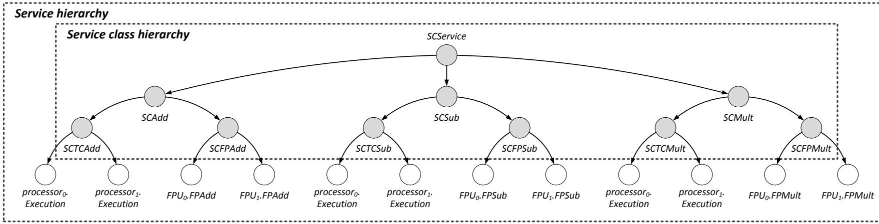


Figure 3.1 – Service Hierarchies. The figure shows parts of the service class hierarchy and the service hierarchy for the example of section 2.1.2. Gray nodes represent service classes in the library and white nodes services in the model.

Definition 3.0 (Service Class, sc^ℓ) A service class sc^ℓ is a singleton $\langle P \rangle$ where $P \subset SC^\ell$ is a set of super-classes of sc^ℓ. The set SC^ℓ of all service classes in a library is ordered as follows:

$$\forall sc_0^\ell, sc_1^\ell \in SC^\ell : (sc_0^\ell \sqsubseteq sc_1^\ell) \leftrightarrow (sc_1^\ell \in sc_0^\ell.P)$$
$$\forall sc_0^\ell, sc_1^\ell, sc_2^\ell \in SC^\ell : ((sc_0^\ell \sqsubseteq sc_1^\ell) \land (sc_1^\ell \sqsubseteq sc_2^\ell)) \rightarrow (sc_0^\ell \sqsubseteq sc_2^\ell)$$

The operator $\sqsubseteq$ naturally imposes a hierarchy on the service classes. This hierarchy is called the *service class hierarchy*. □

A service class hierarchy is specific to a library but is shared for all designs based on that library. Notice that the service class hierarchy does not include the services of the library. Recall that a service in a library is part of a template and thus represents a possibly empty set of service instances in a given model. A service in the library must, however, be associated with one or more service classes meaning that instances based on the service belongs to the associated service classes. Formally, we define a service in the library domain as:

Definition 3.1 (Service, s^ℓ) A service s^ℓ is a pair $s^\ell = \left\langle o^\ell, SC^\ell \right\rangle$ where o^ℓ is the parent option of the service and SC^ℓ is the set of super classes of all instances $s_0, s_1, ..., s_n$ of s^ℓ in any model. □

This leads to the following definition of a service in the model domain:

Definition 3.2 (Service, s) A service s is a pair $s = \langle c, SC \rangle$ where c is the parent component and SC is the set of super classes to which s belongs. □

For a model $m = \langle C, R \rangle$, the services of m and the service class hierarchy of the associated library ℓ are combined into a hierarchy called the *service hierarchy*. Figure 3.1 illustrates the relationship between the service class hierarchy of a library and the service hierarchy of a model.

Definition 3.3 (Service Class, sc) A service class sc in the model domain is a pair $sc = \left\langle sc^\ell, S \right\rangle$ where sc^ℓ is the corresponding service class in the library domain and $S = \{s_0, s_1, \ldots, s_n\}$ is the set of service instances in the model which belongs to the service class. □

Service classes is a simple yet powerful way of assigning meaning to services but it has its limitations and problems. One problem is that the meaning of a service class is not captured formally. In practice, this will lead to misunderstandings showing up as errors for the end user. A more formal approach to describing the meaning of service classes can, however, always be added later. A limitation is that a service must explicitly be declared to be a member of a service class. This means that we cannot later add new classes and then infer membership of services using the existing assertions or other knowledge.

3.2 Assertions

Many components naturally have dependencies, in the form of services that must be accessible to them, in order for them to function properly. Such dependencies are captured by means of assertions. A component can be associated with one or more assertions. An assertion is an expression that must evaluate to the truth value true in order for the use of the component to be considered valid.

$$
\begin{array}{lll}
\mathbf{Expr}_\alpha & \texttt{:=} & e_0 \ \texttt{and} \ e_1 \\
 & | & e_0 \ \texttt{or} \ e_1 \\
 & | & e_0 \ \texttt{implies} \ e_1 \\
 & | & \texttt{not} \ e_0 \\
 & | & sc \ \texttt{@} \ i \\
 & | & sc \ \texttt{@} \ s
\end{array}
$$

Figure 3.2 – Abstract syntax of alpha expressions. Here $e_0, e_1 \in \mathbf{Expr}_\alpha$ are sub-expressions, $sc \in SC$ a service class, $s \in S$ a service and $i \in I$ an interface

$Eval_\alpha : \mathbf{Expr}_\alpha \times (S \cup I \rightarrow \mathcal{P}(S)) \rightarrow \{\texttt{true}, \texttt{false}\}$

$Eval_\alpha(e_0 \ \mathbf{and} \ e_1, SA_m)$	$=$	$Eval_\alpha(e_0, SA_m) \wedge Eval_\alpha(e_1, SA_m)$
$Eval_\alpha(e_0 \ \mathbf{or} \ e_1, SA_m)$	$=$	$Eval_\alpha(e_0, SA_m) \vee Eval_\alpha(e_1, SA_m)$
$Eval_\alpha(e_0 \ \mathbf{implies} \ e_1, SA_m)$	$=$	$Eval_\alpha(e_0, SA_m) \rightarrow Eval_\alpha(e_1, SA_m)$
$Eval_\alpha(\mathbf{not} \ e_0, SA_m)$	$=$	$\neg \ Eval_\alpha(e_0, SA_m)$
$Eval_\alpha(sc \ \texttt{@} \ i, SA_m)$	$=$	$sc.S \cup SA_m[i] \neq \emptyset$
$Eval_\alpha(sc \ \texttt{@} \ s, SA_m)$	$=$	$sc.S \cup SA_m[s] \neq \emptyset$

Table 3.1 – Evaluation of alpha expressions defined as a function taking as input an expression and the service availablity information (SA_m) of the model. Here $e_0, e_1 \in \mathbf{Expr}_\alpha$, $s \in S$, $i \in I$ and $sc \in SC$.

3.2.1 Assertions in Models

Definition 3.4 (Assertion, a) An assertion a is a pair $a = \langle c, e \rangle$ where $c \in C$ is the parent component to which the assertion belongs in the model $m = \langle C, R \rangle$ and $e \in \mathbf{Expr}_\alpha$ is an alpha expression. Figure 3.2 shows the BNF for alpha expressions. □

The meaning of an alpha expression is defined by means of the recursive evaluation function $Eval_\alpha$ given in Table 3.1. The function takes as argument an alpha expression and the map representing the service availablity information of the associated model. The output of the function is the validity (true or false) of the expression and thus the parent assertion with respect to that model.

As can be seen, the meaning of the four logical operators are the usual. The @-expression is used to test for the availablity of a member of a given service class at some point of interest within the component. A point of interest can be any service or interface in the component to which the assertion belongs. The @-expression can only be used to test if a member of a given service class is available – it cannot be used to test for the availablity of a specific service instance which would lead to tight coupling.

```
model quadric {
  component discriminant {
    interface context   : sub(ContextToCaller);
    interface processor : sub(ProcessorToSwe);

    assert(
          (SCExecution @ processor)
      and (SCFPSub @ processor)
      and (SCFPMul @ processor)
      and (SCFPPow @ context)
      and (SCPNReadChannel @ context)
      and (SCPNWriteChannel @ context)
    );
  }
}
```

Listing 3.3 – sSRM description of the discriminant component from the example of section 2.1.2.

Example 3.3 As an example consider the discriminant component of example from section 2.1.2. This component has several dependencies that must be satisfied in order for the component to be able to function properly. Listing 3.3 shows an sSRM description of the discriminant component that includes an assertion for enforcing these dependencies. Here we assume that the three services FPPow, FPSub and FPMul have been declared to be members of the service classes SCFPPow, SCFPSub and SCFPMul respectivily. Similarly, we also assume that the Execution service of the processor has been declared to be a member of the SCExecution service class and that the write and read services of the process network API component have been declared to be members of the service classes SCPNReadChannel and SCPNWriteChannel. ■

Assertions is a simple concept but should be applied with some care. Just because a service is available at some point of interest it does not necessarily mean that it is also free for use. Some services cannot necessarily be shared between several users. An example of such a service is a monolithic service representing the capabilities of a timer. When this service is used by one component it cannot simultaneously be used by another. This is so because the timer represented by the service is a limited resource that can only be used to count one thing at a time. Resource handling in the Service Relation Model will be presented in the next section.

3.2.2 Assertions in the Library

In this section, the difference between assertions in the library domain and assertions in the model domain and the design expansion process for assertions

$$
\begin{array}{lcl}
\mathbf{Expr}_{\alpha}^{\ell} & := & e_0^{\ell}\ \texttt{and}^{\ell}\ e_1^{\ell} \\
 & | & e_0^{\ell}\ \texttt{or}^{\ell}\ e_1^{\ell} \\
 & | & e_0^{\ell}\ \texttt{implies}^{\ell}\ e_1^{\ell} \\
 & | & \texttt{not}^{\ell}\ e_0^{\ell} \\
 & | & sc^{\ell}\ \texttt{@}^{\ell}\ i^{\ell} \\
 & | & sc^{\ell}\ \texttt{@}^{\ell}\ s^{\ell}
\end{array}
$$

Figure 3.3 – Alpha expressions BNF. Here $e_0^{\ell}, e_1^{\ell} \in \mathbf{Expr}_{\alpha}^{\ell}, sc^{\ell} \in SC^{\ell}, s^{\ell} \in S^{\ell}$ and $i^{\ell} \in I^{\ell}$

will be explained. In the library domain, an assertions is defined as:

Definition 3.5 (Assertion, a^{ℓ}) An assertion a^{ℓ} is a pair $a^{\ell} = \left\langle o^{\ell}, e \right\rangle$ where $o^{\ell} \in O^{\ell}$ is the parent option in the library to which the assertion belongs and $e \in \mathbf{Expr}_{\alpha}^{\ell}$ is an alpha library expression. □

The abstract syntax of alpha expression in the library domain is given in figure 3.3. Syntactically there is little difference between alpha expression in the model and in the library. Semantically, however, there is a major difference due to the fact that entities of the library domain refer to classes where as entities in the model domain refers to instances. The difference lies in how the `@`-expression is interpreted when the point of interest is a service or interface defined in another option than the one the assertion belongs to. In this case, the meaning of the `@`-expression is taken to be the conjunction of the availablity test for each included option. Table 3.2 gives a definition of a function $\mathcal{A}_{\ell \to m}$ defining how an alpha expression $\mathbf{Expr}_{\alpha}^{\ell}$ of a particular option and component is expanded into an alpha expression $\mathbf{Expr}_{\alpha}$ in the model domain.

Example 3.4 Listing 3.4 shows a model containing a single component. The component is based on a template that contains an interface `i1`, an option `o` and an assertion. The assertion contains a reference to an interface `i2` contained within the option. The component of the model includes the option *o* twice. In the model, the second `@`-expression of the assertion in the library has been replaced by a conjunction of `@`-expressions requiring an instance of the service class `SCSomeOtherClass` to be accessible at both interface `o1.i2` and `o2.i2`. ■

3.3 Resources and Resource Claims

In this section, the concepts of *resources* and *resource claims* are presented. Collectively, these two concepts provides the means to deal with services (called

$$\mathcal{A}_{\ell\to m} : \mathbf{Expr}^{\ell}_{\alpha} \times C^{\delta} \times O^{\delta} \times M \to \mathbf{Expr}_{\alpha}$$

$$\begin{aligned}
\mathcal{A}_{\ell\to m}(e^{\ell}_0 \ \mathbf{and}^{\ell}\ e^{\ell}_1, c^{\delta}, o^{\delta}, m) &= \mathcal{A}_{\ell\to m}(e^{\ell}_0, c^{\delta}, o^{\delta}, m) \ \mathbf{and}\ \mathcal{A}_{\ell\to m}(e^{\ell}_1, c^{\delta}, o^{\delta}, m) \\
\mathcal{A}_{\ell\to m}(e^{\ell}_0 \ \mathbf{or}^{\ell}\ e^{\ell}_1, c^{\delta}, o^{\delta}, m) &= \mathcal{A}_{\ell\to m}(e^{\ell}_0, c^{\delta}, o^{\delta}, m) \ \mathbf{or}\ \mathcal{A}_{\ell\to m}(e^{\ell}_1, c^{\delta}, o^{\delta}, m) \\
\mathcal{A}_{\ell\to m}(e^{\ell}_0 \ \mathbf{implies}^{\ell}\ e^{\ell}_1, c^{\delta}, o^{\delta}, m) &= \mathcal{A}_{\ell\to m}(e^{\ell}_0, c^{\delta}, o^{\delta}, m) \ \mathbf{implies}\ \mathcal{A}_{\ell\to m}(e^{\ell}_1, c^{\delta}, o^{\delta}, m) \\
\mathcal{A}_{\ell\to m}(\mathbf{not}^{\ell}\ e^{\ell}_0, c^{\delta}, o^{\delta}, m) &= \mathbf{not}\ \mathcal{A}_{\ell\to m}(e^{\ell}_0, c^{\delta}, o^{\delta}, m) \\
\mathcal{A}_{\ell\to m}(sc^{\ell}\ @^{\ell}\ s^{\ell}, c^{\delta}, o^{\delta}, m) &= \begin{cases} sc_{\ell\to m}(sc^{\ell}, m)\ @\ S_{\delta\to m}(s^{\ell}, c^{\delta}, o^{\delta}) & \text{if } s^{\ell} \in o^{\delta}.o^{\ell}.S^{\ell} \\ \widehat{\mathbf{and}}_{o^{\delta} \in SO(s^{\ell}, c^{\delta})} \left(sc_{\ell\to m}(sc^{\ell}, m)\ @\ S_{\delta\to m}(s^{\ell}, c^{\delta}, o^{\delta})\right) & \text{otherwise} \end{cases} \\
\mathcal{A}_{\ell\to m}(sc^{\ell}\ @^{\ell}\ i^{\ell}, c^{\delta}, o^{\delta}, m) &= \begin{cases} sc_{\ell\to m}(sc^{\ell}, m)\ @\ I_{\delta\to m}(i^{\ell}, c^{\delta}, o^{\delta}) & \text{if } i^{\ell} \in o^{\delta}.o^{\ell}.I^{\ell} \\ \widehat{\mathbf{and}}_{o^{\delta} \in IO(i^{\ell}, c^{\delta})} \left(sc_{\ell\to m}(sc^{\ell}, m)\ @\ I_{\delta\to m}(i^{\ell}, c^{\delta}, o^{\delta})\right) & \text{otherwise} \end{cases}
\end{aligned}$$

$$SO(s^{\ell}, c^{\delta}) = \{o^{\delta} : o^{\delta} \in c^{\delta}.O^{\delta} \wedge o^{\delta}.o^{\ell} = s^{\ell}.o^{\ell}.S^{\ell}\}$$

$$IO(i^{\ell}, c^{\delta}) = \{o^{\delta} : o^{\delta} \in c^{\delta}.O^{\delta} \wedge o^{\delta}.o^{\ell} = i^{\ell}.o^{\ell}.I^{\ell}\}$$

Table 3.2 – Here $e^{\ell}_0, e^{\ell}_1 \in \mathbf{Expr}^{\ell}_{\alpha}$, $sc^{\ell} \in SC^{\ell}$, $i^{\ell} \in I^{\ell}$, $s^{\ell} \in S^{\ell}$, $\widehat{\mathbf{and}}$ is an n-ary operator defined as: $((((\texttt{true}\ \mathbf{and}\ x_0)\ \mathbf{and}\ x_1)\ldots)\ \mathbf{and}\ x_n)$ and $sc_{\ell\to m} : SC^{\ell} \times M \to SC$ is a function for retrieving the service class in the model domain corresponding to a given service class in the library domain and a model.

```
library {
  namespace ns {
    ...
    template t {
      interface i1 : sub(SomeRelation);
      option o { interface i2 : sub(SomeRelation); }
      assert(
        (SCSomeClass @ i1) or (SCSomeOtherClass @ o.i2)
      );
    }
  }
}

design MyDesign {
  component c : ns.t {
    include o1 : o;
    include o2 : o;
  }
}

model MyDesign {
  component c {
    interface i1    : sub(SomeRelation);
    interface o1.i2 : sub(SomeRelation);
    interface o2.i2 : sub(SomeRelation);
    assert(
         (SCSomeClass @ i1)
      or ((SCSomeOtherClass @ o1.i2) and (SCSomeOtherClass @ o2.i2))
    );
  }
}
```

Listing 3.4 – Assertion in the library

resources) that are characterized by being available only in finite quantities. The example below illustrates two of the central problems addressed by the concepts presented in this section.

Example 3.5 Figure 3.4 shows a graphical representation of a service relation model representing a simple dual processor platform. Both processors are running an instance of an operating system (OS_0 and OS_1). In order to provide time sliced multi-processing, each of the operating systems requires *exclusive access* to a timer resource. The platform provides two timer/counters that can be used for this purpose. In the model, a timer is represented as a component providing a start and a stop service. The start service is declared to be a member of the service class `SCTimerStart` and the stop service is declared to be a member of the `SCTimerStop` service class.

In the model, the dependency of an operating system component on a timer is modelled using an assertion. The assertion states that an instance of the `SCTimerStart` and `SCTimerStop` services should be available at the proces-

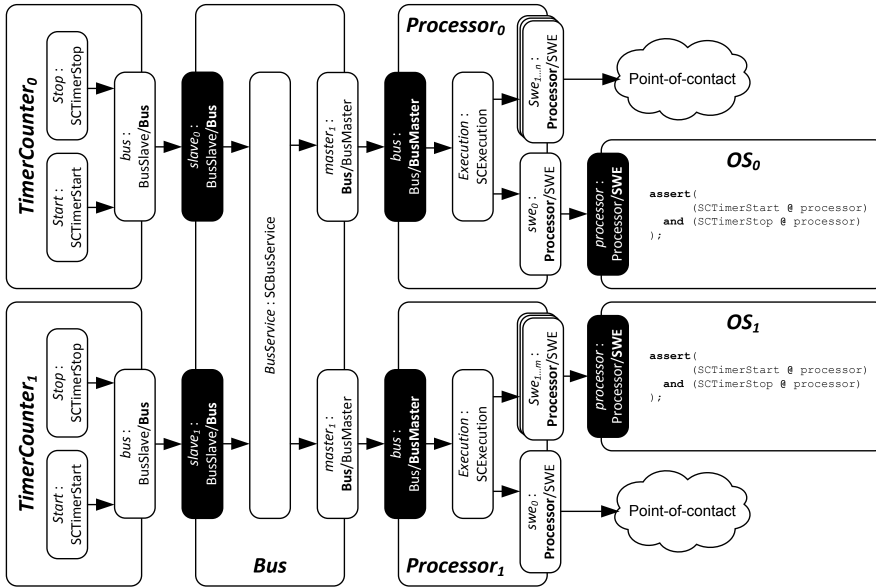


Figure 3.4 – Service relation model of a platform including two timer/counters and two operating systems

sor interface of the operating system component. The model is consistent with respect to assertions because both assertions are true. A problem with the assertions, however, is that they cannot be used to claim exclusive access and, as a consequence, a single timer/counter can be used to satisfy an infinite number of timer requirements. This means that the model of Figure 3.4 will still be consistent with respect to assertions if one of the timer/counters are removed.

The platform of Figure 3.4 has two points of contacts where the platform may interface with an upper layer of software. The capabilities of the platform with respect to the two points-of-contact can be determined by computing the sets of services available at the unconnected `swe` interfaces of the two processors. Both sets will contain the services provided by the two timer/counters despite the fact that these are in fact used internally by the platform. ■

The underlying problem is that services are considered to be available in infinite quantities and that no concept of ”exclusive access” to a service exists. These two shortcomings of the basic Service Relation Model are addressed by introducing the concepts of resources and resource claims. A resource represents an entity that is only available in a limited quantity through one or more services. A key feature of the resource concept is that it integrates seamlessly

with the existing concepts so that our understanding of the existing concepts in general and the concept of service flow in particular is not affected. The concept of a resource claim acts as a replacement for the concept of assertions when dealing with resources. A resource claim can be used by a component to declare that it requires exclusive access to a resource.

3.3.1 Resources

Resources are modeled using the concepts of *resources* which are logically associated with the inner workings of components like service. A resource is a declaration of some amount of resource being provided by a component through its services. In the `sSRM` language, the syntax for declaring a resource is:

```
resource <name> : <ResourceClass> {
    quantity = <Integer>;
    export (<ServiceList>);
}
```

As can be seen, the declaration consists of a name, a resource class, a quantity and a list of services. The name is used to identify to the resource in interactions with the user. Like services, resources are organized in a resource class hierarchy. The purpose of this hierarchy is similar to that of the service class hierarchy: to assign meaning to resources and enable referencing of resources between components without imposing tight coupling. A resource is associated with a quantity, an integer, representing the amount of resource available. Access to a resource is governed by one or more services. This means that a resource cannot be exported to interfaces directly – all resources are accessed through services. A resource is not itself a service although it shares many similarities with services. This is done to separate the concept of resources from that of services computationally. If we had adopted the view that a service is a resource then it would not be possible to do the service availability analysis without taking resources into account. Also, many resources cannot be represented by a single service. For example, a timer cannot be considered a service unless we want to merge all of its provided capabilities (i.e. start, stop, reset) into a single service. Note that a resource can only be made available through services belonging to the same component as the resource.

Example 3.6 Figure 3.5 shows an example of an 8k byte bus-mounted memory component modelled using a resource. The component consists of an interface, a service and a resource. The resource provides 8192 quantities (bytes) of a resource belonging to the `RCMemory` resource. The resource is made accessible

```
model m {
  component memory {
    interface bus : obj(BusSlaveToBus);
    service memoryservice : SCMemoryService {
      export bus;
    }
    resource memory_resource : RCMemory {
      quantity = 8192;
      export memoryservice;;
    }
  }
}
```

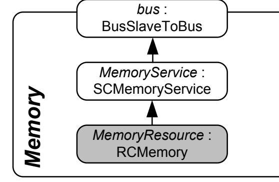


Figure 3.5 – A bus-mounted memory component modelled using a resource. Left: `sSRM` description in the model domain. Right: Graphical representation.

through the memory service meaning that if other components can access the memory service they may also access the resource. ■

3.3.1.1 Resource Availablity

Like services, we can talk about the availablity of a resource. A resource is said to be available at some point x, that can be either a service or an interface, if there exists a path from *any* of the services that the resource is exported through to x in the service flow graph.

Definition 3.6 (Resource Availability Information, RA) Let I_* be the set of interfaces, S_* the set of services and RES_* the set of resources in a model m. The set of services and interfaces $x \in I_* \cup S_*$ where a resource $res \in RES_*$ is available is called the resource availablity information of res and is defined as:

$$\bigcup_{s \in res.S} \overline{AA}[s]$$

where AA is the availablity-at information of m. The resource availablity information of the model m is a map:

$$\overline{RA} : RES_* \to \mathcal{P}(I_* \cup S_*)$$

mapping resources to their resource availablity information. □

Notice that a resource may be available through different services at different locations in a model. A resource representing a hardware FIFO exported through two services representing reading and writing the content of the FIFO, for ex-

ample, may only be available through one of the two services in some parts of a model and through the other in other parts.

3.3.2 Resource Claims

The concept of a resource claim is used to declare that a component requires exclusive access to some amount of resource in order for it to function properly. Exclusive access means that some quantity of resource can at most be claimed by one resource claim. Below, the syntax for a resource claim in the `sSRM` language is given:

claim <*name*> : <*ResourceClass*> {
  quantity = <*Integer*> ;
  mp = <*Multiplicity*> ;
  where <$\mathbf{Expr}_\omega$> ;
}

The structure of the claim is somewhat similar to that of a resource. Claims are associated with a name to help identifying them when interacting with the user. The resource class associated with a claim specifies the set of resources that may satisfy the claim. For example, a claim associated with the resource class `RCMemory` can only be satisfied by resources belonging to this class. A resource claim can claim multiple units of resource. The number of resource units needed is called the quantity of the claim. Besides the name, resource class association and the quantity, a claim also consist of a *multiplicity flag* and a *where expression*. The multiplicity flag is used to specify whether the claim can be satisfied by multiple (different) resources (multiplicity = ”many”) or must be satisfied by exactly one resource (multiplicity = ”one”). A component requiring memory, for example, could set the multiplicity flag to ”one” stating that the quantity of memory claimed must be provided by the same resource. For claims where the quantity is 1 there is no difference between the two kinds of multiplicity as 1 quantity of resource is the smallest unit that can be considered. Together with the resource class, the where expression is used to impose constraints on the resources that may satisfy the claim. Table 3.6 shows the abstract syntax for such expressions.

The where expression acts as a function for filtering out resources on the basis of the service availablity information of the model and on the values of any parameters associated with the classes of resource being claimed. The latter, parameterized resources, will be presented momentarily. The evaluation of a where expression can be thought of as a function:

$$\Omega_e : \mathbf{Expr}_\omega \times RES \times (S \times I \to \mathcal{P}(S)) \to \{\texttt{true}, \texttt{false}\}$$

$$
\begin{array}{lll}
\textbf{Expr}_\omega & := & e_0 \texttt{ + } e_1 \mid e_0 \texttt{ - } e_1 \mid e_0 \texttt{ * } e_1 \mid e_0 \texttt{ / } e_1 \mid \texttt{-}e \\
 & \mid & e_0 \texttt{ > } e_1 \mid e_0 \texttt{ >= } e_1 \mid e_0 \texttt{ < } e_1 \mid e_0 \texttt{ <= } e_1 \mid e_0 \texttt{ = } e_1 \mid e_0 \texttt{ != } e_1 \\
 & \mid & e_0 \texttt{ and } e_1 \mid e_0 \texttt{ or } e_1 \mid e_0 \texttt{ implies } e_1 \mid \texttt{not } e_0 \\
 & \mid & \textit{constant} \mid \textit{resource_parameter} \\
 & \mid & sc \texttt{ @ } s \mid sc \texttt{ @ } i \\
 & \mid & \texttt{->(}sc \texttt{ @ } s\texttt{)} \mid \texttt{->(}sc \texttt{ @ } i\texttt{)}
\end{array}
$$

Figure 3.6 – Abstract syntax for where expressions in the model domain

```
model m {
  component SoftwareComponent {
    interface processor : sub(ProcessorToSwe);
    claim memory : RCMemory {
      quantity = 1200;
      mp = one;
      where ->(SCMemoryService @ processor);
    }
  }
}
```

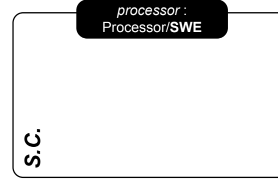


Figure 3.7 – A software component requiring 1200 bytes memory modelled using a resource claim. There is no graphical representation associated with claims.

taking as input the expression, a resource and the service availablity information of the model and returning true or false depending on whether or not the resource is a possible candidate for satisfying the claim. The definition of the Ω_e function is, with the exception of the expressions on the last line, trivial and can be found in appendix B. The two expressions in the last line are called *is-available-through* expressions and are used to constrain the resources being considered using the service flow of the model. The is-available-through expression `->(`sc `@` x`)` evaluates to true if the resource is available through an instance of the service class sc at some point of interest x in the component of the claim. Formally, the value of an is-available-through expression `->(`sc `@` x`)` can be expressed as

$$\overline{RA}[res] \cap sc.S \cap SA[x] \neq \emptyset$$

where RA is the resource availability information and SA the service availability information of the model.

Example 3.7 Figure 3.7 shows an example of a software component requiring memory. The component is modelled using a resource claim, claiming 1200 quantities of the `RCMemory` resource. The where expression constrains the resources under considerations to those available through an instance of the `SCMemoryService` class at the `processor` interface of the component. Notice also that resource claims does not have a graphical representation in the graphical notation. ∎

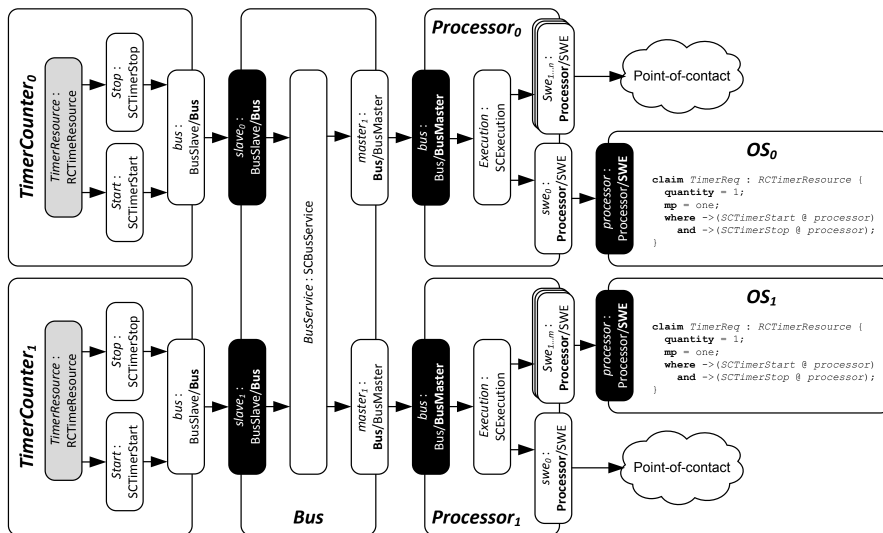


Figure 3.8 – Revisited model of the platform from Figure 3.4 using resources and resource claims.

Example 3.8 Figure 3.8 shows a revisited model of the platform used to motivate the need for resources and resource claims presented previously. In the model of Figure 3.8, the two timer/counters are modelled by means of resources belonging to the resource class `RCTimerResource`. The resources are made accessible for the rest of the system through the start and stop services. The dependency of an operating system on a timer is modelled by means of a resource claim instead of an assertion.

In practice, requiring that a given timer resource is available through some set of services is not sufficient to properly model the dependency of an operating system on a timer. In addition to requiring access to a set of timer services, an operating system will also require that the timer is connected to the interrupt sub-system of the processor allowing for the timer to generate the periodic interrupts used for driving time-sliced multiprocessing. Unfortunately, this requirement cannot be expressed using a resource claim. This shortcoming will be discussed in more detail later in section 3.3.6. ■

```
library {
  namespace MyLibrary {
    resourceclass RCQueue<Depth, Size> : RCResource;
    ...

    template MyComponentTemplate {
      ...
      resource MyQueueResource : RCQueue<10,10> {
        quantity = 1;     '
        export (out_if);
      }
    }

    template MyOtherComponentTemplate {
      ...
      claim MyClaim : RCQueue<Depth,Size> {
        quantity = 1;
        mp = one;
        where Depth > 10 and Size == 10 and ->(SCEnqueue @ in_if);
      }
    }
  }
}
```

Listing 3.5 – Example of the use of parameterized resource classes

3.3.3 Parameterized Resources and Resource Classes

As previously mentioned, resource classes are used to assign meaning to the resources of a model and as a way to enable referencing of resource between components without imposing tight coupling. Resource classes differs from service classes in that they can be *parameterized*. A parameterized resource class is an abstraction of a set of similar resources associated with a set of integer parameters that must be defined for each resource declaring its membership of the class. The values of the parameters associated with a resource class can be referenced from within the where expressions of claims. The next example illustrates the use of parameters with resource classes.

Example 3.9 Listing 3.5 shows an example of the use of parameterized resource classes. The parameterized resource class `RCQueue<Depth,Size>` is used for representing queue resources and has two parameters named `Depth` and `Size`. The `Depth` parameter specifies the minimum number of messages that a particular queue can facilitate and `Size` the maximum size of each message (in bytes). The component `MyComponentTemplate` provides a resource belonging to the parameterized resource class `RCQueue<Depth,Size>` with `Depth` = 10 and `Size` = 10 meaning that it can contain a maximum of 10 messages of 10 bytes each. Finally, the component `MyOtherComponentTemplate` contains a claim for resources of type `RCQueue<Depth,Size>`. The where expression limits

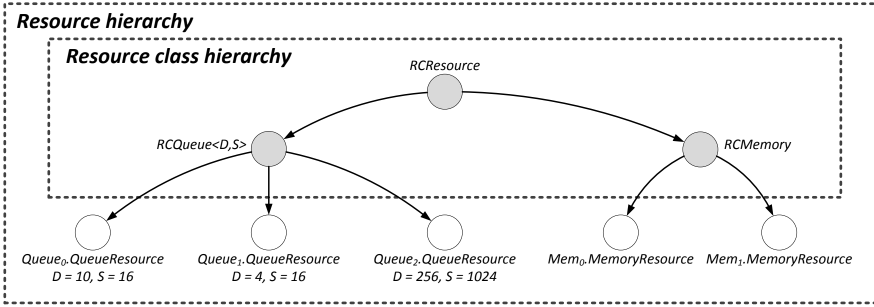


Figure 3.9 – Resource class hierarchy/resource hierarchy example.

the usable resources to those that has a minimum depth and size of 10 and are available through an instance of the `SCEnqueue` service class. ■

The example is given in the library domain because the resource class declarations naturally belongs to this domain. Formally, a resource class in the library domain is defined as:

Definition 3.7 (Resource Class, rc^ℓ, RC^ℓ) A resource class rc^ℓ is a pair $\langle RP, P\rangle$ where $RP = \{rp_0, rp_1, \ldots, rp_n\}$ is a possibly empty set of resource parameters and $P \subset RC^\ell$ is a set of super-classes of rc^ℓ. The set RC^ℓ of all resource classes in the library is ordered as follows:

$$\forall rc_0^\ell, rc_1^\ell \in RC^\ell : (rc_0^\ell \sqsubseteq rc_1^\ell) \leftrightarrow (rc_1^\ell \in rc_0^\ell.P)$$
$$\forall rc_0^\ell, rc_1^\ell, rc_2^\ell \in RC^\ell : ((rc_0^\ell \sqsubseteq rc_1^\ell) \wedge (rc_1^\ell \sqsubseteq rc_2^\ell)) \rightarrow (sc_0^\ell \sqsubseteq sc_2^\ell)$$

The operator $\sqsubseteq$ naturally imposes a hierarchy on the resources classes. This hierarchy is called the *resource class hierarchy*. □

A resource class $rc^\ell = \langle RP, P\rangle$ where $RP \neq \emptyset$ is refered to as a *parameterized resource class* where as a class where $RP = \emptyset$ is refered to simply as a *resource class*. In the model domain, a resource class is defined as:

Definition 3.8 (Resource Class, rc, RC) A resource class rc in the model domain is a pair $rc = \left\langle rc^\ell, RES\right\rangle$ where rc^ℓ is the corresponding resource class of the library and $RES = \{res_0, res_1, \ldots res_n\}$ is the set of resources in a given model that provides an instance of the rc^ℓ resource class. □

As was the case with service classes, the resource classes of a library are

organized into a *resource class hierarchy* similar to the service class hierarchy and the resource classes of a library and the resources of a model m can also be ordered into a *resource hierarchy*. Figure 3.9 illustrates the relationship between the two hierarchies for a model containing three queue resources and two memory resources.

3.3.4 Resources & Resource Claims in Models

Formally, the concepts of a resource and a resource claim in the model domain is defined as:

Definition 3.9 (Resource, res**,** RES**)** A resource is a quadruple $res = \left\langle rc^{\ell}, RP_v, q, S\right\rangle$ where $rc^{\ell} \in RC^{\ell}$ is the resource class, PR_v is a set of resource parameter values, $q \in \mathbb{N}$ is the quantity of resource being provided and S is the set of services through which the resource is accessible. A resource parameter value $rp_v \in RP_v$ is a pair $rp_v = \left\langle rp^{\ell}, val\right\rangle$ where $rp^{\ell} \in rc^{\ell}.RP$ is a resource parameter of the resource class associated with the resource and $val \in \mathbb{Z}$ is the value of the resource parameter for the resource. For a resource $res = \left\langle rc^{\ell}, RP_v, q, S\right\rangle$ the following is always true:

$$\forall rp \in rc^{\ell}.RP : \left(\exists rp_v \in RP_v : \left(rp_v.rp^{\ell} = rp\right)\right) \wedge$$
$$\forall rp_v, rp'_v \in RP_v \times RP_v : \left(rp_v.rp^{\ell} = rp'_v.rp^{\ell} \Rightarrow rp_v = rp'_v\right)$$

meaning that a resource must provide exactly one value for each of the resource parameters defined in the associated resource class. □

Definition 3.10 (Claim, cl**,** CL**)** A claim is a quadruple $cl = \left\langle mp, rc^{\ell}, q, \omega\right\rangle$ where $mp \in \{one, many\}$ is the multiplicity of the claim, rc^{ℓ} is the resource class being claimed, $q \in \mathbb{N}$ the quantity of resource being claimed and $\omega \in \mathbf{Expr}_{\omega}$ is an expression that must evaluate to true for a given resource res in order for res to be able to satisfy the claim. □

3.3.4.1 Resource Distribution

Notice that neither of the two concepts can be used to describe a particular distribution of resources to resource claims. The reason for this is that the purpose of the Service Relation Model is to provide the foundations for determining such

$$
\begin{array}{lll}
\mathbf{Expr}^{\ell}_{\omega} & := & e_0 \texttt{ + } e_1 \mid e_0 \texttt{ - } e_1 \mid e_0 \texttt{ * } e_1 \mid e_0 \texttt{ / } e_1 \mid \texttt{-}e \\
 & \mid & e_0 \texttt{ > } e_1 \mid e_0 \texttt{ >= } e_1 \mid e_0 \texttt{ < } e_1 \mid e_0 \texttt{ <= } e_1 \mid e_0 \texttt{ = } e_1 \mid e_0 \texttt{ != } e_1 \\
 & \mid & e_0 \texttt{ and } e_1 \mid e_0 \texttt{ or } e_1 \mid e_0 \texttt{ implies } e_1 \mid \texttt{not } e_0 \\
 & \mid & \textit{constant} \mid \textit{resource_parameter} \mid \textit{parameter} \\
 & \mid & sc \texttt{ @ } s \mid sc \texttt{ @ } i \\
 & \mid & \texttt{->(}sc \texttt{ @ } s\texttt{)} \mid \texttt{->(}sc \texttt{ @ } i\texttt{)}
\end{array}
$$

Figure 3.10 – Abstract syntax for where expressions in the library domain. The only difference between the where expressions of the model domain and the library domain is that parameters can be referenced in the library domain. In the model domain, all parameters are replaced by their constant values.

distributions rather than representing them. Conceptually, however, the distribution of a resource with quantity $q \in \mathbb{N}$ amongst n different parties can intuitively be be presented as a set of n non-overlapping intervals, called *resource shares*:

Definition 3.11 (Resource Share, rs, RS) A resource share rs is an integer interval $rs = [rs_l, rs_u]$ where $rs_l \in \mathbb{N}$ is the lower bound and $rs_u \in \mathbb{N}$ the upper bound. □

The concept of resource shares is not part of the Service Relation Model but is used by the tool presented in the next chapter.

3.3.5 Configurability

In this section, the concepts of resources and resource classes in the library and design domains will be presented. In the model, the quantities associated with resources and claims are modelled as a constant integer which does not provide much flexibility and, consequently, hinders re-usability. In many cases, the quantity of resource provided or claimed by a component is naturally dependent on the configuration of the component. For example, the number of byte resources provided by a memory component is dependent on the size of the memory – a property that may be configurable. To enable such dependencies, we introduce the concepts of *parameters* and *quantity expressions*.

3.3.5.1 Parameters & Quantity Expressions

Intuitively, a parameter is a constant associated with a component template whose exact value is defined as part of the component instantiation process.

In the template, the name of the parameter can be used in place of a constant value. The use of parameters is not limited to quantities and may also be used in where expressions and for defining the value of resource parameters. To formally enable the use of parameters the concepts used in the model domain are replaced by so called *quantity expressions* in the library domain. Figure 3.11 shows the abstract syntax for such expressions. In addition to allowing the use of parameter values, quantity expressions also allows for some degree of computation to be carried out which adds an extra dimension of flexibility to the descriptions. Similarly, the syntax for the where expressions used in the model domain is replaced by another and very similar syntax (given in Figure 3.10) that allows parameters to be referenced by name. The next example illustrates the different uses of parameters:

Example 3.10 The code in listing 3.6 shows a simple example of the different uses of parameters.

The library of the example contains two component templates called `QueueProvider` and `QueueConsumer`. The `QueueProvider` template contains a resource of the type `RCQueue<Depth,Size>` that has been parameterized so that the quantity of resource being provided (i.e. number of queues) and the values of the two resource class parameters are configurable by means of three parameters `pDepth`, `pSize` and `pQueuesProvided`. The `QueueConsumer` component contains a resource claim on the parameterized resource class `RCQueue<Depth,Size>`. The component has been made configurable by means of three parameters `pDepth`, `pSize` and `pQueuesNeeded`. The parameters `pDepth` and `pSize` are used inside the where expression of the resource claim for constraining the resources that may satisfy the claim and the parameter `pQueuesNeeded` is used for defining the quantity of resource claimed.

Besides the library, the example also include a design including two component instances based on the templates of the library. The values of the parameters are defined as part of the component instantiation. ■

Conceptually, a parameter is associated with a type. For simplicity, we only consider parameters of type integer. The reason for this is that only integer parameters are currently supported by the `xSRM` language and that it is straightforward to encode boolean and enumerable types using integers. Additional types can easily be added if needed.

In the library domain, a parameter is a symbol associated with an option representing an unspecified value:

```
library {
  namespace MyLibrary {
    ...
    template QueueProvider {
      parameter pDepth;
      parameter pSize;
      parameter pQueuesProvided;

      interface out_if : obj(...);

      service Enqueue : SCEnqueue { export(out_if); }
      service Dequeue : SCSequeue { export(out_if); }

      resource res : RCQueue<pDepth,pSize> {
        quantity = pQueuesNeeded;
        export(Enqueue, Dequeue);
      }
    }

    template QueueConsumer {
      parameter pQueuesNeeded;
      parameter pMinDepth;
      parameter pMinSize;

      interface in_if : sub(...);

      claim cl : RCQueue<D,S> {
        quantity = pQueuesNeeded;
        where ->(SCEnqueue @ i) and ->(SCDequeue @ i)
          and (D >= pMinDepth) and (S >= pMinSize);
      }
    }
  }
}

design m {
  ...
  component provider : MyLibrary.QueueProvider {
    set pDepth = 10; set pSize = 16; set pQueuesProvided = 2;
  }
  component consumer : MyLibrary.QueueConsumer {
    set pMinDepth = 2; pMinSize = 10; set pQueuesNeeded = 1;
  }
}
```

Listing 3.6 – Example of the different uses of parameters.

$\mathbf{Expr}_{\text{qty}}$	:=	e_0 `+` e_1 | e_0 `-` e_1 | e_0 `*` e_1 | e_0 `/` e_1 | `-`e
	|	e_0 `>` e_1 | e_0 `>=` e_1 | e_0 `<` e_1 | e_0 `<=` e_1 | e_0 `=` e_1 | e_0 `!=` e_1
	|	e_0 `and` e_1 | e_0 `or` e_1 | e_0 `implies` e_1 | `not` e_0
	|	*constant* | *parameter*

Figure 3.11 – Abstract syntax of quantity expressions

Definition 3.12 (Parameter, p^ℓ) A parameter p^ℓ is a pair $p^\ell = \left\langle o^\ell, id \right\rangle$ where $o^\ell \in O^\ell$ is the parent option and $id \in SYM$ is the symbol representing the parameter. □

An option creates a scope for such symbols where any given symbol can be associated with at most one parameter. The parameters of an option is intuititvily modeled as a map $param : SYM \rightarrow P^{\ell}_{\perp}$ where SYM is the set of all symbols and $P^{\ell}_{\perp} = P^{\ell} \cup \perp$ is the set of all parameters and $\perp$ is used for symbols that are not associated with any parameters.

In the design domain, the assignment of a value to a parameter is represented by means of a "parameter assignment" concept:

Definition 3.13 (Parameter Assignment, p^{δ}_{a}) A parameter assignment p^{δ}_{s} is a pair $p^{\delta}_{a} = \left\langle o^{\delta}, id, value \right\rangle$ where $o^{\delta} \in O^{\delta}$ is option to which the assignment belongs, $id \in SYM$ is the symbol associated with the parameter being assigned a value and $value \in \mathbb{Z}$ is the value to assign to the parameter associated with the symbol id. □

Every option $o^{\delta} = \left\langle c^{\delta}, o^{\ell} \right\rangle$ included in a component c^{δ} must assign values to all parameters associated with the library option o^{ℓ}. Attempting to set the value of a parameter that is not defined is an error.

Parameters can only be referenced from with quantity and where expressions. Expressions can only reference parameters that belongs to the same option as the entity in which the expression (resource or resource claim) is contained. The reason for this is that an option may be included multiple times and the value assigned to their parameters are not necessarily the same. There is one notable exception to this: Parameters defined in the mandatory option can be referenced from expressions belonging to other options as long as the symbol associated with the parameter has not been overwritten in the option.

3.3.5.2 Resources & Resource Claims

The definition of a resource in the library resembles the definition of a resource in the model domain. The main difference between the two is that quantity expressions are used in place of constants for defining the quantity of resource provided and the values of resource parameters.

Definition 3.14 (Resource, res^ℓ**,** RES^ℓ**)** A resource is a quintuple $res = \left\langle o^\ell, rc^\ell, {RP^\ell}_v, q^\ell, S \right\rangle$ where $o \in O^\ell$ is the parent option of the resource, $rc^\ell \in RC^\ell$ is the resource class, ${RP^\ell}_v$ is a set of resource parameter values, $q \in \mathbf{Expr}_{\text{qty}}$ is an expression defining the quantity of resource being provided and S^ℓ is the set of services through which the resource is accessible. A resource parameter value $rp_v \in RP_v$ is a pair $rp_v = \left\langle rp^\ell, val \right\rangle$ where $rp^\ell \in rc^\ell.RP$ is a resource parameter of the resource class associated with the resource and $val \in \mathbf{Expr}_{\text{qty}}$ is an expression defining the the value of the resource parameter for the resource. For a resource $res = \left\langle rc^\ell, {RP^\ell}_v, q, S \right\rangle$ the following is always true:

$$\forall rp \in rc^\ell.RP : \left(\exists rp_v \in RP_v : \left(rp_v.rp^\ell = rp\right)\right) \wedge$$
$$\forall {rp^\ell}_{v,0}, {rp^\ell}_{v,1} \in {RP^\ell}_v \times {RP^\ell}_v : \left({rp^\ell}_{v,0}.rp^\ell = {rp^\ell}_{v,1}.rp^\ell \Rightarrow rp_{v,0} = rp_{v,1}\right)$$

meaning that a resource must provide exactly one value for each of the resource parameters defined in the associated resource class. □

Below the definition of a resource claim in the library domain is given. The definitions differs from the definition of a resource claim in the model domain in that, in the model domain, a quantity expression is used in place of a constant for defining the quantity of resource being claimed.

Definition 3.15 (Claim, cl^ℓ**,** CL^ℓ**)** A claim is a quintuple $cl = \left\langle o^\ell, mp, rc^\ell, q^\ell, \omega^\ell \right\rangle$ where $o^\ell \in O^\ell$ is the parent option of the claim, $mp \in \{one, many\}$ is the multiplicity of the claim, rc^ℓ is the resource class being claimed, $q \in \mathbf{Expr}_{\text{qty}}$ is an expression definting the quantity of resource being claimed and $\omega^\ell \in {\mathbf{Expr}^\ell}_\omega$ is an expression that must evaluate to true for a given resource res in order for res to be able to satisfy the claim. □

During design expansion quantity expressions must be evaluated to a constant value. Formally, this is accomplished by means of a function:

$$Q_{expr \to \mathbb{Z}} : \mathbf{Expr}_{\text{qty}} \times (SYM \to \mathbb{Z} \cup \bot) \to \mathbb{Z}$$

mapping a quantity expression and a parameter/value map into an integer value. The definition of this function is straightforward and has been included in appendix B for reference.

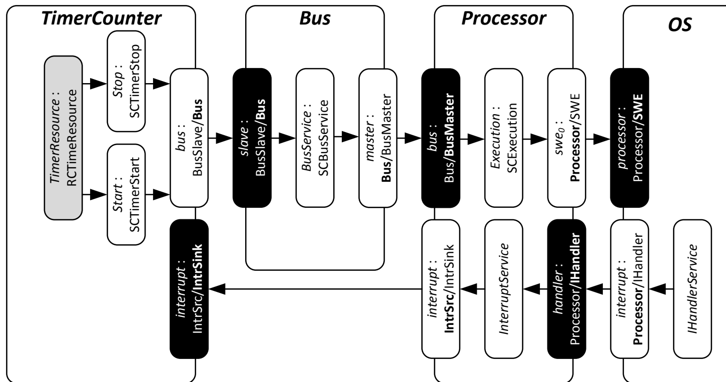


Figure 3.12 – Model of a platform demonstrating ”callbacks”.

3.3.6 Limitations

In this section, the limitations related to assertions and resource claims that has been discovered while working with them will be briefly discussed.

A common problem with both assertions and resource claims is that they are tied very closely to a single component. This means that it is not possible to declare service and/or resource dependencies for multiple components. For example, it is not possible to state that an instance of a service class must be accessible from two different components using assertions. As a consequence of this, it may sometimes be necessary to model parts of a platform or an application as a single component even though it may be more naturally modelled using a set of components. This is, for instance, the case for models of process networks. It seems natural to model the processes of a process network as a set of components representing the individual processes but because the processes share service and resource requirements we are forced to model the entire process network as a single component.

Also, the alpha expressions of assertions and the where expressions of resource claims can only be used to express constraints on the service flow in their parent component. One consequence of this is that ”callback” scenarios cannot be expressed. The example with the two operating system requiring exclusive access to timer resources is an example of this. Figure 3.12 shows a model of a platform containing a timer/counter, a bus, a processor and an operating system. In comparison with the previous examples focused on the dependencies between operating systems and timers, this model also captures the interrupt sub-system that associates an interrupt handler in the operating

system with an interrupt source on the timer/counter. Looking at the model, we may conclude that the timer resource provided by the timer/counter is an acceptable candidate for satisfying the requirement of the operating system because the timer is accessible for the processor through the necessary services and because the service representing the interrupt handler is available at the interface representing the interrupt port of the timer/counter. The last requirement, that the interrupt handler service must be available at the interrupt port of the timer/counter, cannot be expressed using an assertion or a resource claim associated with the operating system component because it involves the service flow of another component.

The obvious solution to both problems is to somehow extend the alpha and where expressions so that they can be used to express constraints on the service flow taking both the service flow and structure of other components into account. Exactly what this extension will look like is an open question and has been left for future work. It must noted that the analysis information provided by the Service Relation Model and its associated analysis method does support the necessary reasoning and that the shortcomings are primarily due to the limited expressibility of assertions and resource claims.

The last of the limitations that needs to be discussed has to do with the ability of the Service Relation Model to capture a particular allocation of resources. The presented concepts does not allow for explicitly declaring that a particular resource claim is satisfied by a particular resource and, as a consequence, the model is not very well suited for expressing allocations of resources to claims. A particular allocation can be represented implicitly by placing the resources of a model in resource classes of their own and then referring to these new classes in the resource claims. This is obviously not a pretty solution and a better solution would be to extend the concept of a resource claim so that the user may specify a named resource that *must* satisfy a given claim as part of the instantiation of the component containing the claim. By introducing this extension some of the procedures presented later in this thesis must be updated accordingly. We see no principal difficulties in doing this. The reason why this has not been done is that we have had no imminent need for it.

3.4 The `xSRM` Framework

The concepts of the Service Relation Model have been implemented as part of a proof-of-concept framework called the `xSRM` framework. Figure 3.13 shows an overview of the framework. The core of the framework consists of three different intermediate representations (or object models) for representing libraries,

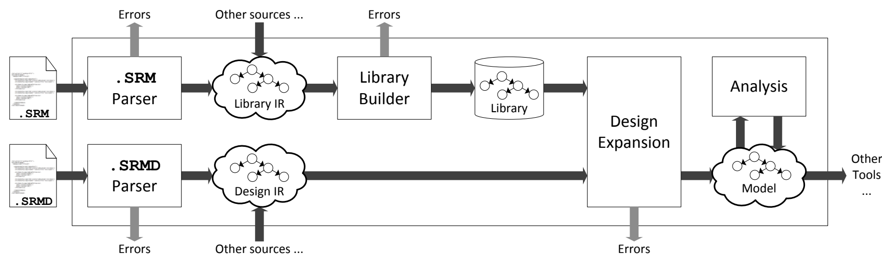


Figure 3.13 – Overview of the xSRM framework.

designs and model. The two central tasks performed by the framework is *library building* and *design expansion*. Library building is the process of combining library entity descriptions into an actual library that can later be used by the design expansion process. The design expansion process, previously described in section 2.3.3, takes as input a design and a library and generates a model. A central back-end tool is the analysis tool that is used to compute the service availablity information of a model. This tool provides an implementation of the alternative analysis algorithm previously described in section 2.4.

Even though the framework is presented as a process in Figure 3.13, the different parts of the framework are in fact semi-independent and can be used differently by different tools.

The implementation of the concepts in the xSRM framework differs slightly from the presentation in this thesis. The most notable difference between the two is that many of the aspects that has to do with configuration (e.g. options, parameters, quantity expressions) are also part of the representation of models and are not striped from the model as part of the design expansion process. This complicates models in the framework a great deal but also allow for a tools to more easily reason about the configurable parts of models. In chapter 5, a tool that exploits this information is presented.

The size of the core framework is about 14k lines of code excluding components and blank lines. A large portion of the code deals with detecting and reporting errors. The size and complexity of some of the designs that we have been working on makes good and precise error messages a must. In addition, a number of tools to assist debugging have also been developed. One of the more useful ones is a tool for dumping a service flow graph of a model as a graph using the GraphViz package and another one is a tool for dumping the information contained in a model as a HTML page with hyper links – being able to easily browse the information embedded within the different object representations of

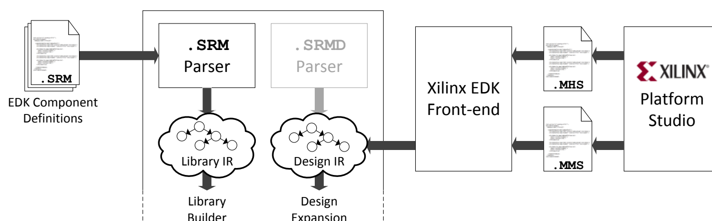


Figure 3.14 – The Xilinx EDK front-end to the `xSRM` framework.

the framework has been crucial for debugging purposes.

The framework itself has been written in C# for .NET 3.0. In restrospect, it is clear that the entire project would have benefited tremendously from using another higher-level implementation platform for doing the initial experimentation. An early attempt at implementing the Service Relation Model as a (graphical) domain specific language using the *DSL tools for Visual Studio* [23] had to be abandoned. The reason for this was that, at that time, the DSL tools only had very limited support for moving data from one language to another. This was needed to make a proper implementation of component instantiation based on re-usable component templates. While it was definitely technically possible to extend the basic framework of the DSL tools to support this, it quickly became apparent that the complexity of such an extension would be well beyond the scope of the project. An implementation based on UML or GME [61] may have proved to be more flexible but has not been attempted.

In the following, we will briefly present a number of extensions to the core `xSRM` framework that have been developed for ding experimentation.

3.4.1 The `xSRM` Front-end

Besides the intermediate representations and the library building an design expansion tasks, the core framework also contains two XML parsers for importing library and design descriptions given in the `xSRM` language. As previously mentioned, the `xSRM` language is an XML-based language for describing library entities and designs that closely resembles the less verbose `sSRM` language used by the examples in this thesis.

Component	Short Description
bram_block	Block RAM Memory (BRAM)
fsl_v20	Fast simplex link (FSL), hardware FIFO
lmb_v10	Local memory bus (LMB)
lmb_bram_if_cntlr	LMB-to-BRAM interface controller
mdm	Microblaze debug module
microblaze	32-bit RISC processor core
mpmc	Multi-port memory controller
plb_v46	Processor local bus (PLB)
plbv46_plbv46_bridge	PLB-to-PLB bridge
xilkernel	Real-time operating system
xps_bram_if_cntrl	PLB/BRAM interface controller
xps_gpio	General purpose IO controller
xps_intc	Interrupt controller
xps_mch_enc	External memory controller
xps_mutex	Hardware mutex
xps_Timer	Timer/counter
xps_uartlite	Universal asynchronous receiver/transmitter (UART)

Table 3.3 – Components of the EDK recognized by the Xilinx EDK front-end and characterized in the Xilinx EDK library package.

3.4.2 The Xilinx EDK Front-end

The Xilinx EDK front-end to the xSRM framework is capable of generating Service Relation Model descriptions of hardware/software platforms created using the EDK. An overview of the front-end is shown in Figure 3.14. The front-end parses the MHS and MSS files of an EDK projects and produces a corresponding design. Besides the MHS/MSS parser, the front-end also consists of a library of descriptions of the components in the EDK. Using the xSRM framework the design and the components of the library can be combined into a service relation model. In most uses of the front-end, the design representing the EDK platform is combined with an application of sorts before expanding the result into a model.

Table 3.3 shows a list of the components supported by the front-end. With the exception of the Xilkernel component, all of the components are hardware components. The EDK contains significantly more components than are currently supported by the front-end. Most of these components, however, are quite specialized and are not supported simply because they have not been needed. Adding support for a new component consists of creating a description of it in the xSRM language and adding an entry in the parser for the MHS and/or MMS files.

3.4.3 The YML Front-end

To support working with process networks, a front-end for importing models of process networks given in the XML-based *Y-chart modeling language* (YML) has been developed.

3.5 Consistency Checking

In this section, we will present a method for computing the consistency of a service relation model based on the concepts previously introduced in this chapter.

A service relation model is said to be *consistent* if and only if **1)** all assertions are true and **2)** all resource claims can be satisfied and otherwise it is said to be *inconsistent*.

The method for checking the consistency of a model exploits the fact that the concepts of resources and assertions are independent of each other to provide a fast and inexpensive consistency check that can be integrated into compilation/system generation loops without affecting the productivity of the designer notably. The two concepts are independent in the sense that the truth the assertions in a given model does not depend on the truth (i.e. satisfiability) of the resources and vice versa and, as a consequence, we may treat them independently of each other.

The consistency checking procedure presented is focused exclusively on determining the availablity or absence of services and resources. This kind of consistency check is by itself not enough to ensure that the platform or system in question is free or errors and is intended to be used in combination with other, more common, checks. Checking that each component has the necessary services and resources available is, however, an important aspect of component-based design that has hugely been left unaddressed in the context of component-framework-free component models.

3.5.1 Assertion Checking

Checking that the assertions of a model are satisfied is simply a question of evaluating the alpha expression of each assertion. The only prerequisite for doing this, is that the service availability information SA_m of the model has

been computed beforehand. Formally, a model m is consistent with respect to assertions if and only if:

$$\bigwedge_{a \in A_*} (Eval_\alpha(a.e, SA_m))$$

where A_* is the set of assertions in m and $Eval_\alpha$ is the function for evaluating an alpha expression. The complexity of the check is linear in the number of terms in the assertion expressions.

3.5.2 Resource Claim Checking

In this section, a simple procedure for checking the consistency of a model with respect to resources and resource claims is presented. The procedure consists of three steps and uses an *satisfiability modulo theories* (SMT) solver supporting the theory of linear arithmetics for deciding whether or not a given model is consistent with respect to resources and resource claims.

3.5.2.1 Satisfiability Modulo Theories (SMT) Solvers & Problems

An SMT solver is a solver capable of evaluating the satisfiability of logical formulae of binary-valued predicates over non-binary valued variables. The type of variables and expressions that can be used with a solver is determined by the *theories* that it supports. The basic theory, supported by all solvers, is first order logic. Most, if not all, solvers supports some variant of linear arithmetics. Some solvers also supports more exotic theories such as bit vectors and uninterpreted functions. Notable examples of SMT solvers include the HySAT solver from Oldenburg University [35], the Z3 solver from Microsoft Research [29] and the Yices solver developed by SRI international [32].

An SMT problem, the input to an SMT solver, consists of a number of assertions. An assertion is a logical formula of binary-valued predicates. An SMT problem is said to be *satisfiable* if there exists a valuation of its variables so that all assertions evaluates to true and otherwise the problem is said to be *unsatisfiable*.

3.5.2.2 The Procedure

The problem addressed by this procedure can be summarized as follows: Given the set $RES_* = \{res_0, res_1, \ldots, res_i\}$ of all resources and the set $CL_* = \{cl_0, cl_1, \ldots, cl_j\}$

of all resource claims in a model m, is it possible to satisfy all claims in CL_* with the resources available in RES_* taking the service flow connecting the claims with the resources into account?

Step 1 (Preparation) The first step of the procedure consists of computing the set of so called *possible matches* between the resources and resource claims in the model. The set of possible matches is used by the next step in the procedure for simplifying the encoding of the problem into an SMT problem.

Definition 3.16 (Possible Match, pm, PM) A possible match pm is a pair $pm = \langle res, cl \rangle$ where $res \in RES_*$ and $cl \in CL_*$ for which the following two conditions are true: 1) the provided resource is a sub-class of the claimed resource class:

$$res.rc \sqsubseteq cl.rc \tag{3.1}$$

and 2) the where expression of cl evaluates to true for the resource res:

$$\Omega_e(cl.\omega, res, SA_m) = true \tag{3.2}$$

Here SA_m is the service availablity information of m and Ω_e is the function for evaluating where expressions defined previously. □

Computing the set of all possible matches for a model can be done by simply evaluating the two conditions for all pairs of resources and resource claims in the model. If the model contains a large number of resources and resource claims computing this set can be computationally expensive. Notice that quantities are not considered when determining if a resource and a resource claim constitutes a possible match. An additional check that the quantity of resource provided by *res* is larger or equal to the quantity of resource being claimed could be added but is not needed.

In the following, let $PM = \{pm_0, pm_1, \ldots, pm_n\}$ be the set of all possible matches, $P_{res} : CL_* \rightarrow \mathcal{P}(RES_*)$ be a function that maps claims into the set of resources that may satisfy the claim and $P_{claim} : RES_* \rightarrow \mathcal{P}(CL_*)$ a function that, given a resource, returns the set of claims that the resource may satisfy.

Step 2 (Encoding) Given the set of possible matches computed in the previous step, the problem has been reduced to a simpler resource allocation problem because the part involving the service flow has been eliminated.

The encoding is based on the use of a set of integer variables for modeling the allocation of resource to resource claims. There is one such variable for each possible match. To simplify the presentation, we will use a map-notation to refer to these variables where $\vec{u}[res, cl]$ gives us the variable representing the amount of resource from *res* allocated to claim *cl*. Since a negative amount of resource cannot be allocated we implicitly require that $\vec{u}[res, cl] \geq 0$. The encoding consists of an assertion of the form:

$$\textbf{assert} \left(\bigwedge_{res \in RES_*} \mathcal{E}_{res}(res) \right) \wedge \left(\bigwedge_{cl \in CL_*} \mathcal{E}_{cl}(cl) \right)$$

where $\mathcal{E}_{res}$ is a function for encoding the contribution of a resource and $\mathcal{E}_{cl}$ a function for encoding the contribution of a resource claim.

Resource Encoding. The restrictions placed on each of the resources are given in equation 3.3. For each resource, we simply require that the sum of all resource allocated to claims does not exceed the quantity of resource available:

$$\mathcal{E}_{res}(res) := res.q \leq \sum_{cl \in P_{claim}(res)} \vec{u}[res, cl] \tag{3.3}$$

Claim Encoding. Claims are encoded differently depending on whether the claim can be satisfied by multiple resources $mp = many$ or must be satisfied by a single resource $mp = one$:

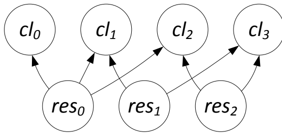
$$\mathcal{E}_{cl}(cl) := \begin{cases} \mathcal{E}_{cl,m}(cl) & \text{if } cl.mp = many \\ \mathcal{E}_{cl,s}(cl) & \text{if } cl.mp = one \end{cases} \tag{3.4}$$

For claims that may be satisfied by multiple resources we require that the sum of all resources allocated to the claim equals the quantity of resource claimed:

$$\mathcal{E}_{cl,m}(cl) := cl.q = \sum_{res \in P_{res}(cl)} \vec{u}[res, cl] \tag{3.5}$$

The encoding for claims that must be satisfied by a single resource is slightly more complicated:

$$\mathcal{E}_{cl,s}(cl) := \bigoplus_{res \in P_{res}(cl)} \left(cl.q = \vec{u}[res, cl] \wedge \bigwedge_{z \in P_{res}(cl) \setminus res} (\vec{u}[z, cl] = 0) \right) \tag{3.6}$$



$$res_0.q = 10 \quad cl_0.q = 10$$
$$res_1.q = 10 \quad cl_1.q = 10$$
$$res_2.q = 10 \quad cl_2.q = 10$$
$$cl_3.q = 10$$

$$\begin{aligned}
\vec{u}[res_0, cl_0] + \vec{u}[res_0, cl_1] + \vec{u}[res_0, cl_2] \leq res_0.q \quad &\wedge \\
\vec{u}[res_1, cl_1] + \vec{u}[res_1, cl_3] \leq res_1.q \quad &\wedge \\
\vec{u}[res_2, cl_2] + \vec{u}[res_2, cl_3] \leq res_2.q \quad &\wedge \\
\vec{u}[res_0, cl_0] = cl_0.q \quad &\wedge \\
\vec{u}[res_0, cl_1] + \vec{u}[res_1, cl_1] = cl_1.q \quad &\wedge \\
\vec{u}[res_0, cl_2] + \vec{u}[res_2, cl_2] = cl_2.q \quad &\wedge \\
((cl_3.q = \vec{u}[res_1, cl_3] \wedge 0 = \vec{u}[res_2, cl_3]) \vee (0 = \vec{u}[res_1, cl_3] \wedge cl_1.q = \vec{u}[res_2, cl_3]))&
\end{aligned}$$

Figure 3.15 – A simple resource allocation problem.

Example 3.11 Figure 3.15 shows an example of an encoding of a simple resource allocation problem with three resources (res_0, res_1, res_2) and four claims (cl_0,cl_1, cl_2 cl_3). The service flow connecting the resources with the resource claims is shown using a graph and the quantity of resource provided and claims is given next to the graph. Claim cl_0 can only be satisfied by resource res_0, claim cl_1 can be satisfied by either res_0 or res_1, claim cl_2 by res_0 and res_1 and, finally, claim cl_3 by res_1 or res_2. Claims cl_0, cl_1 and cl_2 all have single multiplicity meaning that they must be satisfied by a single resource. Claim cl_3 has multiplicity many meaning that it can be satisfied by a combination of the resources. ■

Step 3 (Solving & Decoding) The encoded SMT problem is fed to an SMT solver that will evaluate its satisfiability. If the problem is found to be satisfiable then the input model is consistent with respect to resources and resource claims. If the problem is found to be unsatisfiable then at least one claim could not be satisfied and thus the model is inconsistent.

In the case of a problem being satisfiable, the solver can also provide a satisfiable valuation (called a model) of the free variables in the problem. Given such a valuation, it is straightforward to decode an allocation of resources to resource claims. It is important to note that such an allocation of resources to resource claims is not (necessarily) optimal in any regard. For consistency checking, however, the valuation of the variables are of no interest.

3.5.2.3 Error Reporting

If a model is determined to be inconsistent then the user would like to know what part of the model that causes the inconsistency. For resources, providing precise error reports is troublesome because no information besides ”unsatisfiable” will be returned by the solver. For small models with few resource claims, the user may be able to determine the problem but for larger models determining the problematic claims will prove difficult. One way to improve on the error reports is to split the resource problem into smaller problems. A resource problem can be split if the resource claims can be divided into two or more subsets so that the resource classes claimed by each subset does not overlap with the resource classes claimed by the other subsets. Each such subset will constitute a resource problem of its own and its satisfiability will not be dependent on the resources belonging to the classes claimed by the resource claims in the other subsets. Any inconsistencies in the model will show up as one or more resource problems being unsatisfiable. For each unsatisfiable problem an error report containing only a subset of the resources and resource claims of the model can be generated. Besides enabling more precise error reports, splitting the problem into smaller problems will also have a positive effect on the execution time of the solver.

3.5.3 Consistency Checking in the `xSRM` Framework

The presented procedure for checking the consistency of a model has been implemented as part of the `xSRM` framework in the form of a back-end extension. The implementation takes as input a model and produces a true/false answer to whether or not the model has been determined to be consistent. The Yices SMT solver [32] is used for solving the SMT problem associated with checking the satisfiability of resource claims. Yices was chosen because it, in comparison with many other solvers, offers an API that allowed for the solver to be integrated into the framework and, more importantly, because it can also be used for solving MAXSMT problems and supports the background theory of bit vectors. These last two features of Yices are not used by the procedure for consistency checking but by the two tools presented later in chapter 4 and 5.

3.5.4 Experiments & Results

To determine how large models that can be handled efficiently by the consistency checking procedure an experiment using a set of similar platforms of dif-

	Number of sub systems (n)				
	n = **10**	n = **100**	n = **200**	n = **300**	n = **400**
Service flow graph size					
Number of nodes	271	2701	5401	8101	10801
Number of edges	220	2200	4400	6600	8800
Measured execution time					
Flow graph construction	~0.00 s	0.02 s	0.04 s	0.09 s	0.03 s
Service availablity analysis	0.02 s	2.70 s	20.75 s	69.99 s	163.85 s
Checking assertions	~0.00 s	0.01 s	0.04 s	0.07 s	0,13 s
Compute possible pairs	~0.00 s	0.42 s	2.78 s	9.11 s	20.80 s
Construct SMT problem	~0.00 s	0.61 s	11.15 s	39.32 s	151.54 s
Solve SMT problem	~0.00 s	0.02 s	0.03 s	0.11 s	0.27 s

Table 3.4 – Measured exeuction time of consistency check and size of the service flow graphs for varying number of sub systems.

ferent sizes has been carried out. The point of the experiment is to show that the consistency checking procedure is suitable in a practical scenario.

3.5.4.1 Setup

Figure 3.16 shows the topology of the test platforms created using the Service Relation Model descriptions of the components of the Xilinx EDK that are part of the Xilinx EDK front-end to the `xSRM` framework. The platform consists of a single interconnect in the form of a processor local bus and a variable number of sub- systems. A sub-system, also shown in figure 3.16, consists of a Microblaze processor, a local BRAM block memory, a timer/counter and a Xilkernel operating system. The Microblaze implements the Harvard architecture and, consequently, it has different interfaces for instructions and data. The local BRAM memory is a dual port memory connected to the processor through two local memory buses ((`ilmb` and `dlmb`) and two memory controllers ((`icntrl` and `dcntlr`) – this is the standard way of connecting the Microblaze core with block ram memory in the Xilinx workflow. Each sub system has 1 assertion, 1 resource and 1 resource claim.

Using the `xSRM` framework, a test program has been written for measuring the execution time of the various tasks involved in consistency checking. By varying the number of sub systems, we can vary the size of the service flow graph and the number of assertions, resources and resource claims. All models created using the test program are consistent. The assertion is satisfied internally by the sub systems themselves and the claim for a timer resource is satisfied by the timer through the shared bus.

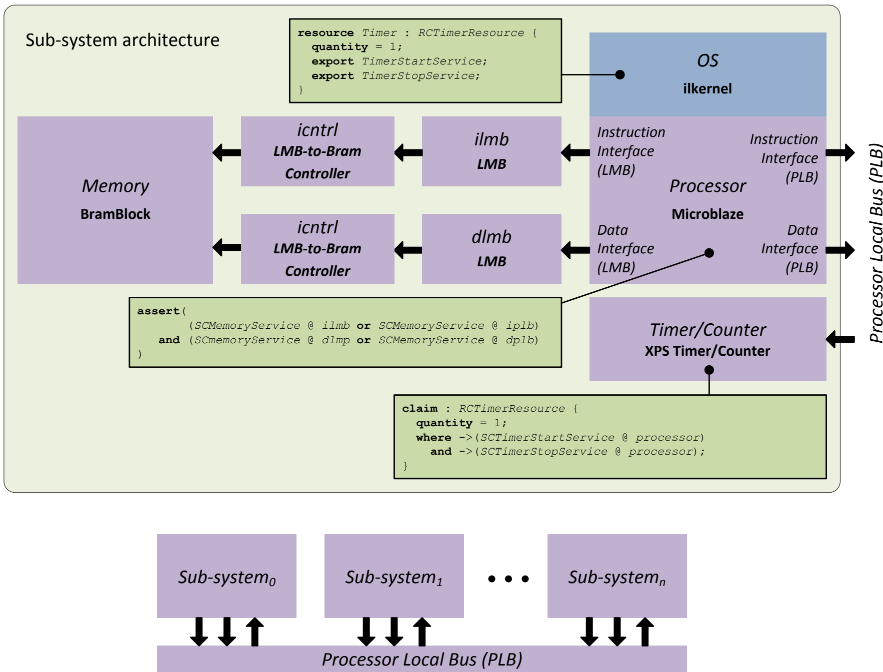


Figure 3.16 – Schematics of the topology of the test platforms and architecture of the sub systems.

3.5.4.2 Results

Table 3.4 shows the measured execution time for five different experiments with $n = 10, n = 100, n = 200, n = 300$ and $n = 400$. The experiments have been run on an Intel Core2 Quad CPU at 2.40 GHz running Microsoft Windows 7. Even though a quad core CPU was used for the test, the program and the solver only uses a single core. The results for $n = 10$ are obviously inaccurate. This is due to the resolution of the timer. For $n = 500$ Yices crashes the test program for unknown reasons. The most obvious explanation would be that the number of variables exceeds some internal limit. With $n = 500$ there are a total of 25000 variables in the problem.

Measuring the time required by Yices for solving the resource allocation problem is slightly problematic. The problem is that part of the solving is done when SMT assertions are added to the logical context of Yices. As table 3.4 show, the time used for actually solving the allocation problem is insignificant for all the experiments compared to the time spend constructing the problem

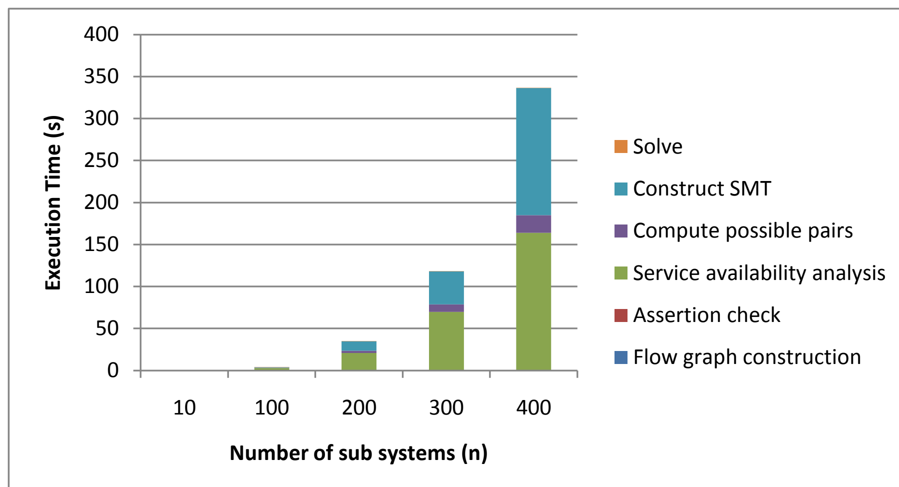


Figure 3.17 – Graphical representation of the results from table 3.4.

(i.e. constructing and adding SMT assertions).

Figure 3.17 shows a graphical representation of the data in Table 3.4. As expected, the figure clearly shows that the execution time of the procedure scales exponentially with the size of the input model. As can be seen, the service availablity analysis, computation of the possible pairs and, especially, the construction (and solving) of the SMT problem dominates the picture for larger n.

The experiment shows that the consistency checking procedure is capable of handling large problems in reasonable time even though the exponential growth in execution time means that the procedure scales poorly. The ∼2 minutes required for checking the platform with $n = 200$ and the ∼5 minutes and 30 seconds required for checking the platform with $n = 400$ may sound like a long time. Whether or not this is case ultimately depends on the context in which the procedure is used. If, for example, the procedure is used in combination with a synthesis tool for hardware synthesis then the time required for running the procedure will most likely be negligible compared to the time used for synthesis. The synthesis of a dual processor design with the Xilinx design flow can easily take up to 10 minutes or more. The synthesis of a design with 100+ cores is likely to take a *very* long time.

3.5.4.3 Optimizations

Even though the procedure may perform sufficiently well to be used in a wide range of practical scenarios it would still be desirable to optimize it further. In this section, some optimizations to the tool used for the experimentation will be discussed. This discussion will focus on the three most time consuming tasks: the service availablity analysis, the computing of the set of possible pairs and the construction and solving of the SMT problem.

Service Availablity Analysis. A number of things can be done to improve the performance of the service availablity analysis. As previously mentioned, a lot of research has been devoted to improving algorithms for computing the transitive closure of graphs. The algorithm used by the `xSRM` framework represents one of the simplest algorithms available for the task. Other, possibly more complicated, algorithms are likely to perform better.

Using a profiler tool, we have been able to determine the bottleneck in the analysis back-end of the `xSRM` framework to be the implementation of the ”is subset of” operation used in the inner loop of the algorithm. The implementation uses the `BitArray` data structure that is part of the Microsoft .NET common runtime library (CLR). For unknown reasons, this data structure does not support a method for testing if all bits are either zero or false and, as a consequence, any comparisons must be done by iteratively comparison of the individual bits. It is possible to do a more efficient implementation of comparison for the special cases of all bits being 0 or 1 but it will require access to the internal representation used by the `BitArray` class. Using another underlying representation supporting this will most likely result in a significant speed up.

Computing Possible Pairs. In the implementation, no attempt has been done at optimizing the computation of the possible satisfying pairs. The implementation itself is rather straightforward and simply compares every resource in a model with every resource claim in the same model. For each pair of resources and claims, the resource classes are first compared and if they are compatible then the where expression of the claim is evaluated. An obvious way of speeding up the computation is to exploit the obvious parallelism in the problem.

SMT Construction & Solving. As such, little can be done to improve the performance of the solver since it is used as black box in the test program. It may be possible to ”optimize” the encoding or to tweek some of the parameters of

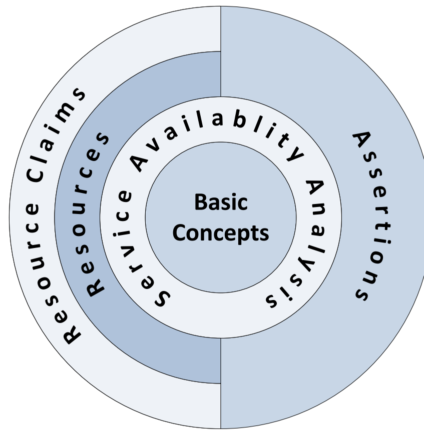


Figure 3.18 – Dependencies between concepts of the Service Relation Model.

the solver to improve its performance for the specific type of problem. The gain in performance for doing this will most likely not be impressive.

3.5.5 Discussion

An important property of the Service Relation Model is that there are no cyclic dependencies between service flow, service availablity information and resource allocation and assertions. Service availablity information is a function of the service flow in a model and service flow is, by definition, static and independent of the service availablity information, the the satisfiability of resource claims and the truth of assertions. The satisfiability of resource claims is dependent on service availablity information and by implication also on service flow but independent of the truth of assertions. Finally, assertions are dependent on service flow and the service availablity information but not on the satisfiability of resource claims. Figure 3.18 illustrates these dependencies between the different concepts of the Service Relation Model.

This *separation of concerns* means that the task of evaluating the consistency of a model can be done in three steps: 1) compute service availablity information, 2) compute resource allocation and 3) evaluate assertions. Because step 2 and 3 are independent their may be switched or done in parallel. The complexity and the computational requirements of each of these steps are, as we shall see, not overwhelming. However, this separation comes at the price of reduced expressibility. Most problematic is the requirement that service flow

must be static and independent of service availability information and resource allocation. This means that we cannot model situations where services are conditionally provided – e.g. a component may provide a given service only if it has access to some other set of services or resources. If we were to allow service flow to be a function of resource allocation then we will need to consider both simultaneously when testing for consistency. Because computing the satisfiability of the resource allocation is in the NP domain the combined problem will also be in the NP domain. This will have a major impact on the practical feasibility of the consistency checking procedure because the resulting problem will be much larger which means that finding a solution may take significantly longer.

3.6 Related Work

In this section, we will briefly review related work. The presentation of related work has been postponed until now because we believe that a basic understanding of the basic concepts of the Service Relation Model will be needed to properly position this work with respect to the literature.

3.6.1 System Level Design Languages

The SystemC language [50] is the De facto standard for modeling at the system-level of abstraction. SystemC is an extension of C++ in the form of a set of libraries. The core of SystemC is an event driven simulator that enables co-simulation of hardware described using the SystemC C++ extensions and ”native” C++ code. The mix of C++ and hardware modeling capabilities is both SystemC's strength and its weakness. A practical problem with the use of SystemC, and other languages based on C++, is that designers must worry about the syntactic and semantic details of the underlying C++ language that are independent of the system model. Also, tools for analyzing SystemC designs are difficult to write because of the complexity of the C++ language.

The Metropolis framework is an embodiment of the platform-based design paradigm [13]. The framework is based on a model called the *Metropolis Meta-Model* that can be used to capture both application and platform. The execution semantics of the model is based on events. Models consists of a composition of processes, media, quantity managers and netlists. A process has its own thread of execution that executes concurrently with the other processes in the system. Communication between processes is accomplished by means of media objects

that implements a set of interface methods. Medias can be connected to processes and other media by means of ports. Quantity managers are used to control accesses to shared media and to assign physical quantities to events. Netlists are container objects used to structure a model into a hierarchy. Besides these basic building blocks the model also supports user-defined constraints using *linear temporal logic* and *logic of constraints*. Metropolis is primarily a composition framework supported by tools for simulation and verification. A subset fo the model may, however, be synthesized using the *xPilot* synthesis system. Since its interception, the team behind Metropolis has identified a number of shortcoming of the framework. A next-generation framework, named *Metro II*, will address these shortcomings [26, 25]. A noticeable problem with the current Metropolis framework is its inability to import and work with pre-defined IP.

Although not explicitly represented, the Metropolis framework (or rather the platform-based design paradigm) incorporates a concept of services somewhat similar to the concept of a service in the Service Relation Model. Metropolis does not have a concept similar to service aggregation and, as a consequence, cannot be used to explicitly describe the flow of service as function of the descriptions of the components. A large part of the Metropolis framework is focused on the estimation of non-functional properties like power usage and performance. Because the similarities between the concept of a service employed by the Service Relation Model and in Metropolis we believe that their ideas for estimation of non-functional properties can also be applied to the Service Relation Model.

Other examples of system-level design languages include SpecC [30] and SystemVerilog [7]. All though all of the languages mentioned here supports modualization neither of them are proper component-models as they lack the means to express dependencies of components and do not distinguish between a components interface and its implementation.

3.6.2 Component-based Design (Hardware)

Bottoms-up approaches to embedded systems design have received a lot of attention from researchers and have shown promising results. In a bottoms up approach, a design is assembled from a set of predefined components. Obviously, if the components are small, such as logical gates or processor instructions, this would not be a challenge. This kind of trivial reuse has a long tradition within both hardware and software design. The challenge is to enable the reuse of complex components such as processors, buses and operating systems. In the field of hardware, the term ”IP-based design” is often used in place of ”component-based design”.

3.6.2.1 Standardized Component Frameworks

Most component-based design approaches are based on the use of standardized component frameworks. Such a framework usually consists of an standard interface used to connect (hardware) components and/or a standard communication infrastructure. Examples of such frameworks include the OpenConnect infrastructure from IBM [24] and the AMBA protocol from ARM [10]. An intrinsic problem with using a specific infrastructure as the foundation of a design approach is that may only be used with components (cores) that implements a compatible interface. The Xilinx EDK [97] and the Altera SoPC [9] are examples of tools that are focused on a set of specific interconnects.

3.6.2.2 Virtual Architecture Models

The Coral [15] framework from IBM is an early attempt at moving the specification of platforms up in abstraction from the register transfer level. This work was one of the first to introduce the concept of a virtual design – an idea that has later been elaborated on by other researchers.

As part of their ongoing research into system-level design methodologies, Jerraya et al. proposes a design flow where an application is decomposed into a set of virtual components connected through a virtual interconnect [19]. A high-level parallel programming model is used to de-couple these components from each other and to abstract away platform details. Much of their effort has been spent trying to automate the generation of wrappers used to connect components with an actual interconnect.

Interestingly, they use a model called the *Service Dependency Graph* to synthesize wrappers between components [60, 84]. Despite the name, the service dependency graph model has little in common with the Service Relation Model presented in this thesis. The Service Dependency Graph is used to capture service requirements and dependencies amongst the sub-components of an interface connecting an implementation of a virtual component to an actual interconnect. The Service Relation Model, on the other hand, is used to capture the flow and availability of services in an entire system composed of components. The difference between the two will be elaborated on later in section 3.7 as part of a more general discussion.

3.6.2.3 Other

Balboa [31] is a component composition framework based on C++ used to build system-level models with an architectural perspective. The framework consists of thee parts: a component integration language, a set of IP component libraries and a set of split-level interfaces used to link between the two. Component design is done using C++ where as component integration is done using a scripting language front-end. The key feature of Balboa is that it raises the abstraction level by using type inference to allow for weaker type dependencies among components than what is possible with the C++ components alone. This allows for the designer to focus on architectural design rather than C++ type matching.

3.6.3 Component-based Design (Software)

The literature contains many examples of models incorporating the concepts of components and/or services. These concepts so are general and used in many different contexts with different meanings that a complete survey is out of the scope of this work.

3.6.3.1 Component Models & Component-based Design

In software engineering, the term ”component-model” is often used as a label for specific technologies supporting component-based design of software systems. In this context, a *component model* is a set of standards and conversions that components must follow to allow proper integration. A component model is associated with a *component framework* – the infrastructure that enables component interaction and manages resources for components [56].

Within the field of enterprise system component-based design and component models have received a lot of attention from major players such as Microsoft, IBM, Intel and Sun Microsystems. Component-based design is used to bring structure to the design of large systems by allowing the development of a software system to be split between three independent groups: component designers, infrastructure (component framework) designers and application designers. Common component-models includes JavaBeans [2, 3], the .NET component model and the COM/DCOM model [48]. Component technology has been applied to the design of embedded systems software with success. The Koala component-model developed at Philips for designing media equipment is

a prime example [95]. Other examples include PECOS [40] and Rubus [72] component models.

A particularly interesting class of component-based design methodologies for enterprise systems is service oriented computing and architecture. Service oriented computing is a paradigm referring to a set of concepts, principles and methods that represent computing in service-oriented architecture (SOA). In such architectures, functionality is viewed as services exposed through service brokers (e.g. CORBA [49]). SOA emphasizes loose-coupling of services with each other and with the underlying platform. From an embedded systems point of view, this loose-coupling is problematic as it implies a rather large overhead, in the form of a service broker framework, that is generally not acceptable. Also, SOA services are too "heavy weight" to be useful in many embedded applications. Despite these problems, SOA has also been employed in the construction of embedded systems [93].

3.6.3.2 Architecture Description Languages

In the software field, an Architecture Description Language (ADL) is a language used to capture design decisions regarding the architecture of a system. There is no commonly agreed upon definition of what exactly constitutes an ADL [65] but according to [21] an ADL is a language capable of expressing the software architecture of a system in a way that suppresses implementation and non-architectural information. Furthermore, an ADL must embody rules about what constitutes a consistent architecture and provide some analysis or code-generation capabilities. The literature contains many examples of ADL's that can be used to analyze various properties of systems composed of components (e.g. Acme [39]). Some ADL's also take hardware (platforms) into consideration to analyze non-functional properties like real-time schedulability and reliability ([16, 8]).

The Service Relation Model could be considered an ADL even though its intended purpose is different from that of ADL's. The Service Relation Model is not intended as a software engineering tool and it does not have much to offer in that regard.

3.6.3.3 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a set of graphical general purpose languages used for object-oriented modeling standardized and managed by the

Object Management Group [17]. UML is not by itself a methodology but may be used in a variety of methodologies for different purposes. A key feature of UML is that it can be extended with domain specific concepts using, so called, *UML profiles*. A number of methodologies for addressing various problems within the field of embedded systems have been proposed using the basic UML infrastructure extended with domain specific concepts in the form of UML profiles.

For example, in [20] a design methodology based on UML, Metropolis and the principles of platform-based design is proposed. The methodology is based on the use of an UML profile for extending UML with a number of concepts from Metropolis and platform-based design. The profile combines the strengths of UML and the Metropolis framework by allowing UML models to be transformed into Metropolis models. Metropolis can thus be used as a back-end for UML for simulation and synthesis purposes. A number of, conceptually similar, UML profiles for linking UML and SystemC have also been proposed (e.g. [96, 67, 81]).

The concepts of the Service Relation Model could most certainly be expressed in UML as a profile. This is not the same as saying that the two are the same. Being a general meta-modeling framework UML is capable of expressing most component-models.

3.6.4 Comparison with the Service Relation Model

There are two aspects that sets the Service Relation Model apart from most of the different component-models mentioned.

One thing that sets the Service Relation Model apart from most other component models is the fact that it does not incorporate the concept of a component framework. This is an essential difference as the existence of a component framework implies that only components compatible with the framework can be used – a key feature of the Service Relation Model is that it does not impose any restrictions on what a component is.

The second difference has to do with the purpose of the model and how dependencies between components are represented. Many component-models provide the means to check the consistency of inter-component dependencies. Often such dependencies amongst components are modeled by means of the concepts of provide and require interfaces [60, 13, 95, 47]. A *provide interface* is an interface through which a component provides other components with one or more services and a *require interface* is an interface where a component consumes service. A model employing the concepts of provide/require interfaces

is consistent when all require interfaces are connected to compatible provide interfaces. Checking for this kind of consistency is simply a matter of inspecting each require interface to determine if it is properly connected.

A potential drawback of using these concepts for modeling dependencies among components is that the dependencies must be explicitly asserted by means of relations connecting require interfaces with provide interfaces. For example, if component a is dependent on services provided by components b and c then this must be modeled using a pair of relations for relating a with b and c. This means that the allocation of service and/or resource from providers to consumers must be available a priori to analyzing the model and, thus, the model cannot be used as a basis for determining this information. For most component-models this is not a problem because they are, in comparison with the Service Relation Model, intended primarily for representing dependencies rather than as a foundation for resolving them. This difference in purpose illustrates well the focus of the Service Relation Model as an analysis model as opposed to a representational model.

3.7 Discussion & Summary

In this chapter, we have presented a number of additional concepts to the Service Relation Model that can be used for checking the consistency of platforms and systems.

In the introduction, a number of problems specific to the Xilinx EDK tool were discussed. By incorporating the concepts of the Service Relation Model into the internal component-model of the EDK many of these shortcomings could be dealt with efficiently. More specifically, it would be possible to make the software generation tool (Libgen) independent of the components of the library and assumptions about the communication topology. This would require that Service Relation Model descriptions of the IP and software components, such as those found in the EDK front-end to the `xSRM` framework, were available. Besides describing the service flow of each components, the description would also formally capture any service/resource dependencies that the component may have. This information can be used to both improve the analysis back-ends of the Libgen tool and to improve the existing check for errors (via consistency checking) that is done prior to synthesis.

CHAPTER 4

Automated Programming

In this chapter, we will focus on the problem of automatically generating an implementation of the communication infrastructure (the channels) of a process network targeting custom execution platform instances.

4.1 Introduction

A process network application consists of a number of processes interacting using an API, acting as an abstraction layer between the platform instance and the application, exposing the process network programming model. The API is tied to the platform by means of a platform abstraction layer (PAL) that provides a realization of the API for the particular platform.

Rather than focusing on attempting to generate a platform abstraction layer for process networks in general we are interested in application-specific implementations where the generated layer supports exactly the features used by the process network in question. An application-specific implementation is more efficient because it only has to support a fixed number of processes and channels and does not have to provide the means for any two processors of the platform to communicate – which may not necessarily be possible.

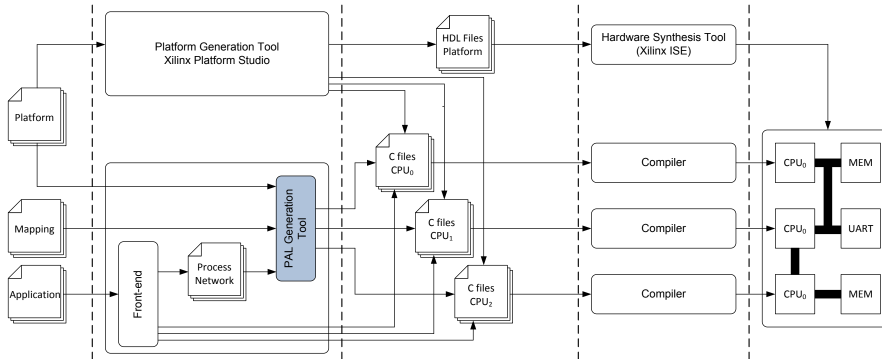


Figure 4.1 – Possible design flow supported by our procedure (PAL generation tool).

Figure 4.1 shows a design flow supported by the procedure presented in this chapter. The design is shown at three different levels of abstraction: the system-level, the register transfer level and the gate level. At the system-level the design consists of an application specification, a platform specification and a mapping between the two. The platform is a composition of components taken from a library of existing hardware and software components. The platform specification specifies which components are used, how they are configured and connected. The application is captured in a high-level language supporting the process network MoC (e.g. TTL) and, conceptually, consists of a number of concurrent processes communicating and synchronizing using unidirectional FIFO channels. Finally, the mapping consists of associations between the processes in the application and processors in the platform. Channels are not part of the mapping as determining their placement is part of the automated design flow.

The platform is refined into a register transfer level description using a platform generation tool (Xilinx Platform Studio for example). The result of running this tool on the platform specification is a set of hardware description language (HDL) files implementing the hardware and a set of C files for each programmable processor implementing the platform software. The HDL files are later processed by a hardware synthesis tool to obtain a gate-level implementation of the hardware.

Using an application translation tool and the mapping, the platform specification and the platform independent application specification are refined into a platform dependent implementation consisting of a set of C files for each programmable processor in the mapping specification. For each processor in the design, the C files generated by the application translation tool and platform generation tools for the processor can be combined and compiled to obtain the final program images.

A tool based on the procedure presented in this chapter is intended to be used as part of the application translation process embedded within the application translation tool of Figure 4.1. This tool will require three inputs: a description of the platform given in the Service Relation Model, a description of the process network embedded in the application and, finally, a mapping of the application processes onto processors in the platform. Given these inputs, the tool produces an implementation of the applications (global) communication infrastructure.

The procedure analyzes the platform description to obtain information about the different alternative implementations, called *possible implementations*, of the channels in the process network. Because the possible implementations of the channels can be dependent on one or more of the (limited) resources provided by the platform it is not possible to arbitrarily choose the actual implementations between the sets of possible implementations. Using information about the resource requirements of the different possible implementations and the resources provided by the platform, the procedure constructs and solves an optimization problem whose solution provides a feasible choice of actual implementations with respect to resources. The choice of actual implementations is fed to a code generation back-end that generates the C code implementation of the channels.

In another version of the procedure, that will not be presented here in detail, the optimization problem is replaced by a manual intervention by the designer. In this procedure, the designer selects, for each channel, an actual implementation from amongst the set of possible implementations. The feasibility of the choice of actual channel implementations is checked and if it is not consistent with respect to the resources provided by the platform (i.e. uses more resource than is available) the designer must alter his or her choices. In comparison with the procedure presented here, the alternative procedure allows for the designer, rather than the procedure, to take control of the choice of channels. Also in favor of the alternative procedure is the fact that it is less computationally expensive because it does not involve solving an optimization problem.

4.2 The Procedure

In this section, the procedure for automated programming is presented. Before presenting the procedure itself, the generic architecture of the abstraction layers generated will be presented.

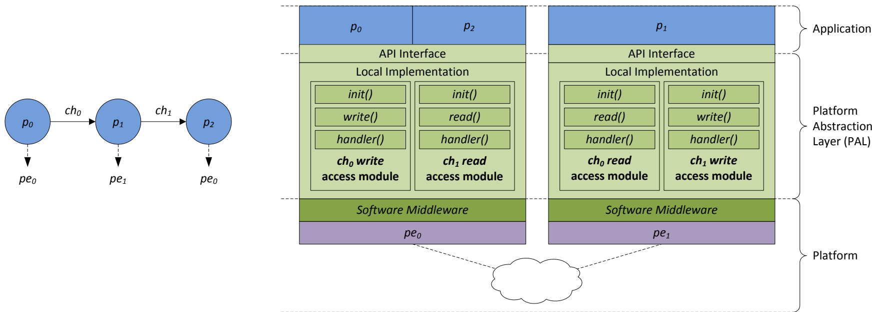


Figure 4.2 – Architecture of the generated platform abstraction layer.

4.2.1 Layer Architecture

Figure 4.2 shows an outline of the architecture of the generated platform abstraction layer produced by our procedure in the context of a simple three process application mapped to a dual-processor platform. The generated process abstraction layer (PAL) is organized as a set of *local implementations*. Each processor in the platform is associated with a single local implementation that services all application processes mapped to that processor though a simple API. Conceptually, the communication infrastructure of a process network application consists of a set of channels accessed by means of read and write primitives with blocking semantics. An implementation of a channel consists of two *channel access modules*. These two channel access modules, a read module and a write module, provide implementations of the read and write primitives associated with the channel. The read module is part of the local implementation associated with the processor where the application process reading from the channel (the consumer) is mapped. Similarly, the write module is part of the local implementation associated with the processor where the application process writing to the channel (the producer) is mapped. An access module is a logical grouping of three C functions: an initialization function, a read or write function (depending on the type of access module) and a handler function.

The initialization functions of the access modules allows for some initialization of the channels to take place before the system begins its operation. At system start-up, each local implementation will call the initialization functions of its access modules and once the inititialization has been completed the system will begin its operation. Determining when the initialization phase is completed can be a problem. Here we assume that all processors agree to wait for a set time before beginning their operations. The set time is long enough to allow all processors to do their initialization. We will not address the problem

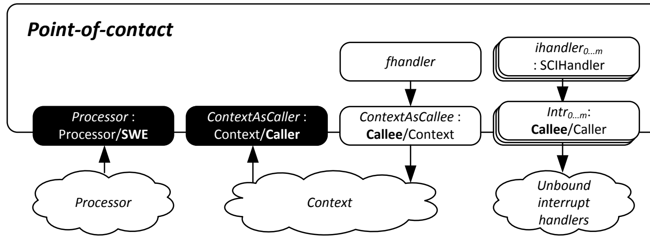


Figure 4.3 – Service relation model component representing a point-of-contact.

of determining when the initialization phase is complete any further.

4.2.2 Inputs

The input to the procedure is a service relation model of a platform instance, an application in the form of a process network, a mapping associates the processes of the process network with the processors of the platform instance and a set of, so called, channel implementation schemes.

Process Network. The process network describing the application is assumed to be available as a graph-like structure:

Definition 4.0 (Process Network) A process network is a pair $pn = \langle P, CH \rangle$ where P is a set of processes and CH a set of channels. A channel $ch \in CH$ is a 4-tuple $ch = \langle p, c, size, depth \rangle$ where p writer process (the producer), c the reader process (the consumer), $size$ the (maximum) size of messages and $depth$ the minimum depth of the channel. □

Details about the internals of the processes are not needed as we assume that the behavior of any process can be implemented on any processing element. Whether or not this assumption is justified can be checked using the approach sketched in the example of section 2.1.2.

Platform. The service relation model describing the platform is assumed to be consistent with respect to service accessibility. Also, we assume that each of the different points of contact, where the processes of the application can be

mapped to the platform, is clearly marked by means of a special component. This component, shown in figure 4.3, can be considered a black-box representing the partition of the application mapped to a single processing element.

The point of contact component has a minimum of three interfaces and one service. The interface `Processor` is used to relate the point of contact with the local processor. Through this interface the local implementation and the part of the application mapped to the point of contact can (conceptually) access services available through the processor. The two interfaces `ContextAsCaller` and `ContextAsCallee` are used to relate the point of contact with the C context of its compilation unit. Through the `ContextAsCaller` interface services provided in the context can be accessed (e.g. services offered by libraries, drivers and so on). The `ContextAsCallee` interface is used to export the service `fhandler` to the context of the local implementations compilation unit. The `fhandler` service is used to determine which other parts of the model can access the local implementation by means of function invocation.

A point of contact may have one or more unbound interrupt handlers that can be used (freely) by the application. In order to allow the procedure to consider them when implementing channels they must be related to the point of contact. Each such free interrupt handler is related to the point of contact by means of its own interface $\texttt{Intr}_x$ through which a service $\texttt{ihandler}_x$ is exported.

For a given platform with n different points of contact we have:

$$POC = \{poc_0, poc_1, \ldots, poc_n\}$$

In the following, we will extend use the "."-operator to also reference the interfaces and services of these components. For example, $poc_1.ContextAsCaller$ will give us the interface named "$ContextOfCaller$" of the component poc_1.

Mapping. The mapping of a process network onto a platform instance is map:

Definition 4.1 (Mapping, $\vec{m}$, $\vec{M}$) A mapping of a process network $pn = \langle P, CH \rangle$ onto a platform is a map $\vec{m} : P \to POC$ mapping processes of the process network to points-of-contacts in the application instance. □

The arrow in the map symbol is used to differentiate it from the symbol used for service relation models m. Channels are not considered in the mapping as determining their placement is the focus of the procedure.

Implementation Scheme. An implementation scheme is a generic template for creating an implementation of a channel in a process network. Different implementation schemes provides different kinds of channel implementations. For example, we may have one scheme for implementing channels in memory and another for implementing channels using hardware FIFO's. An implementation scheme can be described as two functions: one for mapping a channel, a mapping and a service relation model of the platform to a set of *possible implementations* and another for mapping a single possible implementation to a concrete implementation (i.e. a pair of access modules given in C).

A possible implementation is an information object recording the needs of the implementation in terms of resources, interrupt handlers and services should it be chosen:

Definition 4.2 (Possible Implementation, *pi*, *PI*) A possible implementation is a triple $pi = \langle RQ, S, H\rangle$ where RQ is a set of resource requirements, S is a set of services in the platform and H is a set of services representing the interrupt handlers associated with the different points of contact in the platform. A resource requirement $rq \in RQ$ is a pair $rq = \langle res, q\rangle$ where $res \in RES$ is a resource in the platform and $q \in \mathbb{N}$ is the quantity of the resource required. □

If chosen, a possible implementation will be transformed into an *actual implementation*:

Definition 4.3 (Actual Implementation, *ai*, *AI*) A actual implementation is a triple $ai = \langle SRR, S, H\rangle$ where SRR is a set of satisfied resource requirements, S is a set of services in the platform and H is a set of services representing the interrupt handlers associated with the different points of contact in the platform. A satisfied resource requirement srr is pair $srr = \langle res, rs\rangle$ where $res \in RES$ is a resource in the platform and rs a resource share (see section 3.3.4.1) defining the portion of res allocated to satisfy the requirement. □

The difference between the object describing a possible implementation and the object describing an actual implementation is that resource requirements have been replaced by satisfied resource requirements.

Using these two definitions an implementation scheme can be defined as:

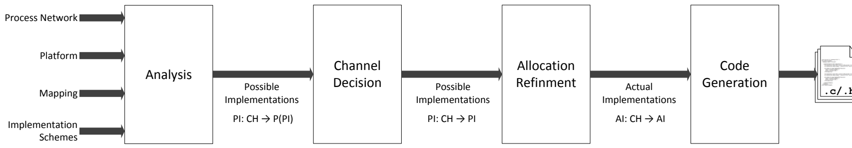


Figure 4.4 – Overview of the automated programming procedure.

Definition 4.4 (Implementation Scheme, is**,** IS**)** An implementation scheme is a pair of functions $is = \langle a, g \rangle$ where

- $a : M \times \vec{M} \times CH \rightarrow \mathcal{P}(PI)$ is the analysis function
- $g : AI \rightarrow AM_{wr} \times AM_{rd}$ is the implementation function where AM_{wr} is the domain of write access modules and AM_{rd} is the domain of read access modules

The former of these functions is called the *analysis function* of the implementation scheme and the latter the *implementation function*. □

4.2.3 Procedure Steps

Figure 4.4 shows an overview of the procedure that consists of a four steps. The sections following this one will elaborate on some of the steps that are more involved.

Step 1 (Analysis) In this step, the set of possible implementations for each channel in the application is computed. This is done straightforward by means of the analysis functions of the different implementation schemes. The result of this step is a map $PI : CH \rightarrow \mathcal{P}(PI)$ mapping channels to possible implementations:

$$PI[ch] = \bigcup_{is \in IS} (is.A(m, \vec{m}, ch)) \tag{4.1}$$

If one or more channels does not have any possible implementations the procedure terminates with error.

Step 2 (Decision) The objective of step 3 is to decide, for each channel, which of its possible implementations should be chosen as its actual implementation.

For this purpose, we formulate an optimization problem using the minimization of the sum of a cost associated with choosing a possible implementation as an actual implementation as the objective. The details of this is presented later in section 4.4.2. The result of this step is another map $CI : CH \rightarrow PI$ mapping each channel to exactly one possible implementation that we will refer to as the *chosen implementation* of the channel.

We assume that the resources provided by the platform can be used freely and that any resource requirements of the platform components have already been accounted for. Ideally, the resource requirements of the platform components should also be taken into consideration. This is, in principle, straightforward but has been left out for the sake of simplifying the presentation.

Step 3 (Allocation Refinement) From the previous step, we know that the resource requirements of the chosen implementations are guaranteed to be satisfied and we know what resources are used to satisfy each resource requirement. Next, we need to determine, for each resource, how it is distributed among the actual implementations that require some quantity of it. The resource requirements on a resource is turned into a corresponding set of resource shares by simply distributing the available resource from one end. For example, for a resource *res* with quantity 10 and two resource requirements for 2 and 5 quantities of the resource we will have the two satisfied resource shares: $srr_0 = \langle res, [0;1] \rangle$ and $srr_0 = \langle res, [2;7] \rangle$.

The result of this step is yet another map $AI : CH \rightarrow AI$ mapping each channel in the process network it an actual implementation. This map is constructed on the basis of the *CI* map of the previous step by simply replacing each resource requirement in the possible implementations of *CI* with the corresponding satisfied resource requirement.

Step 4 (Code Generation) The final step of the procedure is concerned with generating the code for the abstraction layer. This is done by means of the implementation functions of implementation schemes. For each channel *ch* in the process network the two access modules $\langle am_{rd}, am_{wr} \rangle$ are generated by applying the implementation function g to its actual implementation $AI[ch]$. The read access module am_{rd} is added to the local implementation belonging to the point of contact where the consumer process of the channel is mapped $\vec{m}[ch.c]$ and, similarly, the write access module am_{wr} to the local implementation belonging to the point of contact where the producer process is mapped $\vec{m}[ch.p]$.

4.3 Code Generation

The implementation function of an implementation scheme is used to generate C code implementations. The input to an implementation function is an abstract representation of the desired implementation in the form of an actual implementation. In this section an approach, called *service invocation synthesis*, to generating a concrete C code implementation on the basis of an abstract representation such as an actual implementation is presented. This approach requires that the services and service exchange relations used in the Service Relation Model description of the platform have previously been characterized as function as described in section 2.2.

4.3.1 Service Invocation Synthesis

A trace of relations, services and interfaces connecting a service x with another service or interface y where x is available ($x \in SA[y]$) in the service flow graph is called a *service aggregation chain*. A service aggregation chain represents one way to invoke a service offered by a foreign component through a possibly very complex structure of other components. A service aggregation chain can easily be extracted from a service flow graph using backtracking. There might exist multiple different service aggregation chains each representing different ways to invoke a given service at a particular node in the service flow graph.

Example 4.1 Figure 4.5 shows a service relation model of a simple platform consisting of a processor, a bus, a memory and a UART. The processor is assumed to be connected to a number of software components (not shown). The service aggregation chain connecting the `Receive` service of the UART with the interface $\mathtt{SWE}_0$ of the processor is also shown. ■

When a service is available at a location it means that it may somehow be accessed from that location. The service aggregation chain linking a location with an available service tells us which services, interfaces and import/export relations will be involved in accessing the service from that location. This information is not sufficient to actually realize the invocation of the service because we cannot uniquely identify the service amongst other services also aggregated at the nodes of the chain. The missing information is added to the model by means of the concepts of *invocation parameters* and *invocation functions*. An invocation parameter is simply a label/value pair associated with an entity of the Service Relation Model. Invocation parameters can be used to store addi-

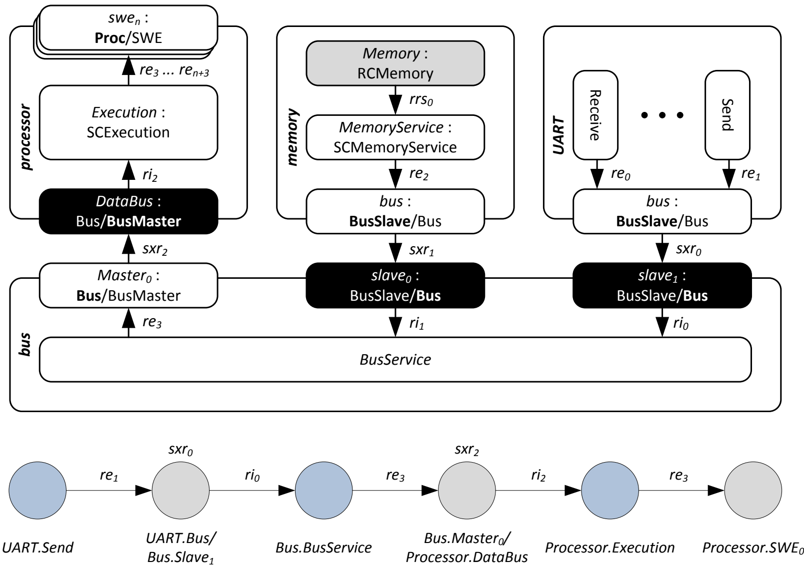


Figure 4.5 – Top: A simple platform comprising a processor, a memory and a UART. **Bottom:** Service aggregation chain for invoking the `Receive` service of the UART component from the $\mathtt{SWE}_0$ interface of the processor.

tional information about the system such as the offset of a slave peripheral in the memory space of bus component. Note that the concept of an invocation parameter is essentially identical to the parameter concept presented previously in section 3.3.5 with the exception that invocation parameters are accessible in models (i.e. they are not replaced by their constant value).

An invocation function is a function associated with an import, export or service exchange relation and is used to extract information about how a specific service available at the source node of the relation is to be invoked from the target node. The role of an invocation function is as follows: given information about how to invoke a service s from a node n_0, the invocation function associated with a relation having n_0 as source and node n_1 as target will provide the information needed to invoke s from n_1. In order to compute the desired output, the function can use the input information and any invocation parameters associated with the service and interface represented by the two nodes. The type of input expected and output produced by an invocation function is dictated by the service and interface related by the associated relation. More specifically, the type of output produced and the type of input expected by two functions associated with a pair of relations connected to the same node in a service aggregation chain must be the same. The process of determining how

to invoke a service, given a service aggregation chain, consists of applying the invocation functions associated with the relations of the chain to each other in a backwards manner as shown in the example below.

Example 4.2 Given the service service aggregation chain of figure 4.5 and information about invoking how to invoke the `Receive` service at itself I_{in} we can compute the information needed to invoke the `Receive` service from the $\texttt{swe}_0$ interface of the processor I_{out}:

$$I_{out} = re_3(ri_2(sxr_2(re_3(ri_0(sxr_0(re_1(I_{in})))))))$$

Here the names of the import, export and service exchange relations in the aggregation chain denotes the invocation function associated with that particular relation. ■

The information about "how" to invoke a given service from a call node n is described as valuations of the inputs associated with the functional characterization node (see section 2.2). Since not all input values are necessarily known at design time the use of variables are allowed. This means that, formally, a service invocation function is not a mapping between valuations of different inputs sets but rather a mapping between constructors of such valuations.

The process of synthesizing an invocation of a service can be broken into a relatively simple four step procedure:

Step 1 (Service Identification) First, we identify the service s to be invoked and the call node n (a service or an interface) from which we want to invoke s. The service aggregation chain linking s with n is extracted from the service flow graph. If the chain does not exist, it is not possible to invoke s from n and the procedure terminates with error. If there exists multiple chains, one is chosen arbitrarily.

Step 2 (Target Valuation) Second, the desired operation, represented by the service s, is chosen by choosing a valuation of the inputs of s. The chosen valuation may contain variables that can be used to postpone the choice of the actual operation until run-time.

Step 3 (Apply Invocation Functions) Third, the invocation functions are applied bottoms-up using the chosen valuation of the inputs of s as the input.

Step 4 (Decode Information) Fourth, the valuation of the inputs at the call node n resulting from applying the invocation functions is transformed into an implementation of the service access. Because variables can be used as input to the invoked service the valuation of the inputs at the call site are not values but rather constructors (e.g. expressions) for generating values.

Example 4.3 Below the definitions of the service invocation functions for the service aggregation chains of figure 4.5 are given:

$$
\begin{array}{rcl}
sxr_3(\langle name, arg_0, \ldots, arg_n \rangle) & = & \langle name, arg_0, \ldots, arg_n \rangle \\
re_3(\langle op, w, addr, val \rangle) & = & \begin{cases} \langle store, addr, val, \bot, \ldots \rangle & \text{if } op = \texttt{wr} \\ \langle load, addr, \bot, \ldots \rangle & \text{if } op = \texttt{rd} \end{cases} \\
ri_2(\langle op, w, addr, val \rangle) & = & \langle op, w, addr, val \rangle \\
sxr_2(\langle op, w, addr, val \rangle) & = & \langle op, w, addr, val \rangle \\
rs_3(\langle op, w, addr, val \rangle) & = & \langle op, w, addr, val \rangle \\
ri_0(\langle op, w, addr, val \rangle) & = & \langle op, w, \texttt{slave}_1\texttt{.offset} + addr, val \rangle \\
sxr_0(\langle op, w, addr, val \rangle) & = & \langle op, w, addr, val \rangle \\
re_1(\langle \rangle) & = & \langle \texttt{rd}, 1, 0\text{x}0, 0 \rangle
\end{array}
$$

Here we have assumed that information about which segment of the address space of the bus is occupied by the UART peripheral is stored with the bus and that the offset of this segment may be retrieved from an invocation parameter. This information could also have been stored with the UART peripheral in which case the alteration of the address should be moved from ri_0 to re_1 or sxr_0.

Given the above definitions and assuming that the UART peripheral is mounted at offset 0x200000 in the address space of the bus, the service invocation information object describing how the service `Receive` is invoked from the `proc` interface can now be computed:

$$\langle load, 0x200000 + 0x0 \rangle$$

The add operator in the result shows that the result is not a value but rather a constructor for determining a value. ■

```
void store(int* addr, int value) {
  *((int*)addr) = value;
}

int load(int* addr) {
  return *((int*)addr);
}
```

Listing 4.1 – Examples of C-binding wrapper functions

4.3.2 Integration to C

A processor is characterized by providing a service of the class `SCExecution` that represents the instructions of its instruction set. This service is made accessible for the software entities executing on the processor. A software entity can be implemented using different languages. To keep things simple, we will assume that all software is given as C code. In order to facilitate the invocation of services across the boundary between software (C code) and the processor (instructions in the instruction set) we assume to have available at set of C functions wrapping the instructions of the instruction set that may be interesting to invoke. Listing 4.1 show two such functions wrapping a word LOAD instruction and a word STORE instruction.

In the following chapter on automated programming we will need the ability to invoke services from C code. The procedure described in this chapter can be used together with a pre-processor to insert synthesized service access expressions in the code. To invoke the service *MyService* from C, we will use the following syntax:

$$[\langle o_0, \ldots, o_m \rangle = \mathit{MyService}(i_0, \ldots, i_n)]$$

where $i_0, \ldots, i_n$ are the inputs required by the service and $o_0, \ldots, o_m$ the outputs. We will allow the use of program variables in place of the input and output values. The value of some inputs must be constant and known at compile time. The *op* input of the memory service is an example of such an input. When a program variable is used, we assume there exists a conversion between the C type of the variable and the domain of the corresponding input. Furthermore, we will also allow for the use of the upper and lower bounds of resource shares to be used for specifying inputs with the understanding that they can be treated as constants.

Example 4.4 As an example of integrating the service invocation synthesis procedure with C consider the partial model of figure 4.6. The figure shows

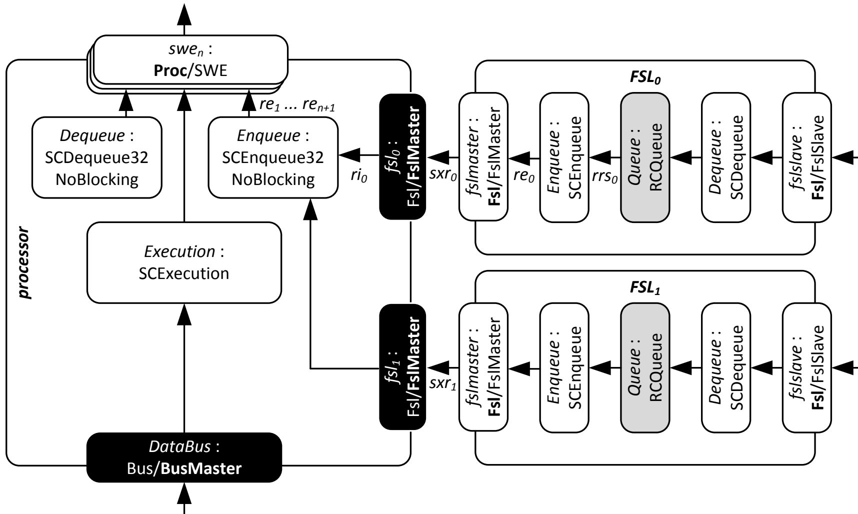


Figure 4.6 – Partial model of a platform using fast simplex links (FSL).

three hardware components: a Microblaze processor and two fast simplex links (FSL). A FSL is a hardware FIFO available in the Xilinx EDK used primarily to connect processors to each other. The two FSL's can be accessed from software by means of a set of special instructions in the instruction set of the Microblaze processor.

A FSL queue is modeled as a single component with two services, two interfaces and a single resource. The resource is an instance of the parameterized resource class `RCQueue<D,S>` where D defines the depth of the queue and S the maximum size of each token. The two services represent the familiar enqueue and dequeue operations and are declared as members of the appropriate abstract service classes. The resource is exported through both services. The two interfaces of type `Fsl/FslMaster` and `Fsl/FslSlave` are used respectively to connect the FSL to a master and a slave component.

The Microblaze processor provides access to FSL components by means of a set of special purpose instructions. The instructions can be used to enqueue and dequeue up to a maximum of 32-bits of data. In the Service Relation Model, the instructions to enqueue and dequeue 32-bits of data in a non-blocking fashion can be modeled using a pair of services belonging to the following service

classes:

$$\langle success \rangle = \texttt{SCEnqueue32NoBlocking}\,(qid, val)$$
$$\langle success, val \rangle = \texttt{SCDequeue32NoBlocking}\,(qid)$$

where $qid \in [0, 7]$ is the id of the FSL queue interface of the processor to access, $val \in [0, 2^{32} - 1]$ is the value read or written and $success \in \{\texttt{true}, \texttt{false}\}$ is an output indicating whether the operation was a success or not. The Microblaze also offers blocking versions of the FSL enqueue and dequeue instructions that are not shown in the figure.

The code below shows how to write to a queue from C using service invocation synthesis. The function takes as argument an array of 32-bit data and an integer representing the number of words to write to the queue. The writing of the queue has been abstracted using a resource share and a service:

```
void write_fsl<RCQueue<D,S> r, SCEnqueue32NoBlocking s>
  (int[] data, int len) {
    int i = 0; BOOL success = FALSE;
    for(; i < len; i++) {
      do {
        [<success> = s(r.l, data[i])];

      } while(!success)
    }
  }
```

Syntactically, the function has been parameterized by extending the header with a list of required services and resource shares. Notice that the resource share is given by the type of the resource that is belongs to rather than by the, not very descriptive, name "ResourceShare". Notice the use of the lower bound of the resource share *r* is used as the first input to specify the id of the FSL to access.

For a resource share $rs = [0; 0]$ belonging to the queue resource of FSL_0 and the Enqueue service of the processor in figure 4.6 we get the following result:

```
void write_fsl(int[] data, int len) {
  int i = 0; BOOL success = FALSE;
  for(; i < len; i++) {
    do {
      success = fsl_enqueue(0, data[i]);
    } while(!success)
  }
}
```

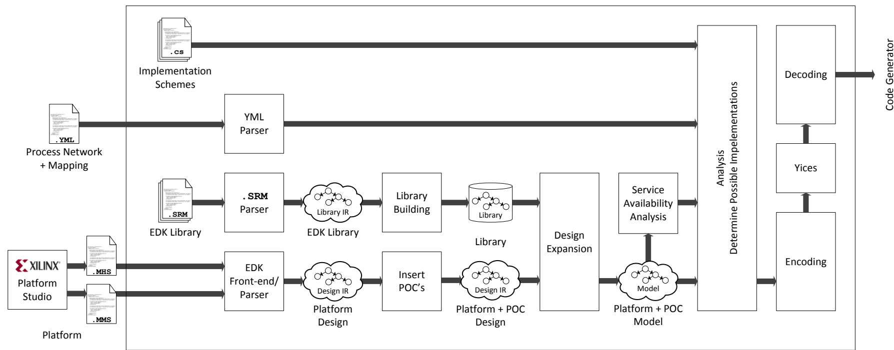


Figure 4.7 – Overview of the PAL generation tool.

The service invocation procedure inserts the proper code for accessing the enqueue service provided by the processor through the queue resource of the FSL component. ■

Notice that when using service invocation synthesis, the choice of input values is relative to the location of the invoked service. This is a key feature since it makes the invocation of a service independent of the other parts of the system. Usually, when programming access to a device the inputs to the device must be specified relative to the location of the caller (e.g. an embedded processor). For example, when reading address x in a memory from some processor p the effective address of x relative to p must be provided. The effective address of x relative to p depends on the topology and allocation of the system and it need not be the same for different choices of p.

4.4 The PAL Generation Tool

Figure 4.7 shows an overview of our proof-of-concept tool, based on the xSRM framework, implementing the procedure for automated programming used for experimental purposes. As can be seen, the tool only implements the analysis and decision making parts of the procedure – code generation is not supported at this time. The reason for this is that our purpose with the procedure is primarily to demonstrate the analysis capabilities of the Service Relation Model and that time has permitted us from completing this part of the tool.

The tool takes as input a platform designed using the Xilinx EDK tool and a YML file containing a specification of a process network and a mapping of its

processes to the processors of the application. The YML file also specifies the minimum size and depth of each channel in the process network. The MHS and MSS files describing the platform are transformed into a corresponding design using the Xilinx EDK front-end of the `xSRM` framework. Before the design is expanded into a model point-of-contact components are automatically inserted into the design.

The implementation schemes supported by the tool are defined as classes compiled into the tool itself. Adding new implementation schemes is done straightforward using class inheritance but does require a recompilation of the tool. The supported implementation schemes will be presented in detail the next section. In the tool, the optimization problem of step 2 is encoded and solved as an MAXSMT problem using the Yices solver. The details of MAXSMT and the encoding are presented in detail later in section 4.4.2.

4.4.1 Implementation Schemes

In this section, we will present a collection of implementation schemes that we have developed for testing purposes. The analysis function of each scheme is given as a query on the structure of the service relation model of the platform presented in an implementation independent way using set constructor notation. Each element in the set represents a possible implementation of the channel with respect to the implementation scheme. The implementation functions will not be presented explicitly. Instead, the resulting access modules will be given as C code parameterized using the notation for synthesizing service invocation presented previously in section 4.3.

The schemes presented here are divided into two categories: *synthesized channel implementation schemes* that provides channels implemented as circular buffers in memory and *platform-provided implementation schemes* that provides channels based on existing channels (e.g. hardware FIFO's) in a platform.

4.4.1.1 Synthesized Channel Implementation Schemes

The synthesized channel implementation schemes produces channels realized as circular buffers in memory. A buffer is organized as a series of equally sized tokens each of which is capable of containing a single message. Besides the buffer a synthesized channel also consists of two control variables (read index and write index). The read index holds the position of the first full token in the buffer. Similarly, the write index holds the position of the first free token in the

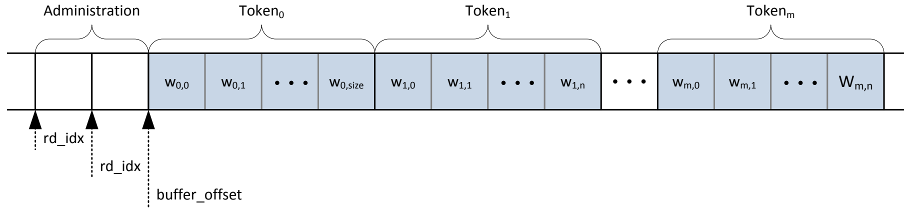


Figure 4.8 – Layout of a synthesized channel in memory. The tokens of the channel has size n and the depth of the channel is m.

buffer. The layout of a channel in memory is shown in figure 4.8. The status of the channel can be determined by comparing the two indices. If the read and write indices are equal the channel is either full or empty. To distinguish between the two situations the channel is extended with an additional token which is never used. Now, the following expressions can be used to test the status of a channel:

$$\text{full}(ch) = ((\texttt{ch.wr_idx} + 1) \textbf{ mod } \texttt{ch.depth} = \texttt{ch.rd_idx})$$
$$\text{empty}(ch) = (\texttt{ch.wr_idx} = \texttt{ch.rd_idx})$$

Assuming that we use two times 32-bits to store the read and the write indices a channel $ch = \langle p, c, size, depth \rangle$ requires $size \times (depth + 1) + 2 \times 4$ bytes of memory.

In the following, four different implementation schemes based on circular buffers will be presented. The implementation schemes differs in how they implement synchronization as shown in the table below.

Scheme	Producer to consumer	Consumer to producer
$\texttt{IS}_0$	Polling	Polling
$\texttt{IS}_1$	Notification (function)	Notification (function)
$\texttt{IS}_2$	Notification (interrupt)	Notification (interrupt)
$\texttt{IS}_3$	Polling	Notification (interrupt)
$\texttt{IS}_4$	Notification (interrupt)	Polling

Polling Synchronization. This implementation scheme provides a memory-backed channel synchronized using polling. Listings 4.2 and 4.3 shows the template C code for implementing the access modules of this scheme. The template uses a number of functions shared with other implementation schemes. These shared functions are given in listings 4.4.

In order for this scheme to applicable, we need $(ch.depth+1)\times ch.size+8$ bytes

```
#include <shared.c>
void chx_write_init<RCMemory r, SCMemoryService s>() {
  initialize_memory<r, s>();
}

void chx_write<RCMemory r, SCMemoryService s>(void* data) {
  while(is_full<r,s>());
  copy_to_token<r,s>(data);
  inc_wr_idx<r,s>();
}

void chx_write_handler() {
  /* empty */
}
```

Listing 4.2 – Template for the write access module of $\texttt{IS}_0$.

```
#include <shared.c>
void chx_read_init() { /* empty */ }

void chx_read<RCMemory r, SCMemoryService s>(void* data) {
  while(is_empty<rs,s>());
  copy_from_token<rs,s>(data);
  inc_rd_idx<rs,s>();
}

void chx_read_handler() { /* empty */ }
```

Listing 4.3 – Template for the read access module of $\texttt{IS}_0$.

```
#define RD_IDX_OFFSET 0 /* local in memory rs */
#define WR_IDX_OFFSET 4
#define BUFFER_OFFSET 8

void initialize_memory<RCMemory rs, SCMemoryService s>() {
  int i = 0;
  for(; i < CH_DEPTH * CH_SIZE, i++) { /* clear buffer */
    int address = BUFFER_OFFSET + i;
    [s(wr, 1, ⌊rs⌋ + address, 0)];
  }
  /* initialize read/write indices */
  [s(wr, 4, ⌊rs⌋ + RD_IDX_OFFSET, 0)];
  [s(wr, 4, ⌊rs⌋ + WR_IDX_OFFSET, 1)];
}

void copy_to_token<RCMemory rs, SCMemoryService s>(void* data) {
  int wr_idx, i = 0;
  [wr_idx = s(rd, 4, ⌊rs⌋ + RD_IDX_OFFSET, 0)];
  for(; i < CH_SIZE; i++) {
    int address = BUFFER_OFFSET + (CH_SIZE * wr_idx) + i;
    [s(wr, 1, ⌊rs⌋ + address, data[i])];
  }
}

void copy_from_token<RCMemory rs, SCMemoryService s>(void* data) {
  int rd_idx, i = 0;
  [rd_idx = s(rd, 4, ⌊rs⌋ + WR_IDX_OFFSET, 0)];
  for(; i < CH_SIZE; i++) {
    int address = BUFFER_OFFSET + (CH_SIZE * rd_idx) + i;
    [data[i] = s(rd, 1, ⌊rs⌋ + address, 0)];
  }
}

void inc_rd_idx<RCMemory rs, SCMemoryService s>() {
  [int rd_idx = s(rd, 4, ⌊rs⌋ + RD_IDX_OFFSET, 0)];
  int new = (rd_idx + 1) % CH_DEPTH;
  [s(wr, 4, ⌊rs⌋ + RD_IDX_OFFSET, new)];
}

void inc_wr_idx<RCMemory rs, SCMemoryService s>() {
  [int wr_idx = [s(rd, 4, rs_l + WR_IDX_OFFSET, 0)];
  int new = (wr_idx + 1) % CH_DEPTH;
  [s(wr, 4, rs_l + WR_IDX_OFFSET, new)];
}

BOOL is_full<RCMemory rs, SCMemoryService s>() {
  [int rd_idx = s(rd, 4, ⌊rs⌋ + RD_IDX, 0)];
  [int wr_idx = s(rd, 4, ⌊rs⌋ + WR_IDX, 0)];
  return ((wr_idx + 1) % CH_DEPTH) == rd_idx;
}

BOOL is_empty<RCMemory rs, SCMemoryService s>() {
  [int rd_idx = s(rd, 4, ⌊rs⌋ + RD_IDX, 0)];
  [int wr_idx = s(rd, 4, ⌊rs⌋ + WR_IDX, 0)];
  return wr_idx == rd_idx;
}
```

Listing 4.4 – Shared functions of the synthesized channel access modules. The number of tokens and size of each token is given by means of the two definitions `CH_DEPTH` and `CH_SIZE`.

of memory in a memory accessible from both the producer and consumer points of contact. In terms of analysis, this means that we need to query the platform for `RCMemory` resources accessible through an instance of the `SCMemoryService` service class at the two points-of-contact where the reading and writing processes are mapped. Here we assume that the members of `RCMemory` resource class represent bytes of memory and that members of the `SCMemoryService` are parameterized using the four parameters $\langle op, w, addr, val\rangle$. The analysis function of the scheme is given below:

Definition 4.5 ($\mathtt{IS}_0$: Analysis)

$$
\begin{aligned}
&\mathtt{IS}_0(m, \vec{m}, ch) = \{\ \langle\ \{\ \langle r, (ch.depth + 1) \times ch.size + 8\rangle\ \}, \{\ s\ \}, \emptyset\rangle \in PI : \\
&\quad r \in \mathtt{RCMemory}.RES\ \wedge \\
&\qquad s \in \mathtt{SCMemoryService}.S \cap \overline{RA}[r] \cap \\
&\qquad\quad (SA[\vec{m}[p].\mathtt{Processor}] \cup SA[\vec{m}[p].\mathtt{ContextAsCaller}) \cap \\
&\qquad\quad (SA[\vec{m}[c].\mathtt{Processor}] \cup SA[\vec{m}[c].\mathtt{ContextAsCaller}]) \\
&\}
\end{aligned}
$$

□

Notice that we require for the memory resource to accessible through the *same* instance of the `SCMemoryService` class at both points of contact. A more general solution would allow the resource to be accessible through two possibly different instances of the class. This would allow the implementation scheme to be used with a wider range of different representations of memories. In the interest of keeping things simple, however, we will refrain this.

The channel access protocol is inspired by a similar protocol used in a TTL implementation for a Philip's DSP platform, [94]. A key feature of the protocol is that it ensures safe concurrent access to the channel buffer without the use of synchronization primitives such as locks or semaphores [5]. This implementation scheme is used as a "fallback" since it requires very little of the platform.

Notification Synchronization. The previous protocol used for synchronization wastes computing cycles on busy waiting. A more efficient protocol would use notification (e.g. via interrupts) to signal changes to the status of the channel thus allowing a process waiting for the channel to become either not-empty or not-full to be suspended. This, however, is only possible if the waiting process is hosted on a processor with an operating system providing the service of blocking and if there exist a path in the platform for propagating the notification between the two communicating processes.

Figure 4.9 – Tri-processor system demonstrating accessability of interrupt handlers.

Notification is applicable in two slightly different scenarios. In the first scenario, the two processes communicating via a given channel are mapped to the same point-of-contact. In this case, notification can be done directly by means of function calls since both processes exists in the same context. In the other scenario, the two processes are mapped to different processors which complicates things somewhat. In order for a channel access module of a local implementation on one processor to invoke a handler in a local implementation on another processor there must be a path in the platform through which the call can be facilitated. An obvious and efficient way of implementing this in a platform is by means of interrupts. For a particular processor there are only a limited number of interrupt inputs available and each interrupt input can be used to invoke at most one handler. This means that the interrupts must be treated as finite resources. Furthermore, the set of interrupts of a processor may not necessarily have the same accessibility. This is illustrated by means of a small example:

Example 4.5 Figure 4.9 shows a schematic of a tri-processor system, divided into three sub-systems, demonstrating accessibility of interrupt handlers. The sub-systems are fairly identical and connected to each other by means of a set of interrupt signals. The single interrupt port of each processor is increased to 32 by means of interrupt controllers. The three "bus-to-interrupt" controllers are used for raising interrupts through a memory mapped bus interface. The topology of the platform restricts the access to these controllers meaning that not all of the processors can interrupt each other: The software running on processor pe_0 can raise interrupts in processor pe_1 through bi_0 The software running on processor pe_1 can raise interrupts in processor pe_0 through bi_1 and bi_2 and in pe_2 through bi_2. Finally, the software running on processor pe_2 can raise interrupts

in itself and in processor pe_0. Notice also that one of the interrupt ports of the interrupt controller ic_0 is used for connecting processor pe_0 to a timer/counter.

The software running on the three processors includes interrupt handlers for each of their 32 interrupt inputs. Using a Service Relation Model description of the platform, we may compute the accessibility of these handlers. For the 32 interrupt handlers of Sub-system$_0$ we get the following result:

Interrupt	**Accessible from**
0–3	Sub-system$_1$, Sub-system$_2$
4–19	Sub-system$_1$
20–30	*Not used*
31	Timer/Counter tc_0

The example shows that the different handlers (interrupts) cannot be used arbitrarily for synchronization purposes. ■

For particular channel, we need to synchronize both the reading and the writing. The synchronization of the reading and the writing can be done independently of each other. For example, we may use notification to let the reader process notify the writer process that the channel is no longer full and polling to let the writer process notify the reader process that the channel is no longer empty. The are a total of four different possibilities of combining polling and notification.

A process may be notified that the status of a channel has been changed by invoking the handler function associated with one of its access modules. Since the invocation of a handler function happens asynchronously with respect to the process it conceptually belongs it is necessary to synchronize the two. For this purpose a (counting) semaphore is used. A semaphore is represented as a resource `RCSemaphore` and three service classes:

$$\begin{array}{ll}\langle\rangle = \texttt{SCSemInit}\,(semid, val) & semid, val \in \mathbb{N}\\ \langle\rangle = \texttt{SCSemWait}\,(semid) & semid \in \mathbb{N}\\ \langle\rangle = \texttt{SCSemPost}\,(semid) & semid \in \mathbb{N}\end{array}$$

`IS`$_1$: **Two-way Notification via Function Invocation** This implementation scheme is used for implementing channels where both the producer and the consumer are mapped to the same point of contact ($\vec{m}[p] = \vec{m}[c]$). Channel implementations based on this scheme are synchronized using sempahores and

the notification is implemented by means of function invocation. The analysis function of the implementation scheme is given below:

Definition 4.6 ($\texttt{IS}_1$: Analysis)

$$\texttt{IS}_1(m, \vec{m}, ch) = \{ \langle \{ \langle r_0, ch.depth \times ch.size + 8 \rangle, \langle r_1, 1 \rangle, \langle r_2, 1 \rangle \}, \{ s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7 \}, \emptyset \rangle :$$

$$r_0 \in \texttt{RCMemory}.RES \wedge$$
$$s_0 \in \texttt{SCMemoryService}.S \cap \overline{RA}[r_0] \cap (SA[\vec{m}[p].\texttt{Processor}] \cup SA[\vec{m}[p].\texttt{ContextAsCaller}) \wedge$$
$$r_1 \in \texttt{RCSemaphore}.RES \wedge$$
$$s_1 \in \texttt{SCSemInit}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\texttt{ContextAsCaller} \wedge$$
$$s_2 \in \texttt{SCSemWait}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\texttt{ContextAsCaller} \wedge$$
$$s_3 \in \texttt{SCSemPost}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\texttt{ContextAsCaller} \wedge$$
$$r_2 \in \texttt{RCSemaphore}.RES \wedge$$
$$s_4 \in \texttt{SCSemInit}.S \cap \overline{RA}[r_2] \cap SA[\vec{m}[p].\texttt{ContextAsCaller} \wedge$$
$$s_5 \in \texttt{SCSemWait}.S \cap \overline{RA}[r_2] \cap SA[\vec{m}[p].\texttt{ContextAsCaller} \wedge$$
$$s_6 \in \texttt{SCSemPost}.S \cap \overline{RA}[r_2] \cap SA[\vec{m}[p].\texttt{ContextAsCaller} \wedge$$
$$\vec{m}[c].FHandler \in SA[\vec{m}[p].\texttt{ContextAsCaller}] \wedge$$
$$\vec{m}[p].FHandler \in SA[\vec{m}[c].\texttt{ContextAsCaller}]$$

$$\}$$

□

The scheme requires a total of three resources and eight services. The first resource r_0 is the memory where the channel buffer and indices should be stored. For this resource, we require that the memory must be accessible through an instance of the `CSMemoryService` class at the `Processor` interface of the point of contact to which the producer and consumer processors are mapped. The other two resources r_1 and r_2 represent the two counting semaphores that are needed to implement the synchronization. Both semaphore resources must be accessible through instances of the `SCSemInit`, `SCSemWait` and `SCSemPost` service classes at the `ContextAsCaller` interface of the point of contact.

The requirement that the producer and the consumer are mapped to the same point of contact is modeled indirectly by requiring that the `FHandler` service of $\vec{m}[c]$ is accessible at the `ContextAsCaller` interface of $\vec{m}[p]$ and vice versa. Alternatively, we could simply have required that $\vec{m}[c] = \vec{m}[p]$ but we prefer the other formulation as it more explicitly captures our intent. Also, notice that we do not use the additional token for channels implemented using this

```
void chx_write_init
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemInit s1>() {
    initialize_memory<r0,s0>();
    [s1(⌊r1⌋, CH_DEPTH)]; /* initialize semaphore to CH_DEPTH */
  }


void chx_write
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemWait s2>(void* data) {
    [s2(⌊r1⌋)]; /* wait */
    copy_to_token<r0,s0>(data);
    inc_wr_idx<r0,s0>();
    chx_read_handler(); /* notify */
  }

void chx_write_handler<RCSemaphore r1, SCSemPost s3>() {
  [s3(⌊r1⌋)]; /* post */
}
```

Listing 4.5 – Template for the write access module of $\mathtt{IS}_1$.

```
void chx_read_init<RCSemaphore r2, SCSemInit s4>() {
  [s4(⌊r2⌋, 0)]; /* initialize semaphore to 0 */
}

void chx_read
  <RCMemory r0, RCSemaphore r2, SCMemoryService s0, SCSemWait s5>(void* data) {
    [s5(⌊r2⌋)]; /* wait */
    copy_from_token<r0,s0>(data);
    inc_rd_idx<r0,s0>();
    chx_write_handler(); /* notify */
  }

void chx_read_handler<RCSemaphore r1, SCSemPost s6>() {
  [s6(⌊r2⌋)]; /* post */
}
```

Listing 4.6 – Template for the read access module of $\mathtt{IS}_1$.

scheme. This is not necessary because the two semaphores are used to keep track of the queue status.

Listing 4.5 and 4.6 gives the C code templates for implementing the access modules of this scheme.

Scheme $\mathtt{IS}_2$ & $\mathtt{IS}_3$: One-way Notification via Interrupt. Implementation scheme $\mathtt{IS}_2$ generates channels where the consumer-to-producer direction is synchronized using notification and the producer-to-consumer direction is implemented using polling. An implementations based on $\mathtt{IS}_2$ is possible if there exists a memory in the platform accessible from both the reading and writing processes, a semaphore accessible through a wait, a signal and an init service at

the writer process and an interrupt handler in the writer which can be invoked by the reader. The analysis function of $\mathtt{IS}_2$ is given below:

Definition 4.7 ($\mathtt{IS}_2$: Analysis)

$$
\begin{aligned}
&\mathtt{IS}_2(m, \vec{m}, ch) = \{\ \langle\ \{\ \langle r_0, (depth+1) \times size + 8\rangle\,, \langle r_1, 1\rangle\ \}, \{\ s_0, s_1, s_2, s_3\ \}, \{\ i\ \}\rangle \in PI :\\
&\quad r_0 \in \mathtt{RCMemory}.RES\ \wedge\\
&\qquad s_0 \in \mathtt{SCMemoryService}.S \cap \overline{RA}[r_0] \cap\\
&\qquad\quad (SA[\vec{m}[p].\mathtt{Processor}] \cup SA[\vec{m}[p].\mathtt{ContextAsCaller}) \cap\\
&\qquad\quad (SA[\vec{m}[c].\mathtt{Processor}] \cup SA[\vec{m}[c].\mathtt{ContextAsCaller}])\ \wedge\\
&\quad r_1 \in \mathtt{RCSemaphore}.RES\ \wedge\\
&\qquad s_1 \in \mathtt{SCSemInit}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[p].\mathtt{ContextAsCaller}\ \wedge\\
&\qquad s_2 \in \mathtt{SCSemWait}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[p].\mathtt{ContextAsCaller}\ \wedge\\
&\qquad s_3 \in \mathtt{SCSemPost}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[p].\mathtt{ContextAsCaller}\ \wedge\\
&\quad i \in \vec{m}[p].IHandlers \cap (SA[\vec{m}[c].\mathtt{Processor}] \cup SA[\vec{m}[c].\mathtt{ContextAsCaller}])\\
&\}
\end{aligned}
$$

□

Implementation scheme $\mathtt{IS}_3$ is the opposite of $\mathtt{IS}_2$ and uses notification in the producer-to-consumer direction and polling in the consumer-to-producer direction. The analysis function of $\mathtt{IS}_3$ is given below:

Definition 4.8 ($\mathtt{IS}_3$: Analysis)

$$
\begin{aligned}
&\mathtt{IS}_3(m, \vec{m}, ch) = \{\ \langle\ \{\ \langle r_0, (depth+1) \times size + 8\rangle\,, \langle r_1, 1\rangle\ \}, \{\ s_0, s_1, s_2, s_3\ \}, \{\ i\ \ \}\rangle \in PI :\\
&\quad r_0 \in \mathtt{RCMemory}.RES\ \wedge\\
&\qquad s_0 \in \mathtt{SCMemoryService}.S \cap \overline{RA}[r_0] \cap\\
&\qquad\quad (SA[\vec{m}[p].\mathtt{Processor}] \cup SA[\vec{m}[p].\mathtt{ContextAsCaller}) \cap\\
&\qquad\quad (SA[\vec{m}[c].\mathtt{Processor}] \cup SA[\vec{m}[c].\mathtt{ContextAsCaller}])\ \wedge\\
&\quad r_1 \in \mathtt{RCSemaphore}.RES\ \wedge\\
&\qquad s_1 \in \mathtt{SCSemInit}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\mathtt{ContextAsCaller}\ \wedge\\
&\qquad s_2 \in \mathtt{SCSemWait}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\mathtt{ContextAsCaller}\ \wedge\\
&\qquad s_3 \in \mathtt{SCSemPost}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\mathtt{ContextAsCaller}\ \wedge\\
&\quad i \in \vec{m}[c].IHandlers \cap (SA[\vec{m}[p].\mathtt{Processor}] \cup SA[\vec{m}[p].\mathtt{ContextAsCaller}])\\
&\}
\end{aligned}
$$

□

The C code templates for these two implementation schemes are very similar and mere hybrids of the templates used for $\texttt{IS}_0$ and $\texttt{IS}_1$ and have been moved to appendix C.

$\texttt{IS}_4$**: Two-way Notification (via Interrupt).** The last of the synthesized channel implementation schemes is $\texttt{IS}_4$ which implements synchronization using notification in both directions.

Definition 4.9 ($\texttt{IS}_4$**: Analysis)**

$$
\begin{aligned}
&\texttt{IS}_4(m,\vec{m},ch) = \{\langle\{\langle r_0, ch.depth \times ch.size + 8\rangle, \langle r_1, 1\rangle, \langle r_2, 1\rangle\}, \{s_0,s_1,s_2,s_3,s_4,s_5,s_6,s_7\}, \{i_0,i_1\}\rangle : \\
&\quad \vec{m}[p] \neq \vec{m}[c] \wedge \\
&\quad r_0 \in \texttt{RCMemory}.RES \wedge \\
&\qquad s_0 \in \texttt{SCMemoryService}.RES \cap \overline{RA}[r_0] \cap \\
&\qquad\quad (SA[\vec{m}[p].\texttt{Processor}] \cup SA[\vec{m}[p].\texttt{ContextAsCaller}) \cap \\
&\qquad\quad (SA[\vec{m}[c].\texttt{Processor}] \cup SA[\vec{m}[c].\texttt{ContextAsCaller}]) \wedge \\
&\quad r_1 \in \texttt{RCSemaphore}.RES \wedge \\
&\qquad s_1 \in \texttt{SCSemInit}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\texttt{ContextAsCaller} \wedge \\
&\qquad s_2 \in \texttt{SCSemWait}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\texttt{ContextAsCaller} \wedge \\
&\qquad s_3 \in \texttt{SCSemPost}.S \cap \overline{RA}[r_1] \cap SA[\vec{m}[c].\texttt{ContextAsCaller} \wedge \\
&\quad r_2 \in \texttt{RCSemaphore}.RES \wedge \\
&\qquad s_4 \in \texttt{SCSemInit}.S \cap \overline{RA}[r_2] \cap SA[\vec{m}[p].\texttt{ContextAsCaller} \wedge \\
&\qquad s_5 \in \texttt{SCSemWait}.S \cap \overline{RA}[r_2] \cap SA[\vec{m}[p].\texttt{ContextAsCaller} \wedge \\
&\qquad s_6 \in \texttt{SCSemPost}.S \cap \overline{RA}[r_2] \cap SA[\vec{m}[p].\texttt{ContextAsCaller} \wedge \\
&\quad i_0 \in \vec{m}[c].IHandlers \cap (SA[\vec{m}[p].\texttt{Processor}] \cup SA[\vec{m}[p].\texttt{ContextAsCaller}]) \wedge \\
&\quad i_1 \in \vec{m}[p].IHandlers \cap (SA[\vec{m}[c].\texttt{Processor}] \cup SA[\vec{m}[c].\texttt{ContextAsCaller}]) \\
&\}
\end{aligned}
$$

□

Notice that $\texttt{IS}_3$ is only applicable if $\vec{m}[p] \neq \vec{m}[c]$ meaning that the producer and the consumer are mapped to *different* points of contacts. This is done to keep the number of possible channels down. If this scheme could be used for channels where both the producer and the consumer are mapped to the same point of contact then the number of possible implementations is $\binom{h}{2} \times m \times s^2$ where h is the number of usable handlers, m the number of usable memory resources and s the number of usable semaphore resources. The presence of

the binomial coefficient in the equation means that the number of channels will grow rapidly when h is increased. For a channel with 32 usable interrupt handlers, 1 memory resource and 1 semaphore resource there will be a total of 496 possible implementations that differs only on the way the two interrupt handlers are chosen.

Using interrupts to implement notification between two processes mapped to the same processor must be considered inferior to using function invocation and, thus, the previously presented implementation scheme $\texttt{IS}_1$, for implementing two-way notification via function invocation, should be used instead.

The C templates used for generating the access modules used with $\texttt{IS}_4$ is very similar to those used for $\texttt{IS}_1$ and can be found in appendix C. Finally, notice that we do not use the additional token for channels implemented using this scheme. This is not necessary because the two semaphores are used to keep track of the queue status.

4.4.1.2 Platform Provided Implementation Schemes

A platform provide channel is a channel that exists in the platform such as a point-to-point link between two processors or a message queue provided by an operating system. It is important to consider such channels as possible candidates for realizing the channels of the application for two reasons: First, since the channels are already part of the system it makes sense to use them before introducing synthesized software queues. Second, it is not always possible to realize a channel as a synthesized software queue and in these cases it may be possible to use a platform provided channel instead. For a given platform, there are only a fixed number of platform-provided channels available and, consequently, they need to be treated as resources.

We assume that platform-provided channels are represented by resources belonging to the parameterized resource class `RCQueue<D,S>` where the parameters D and S respectively describes the depth and size of the member queues. A particular channel $ch = \langle p, c, size, depth \rangle$ can potentially be implemented by resources belonging to the resources classes where $D \geq depth$ and $S \geq size$.

We will consider two slightly different kinds of platform provided implementation schemes. One scheme will be used for implementing channels using low-level processor-to-processor FIFO's and the other for implementing channels using software message queues. The two schemes differs mainly in how the queue is accessed.

```
void chx_write_init<>() { /* empty */ }

void chx_write<RCQueue<D,S> r, SCEnqueue32NoBlocking s0>(void* data) {
  int i = 0;
  for(; i * 4 < CH_SIZE; i++) {
    while(![r->s0(((INT32*)data)[i])]);
  }
}

void chx_write_handler<>() { /* empty */ }
```

Listing 4.7 – Template for the write access module of IS$_5$.

Scheme IS$_5$: 32-bit Enqueue/Dequeue, No Blocking. This implementation scheme is based on queues accessed via services representing non-blocking enqueue/dequeue operations with a capacity of 32-bits:

$$\begin{array}{ll} \langle success \rangle = \texttt{SCEnqueue32NoBlocking}\,(val) & success \in \{\texttt{true},\ \texttt{false}\}, val \in \mathbb{Z}_{32} \\ \langle success, val \rangle = \texttt{SCDequeue32NoBlocking}\,() & success \in \{\texttt{true},\ \texttt{false}\}, val \in \mathbb{Z}_{32} \end{array}$$

Because the size of the tokens is fixed at 32-bit this scheme will split larger tokens into chunks of 32-bit. This means that for the channel $ch = \langle p, c, size, depth \rangle$ we need a queue resource with at least depth $depth \times size$ and $size \geq 4$ bytes. The analysis function of this implementation scheme is given below and the C templates for the channel access modules of this scheme are shown in listings 4.7 and 4.8.

Definition 4.10 (IS$_5$: Analysis)

$$\begin{array}{l} \texttt{IS}_4(m, \vec{m}, ch) = \{ \langle \{ \langle r, 1 \rangle \}, \{ s_0, s_1 \}, \rangle \in PI : \\ \qquad r \in \texttt{RCQueue<D,S>}.RES \wedge D \geq ch.depth \times ch.size \wedge S \geq 4 \wedge \\ \qquad\qquad s_0 \in \texttt{SCEnqueue32NoBlocking}.S \cap \overline{RA}[r] \cap \\ \qquad\qquad\qquad (SA[\vec{m}[p].\texttt{Processor}] \cup SA[\vec{m}[p].\texttt{ContextAsCaller}) \wedge \\ \qquad\qquad s_1 \in \texttt{SCDequeue32NoBlocking}.S \cap \overline{RA}[r] \cap \\ \qquad\qquad\qquad (SA[\vec{m}[c].\texttt{Processor}] \cup SA[\vec{m}[c].\texttt{ContextAsCaller}]) \\ \} \end{array}$$

□

Scheme IS$_y$: Unlimited Enqueue/Dequeue, Blocking. The target of this scheme is (software) message queues with unlimited capacity providing blocking enqueue/dequeue operations. An example of such a queue, represented in

```
void chx_read_init<>() { /* empty */ }

void chx_read<RCQueue<D,S> r, SCDequeue32NoBlocking s1>(void* data) {
  int i = 0;
  for(; i * 4 < CH_SIZE; i++) {
    BOOL success = FALSE;
    while(!success) {
      [<success, val> = r->s1(((INT32*)data)[i])];
    }
  }
}

void chx_read_handler<>() { /* empty */ }
```

Listing 4.8 – Template for the read access module of $\mathtt{IS}_5$.

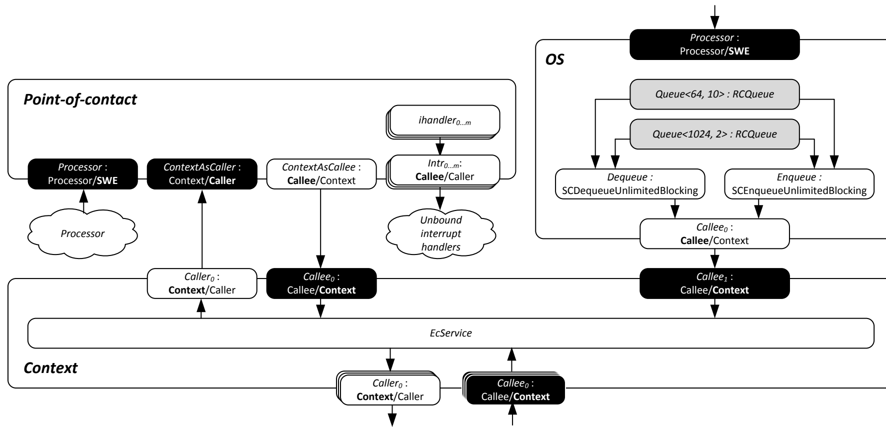


Figure 4.10 – Partial model showing a software message queue provided by an operating system component.

the Service Relation Model, is shown in figure 4.10. In the figure, the software queue is modeled as a resource provided by a component representing an operating system and exported to the execution context associated with the processor through two services representing C functions for enqueuing and dequeuing data:

$$\langle\rangle = \mathtt{SCEnqueueUnlimitedBlocking}(ptr) \quad ptr \in \text{c pointer}$$
$$\langle\rangle = \mathtt{SCDequeueUnlimitedBlocking}(ptr) \quad ptr \in \text{c pointer}$$

```
void chx_write_init<>() { /* empty */ }

void chx_write<RCQueue<D,S> r, SCEnqueueUnlimitedBlocking s0>(void* data) {
  [r->s0(data)];
}

void chx_write_handler<>() { /* empty */ }
```

Listing 4.9 – Template for the write access module of $\texttt{IS}_6$.

```
void chx_read_init<>() { /* empty */ }

void chx_read<RCQueue<D,S> r, SCDequeueUnlimitedBlocking s1>(void* data) {
  [r->s1(data)];
}

void chx_read_handler<>() { /* empty */ }
```

Listing 4.10 – Template for the read access module of $\texttt{IS}_6$.

Definition 4.11 ($\texttt{IS}_6$: **Analysis)**

$$
\begin{aligned}
\texttt{IS}_6(m, \vec{m}, ch) = \{ \langle \{ \langle r, 1 \rangle \}, \{ s_0, s_1 \}, \rangle \in PI : \\
r \in \texttt{RCQueue<D,S>}.RES \wedge D \geq ch.depth \wedge S \geq ch.size \wedge \\
s_0 \in \texttt{SCEnqueueUnlimitedBlocking}.S \cap \overline{RA}[r] \cap \\
SA[\vec{m}[p].\texttt{ContextAsCaller} \wedge \\
s_1 \in \texttt{SCDequeueUnlimitedBlocking}.S \cap \overline{RA}[r] \cap \\
SA[\vec{m}[c].\texttt{ContextAsCaller}] \\
\}
\end{aligned}
$$

□

As can be seen, the analysis used for this implementation scheme is almost identical to the analysis used for the previously presented scheme targeting FIFO's with 32-bit data capacity. The C templates for the read and write access modules of the implementation scheme are given in 4.9 and 4.10.

4.4.2 Channel Implementation Selection

In this section, we will present a formulation of the optimization as a MAXSMT problem using the Yices SMT solver.

4.4.2.1 A Short Introduction to MAXSMT and Yices

MAXSMT is an extension to SMT that allows for the encoding of optimization problems, [69]. A MAXSMT problem consists of a number of clauses, called assertions, each of which is associated with a weight. A clause is a first-order logic formula where the predicates may be binary-valued functions over non-binary variables. The type of functions and variables allowed is defined by the theories supported (e.g. linear arithmetics, bit vectors, uninterpreted functions). A solution to a MAXSMT problem is a valuation of the variables so that the sum of the weights associated with clauses falsified by the valuation is minimized. A more appropriate name for this class of problems would be ”weighted MINSMT” but since the name MAXSMT is used by the Yices solver – the only solver, that we know of, that is capable of evaluating these kind of problems – we will stick with the name MAXSMT.

Example 4.6 As a small example of a MAXSMT problem consider the four clauses given below:

$$\textbf{assert_w}(\neg a, 8),\ \textbf{assert}(b \vee c \rightarrow a),\ \textbf{assert_w}(a \wedge b, 3),\ \textbf{assert_w}(c, 4)$$

Each of the four clauses with the exception of the second consists of a formula and a weight. The second clause does not have a weight associated with it meaning that it must be true. Notice that the conjunction of the four formulas alone is unsatisfiable. The solution with the minimal cost of 7 is achieved by choosing $a, b, c = false$ which falsifies clauses 3 and 4.

To keep the example simple only boolean logic was used. An important feature of MAXSMT, however, is that it supports the mixing of multiple different theories. ■

MAXSMT is quite expressive but computationally expensive. The MAXSMT problem is a generalization of the SAT problem which is known to be NP complete. For this reason, MAXSMT is at least as hard. Although MAXSMT will most likely be too slow to handle larger problems it was chosen because its expressibility. This is acceptable because our primary focus is to show the worth of the Service Relation Model.

4.4.2.2 Problem Encoding

The formulation uses a number of variables to describe whether or not a particular possible implementation should be chosen as an actual implementation and

to represent the allocation of resources and interrupt handlers to possible implementations. To simplify the presentation, these variables are ordered into three groups where a member variable of a group can be retrieved using a map-like notation. The three groups are:

$s[pi]$	boolean	Possible implementation pi chosen as an actual implementation
$\vec{r}\ [pi, r]$	integer	Quantity of resource r allocated to possible implementation pi
$\vec{h}\ [pi, h]$	boolean	Handler h allocated to possible implementation pi

We implicitly require that the value of all $\vec{r}\ [pi, r]$ variables must be non-negative.

In addition to the variables, the optimization problem consists of a number of assertions. The first of these assertions, 4.2, encodes the requirements for each possible implementation in terms of resources and handlers and the requirement that each channel must have exactly one actual implementation:

$$\textbf{assert} \bigoplus_{ch \in CH} \left(\bigwedge_{pi \in PI(ch)} \left(s[pi] \wedge \bigwedge_{rq \in pi.RQ} \left(\vec{r}\ [pi, rq.r] = rq.q \right) \wedge \bigwedge_{h \in pi.H} \vec{h}\ [pi, h] \right) \right) \tag{4.2}$$

Here $PI : CH \rightarrow \mathcal{P}(PI)$ is a function returning the set of possible implementations of a given channel. The next two assertions 4.3 and 4.4 state that, for each resource in the platform, the amount of resource allocated to each possible implementation cannot exceed the total amount of resource available and that each interrupt handler can at most be allocated to one possible implementation.

$$\textbf{assert} \bigwedge_{h \in H} \left(\bigoplus_{pi \in X(h)} \left(\vec{h}\ [pi, h] \right) \oplus \bigwedge_{pi \in X(h)} \neg \vec{h}\ [pi, h] \right) \tag{4.3}$$

$$\textbf{assert} \bigwedge_{r \in R} \left(r.q \geq \sum_{pi \in T(r)} \left(\vec{r}\ [pi, r] \right) \right) \tag{4.4}$$

Here $X : H \rightarrow \mathcal{P}(PI)$ is a function returning the set of possible implementations that requires a given handler and $T : R \rightarrow \mathcal{P}(PI)$ is a function returning the set of possible implementations which requires some quantity of a given resource.

Collectively, the three assertions 4.2, 4.3 and 4.4 encodes the constraints on the decision variables but they do not comprise an optimization problem. By adding additional weighted assertions modeling the cost of picking a particular

possible implementation as the actual implementation we get an optimization problem. We will consider two slightly different ways of modeling the optimization problem. First, we consider a scenario where each possible implementation is associated with a cost. In Yices MAXSMT, this can be modeled by including a weighted assertion of the following form for each possible implementation *pi*:

$$\textbf{assert_w}\left(\neg s[pi],\ cost_{pi}\right) \tag{4.5}$$

where $cost_{pi}$ is the cost associated with choosing *pi* as an actual implementation. Alternatively, we may use the same cost for all possible implementations based the same implementation scheme. This can be done by adding a weighted assertion of the following form for each channel:

$$\textbf{assert_w}\left(\neg\left(\bigvee_{pi \in PI(ch,is)} s[pi]\right),\ cost_{is}\right) \tag{4.6}$$

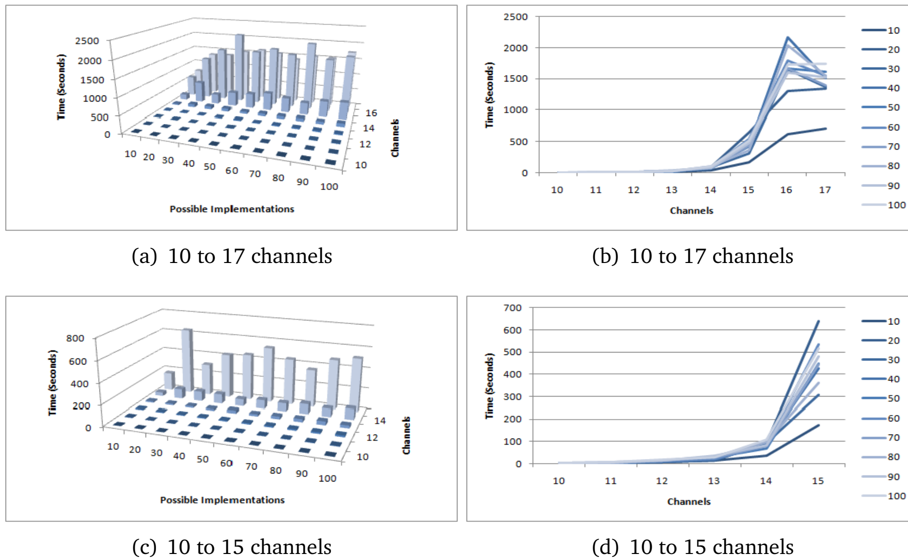
where *PI* is a function returning all possible implementations of channel *ch* based on implementation scheme *is* and $cost_{is}$ is the cost of choosing an actual implementation based on the *is* scheme.

Obviously, 4.5 provides the most flexibility but it is also computationally more demanding than 4.6 because, in general, the number of possible implementations will out number the set of implementation schemes many fold.

4.4.2.3 Experiments

The primary motivation for using MAXSMT for optimization is its expressiveness. The encoding of the optimization problem is, as can be seen, straightforward in MAXSAT. Also, MAXSAT is not a meta heuristic and will find the optimal solution. Unfortunately, MAXSMT is also computationally very demanding with a worst-case execution time that increases exponentially with the size of the problem. The MAXSMT problem is only supported by a handful of solvers. Most of these solvers are proof-of-concept tools and does not meet the requirements for this project. The Yices solver was chosen because it was the only MAXSMT solver that supports bit vectors and because it provides an API enabling an easier integration with the `xSRM` framework.

In an attempt at quantifying how large problems can be handled with Yices MAXSMT using the described encoding, a set of small experiments have been carried out. In the experiments, the time required by Yices to solve problems of increasing complexity has been measure. A problem consists of *n* channels each of which has *m* possible implementations that does not require any resources.



(a) 10 to 17 channels

(b) 10 to 17 channels

(c) 10 to 15 channels

(d) 10 to 15 channels

Figure 4.11 – Measured execution time. The measurements depicted by the graphs can be found in appendix D.

The graphs of Figures 4.11(a) and 4.11(b) shows the execution time measured for $m \in [10, 17]$ and $n \in [10, 100]$ using the cost per possible implementation encoding. The data forming the basis for the graphs can be found in appendix D. The problems with 16 and 17 channels all ran about 20-30 minutes before Yices reported the problem as being "undefined". This value is returned when Yices gives up trying to find a solution. When this happens can be controlled indirectly by means of two parameters "max conflicts in MAXSMT iterations" and "max number of iterations in MAXSMT". For the experiments, they were both arbitrarily set to 100000. To check that this was indeed the problem, another experiment with 17 channels and 20 possible implementations were run where the value of both parameters were set to 1000000. This experiment returned the value true, meaning that a solution was found, after 49 minutes and 57 seconds. Another test with 20 channels each with 20 possible implementations took 10 hours and 35 minutes before Yices gave up.

Figures 4.11(c) and 4.11(d) shows the measured execution time for the problems where a solution was found. The figure clearly shows that the execution time is exponential in the number of channels. Furthermore, the execution time quickly ramps up over a quite small interval meaning that the approach is sensitive to even small increases in complexity. The execution time for problems with less than 11 channels cannot be measured accurately because of the timer

resolution.

The variance in the data is believed to be due to ”random” restarts and other average-case optimizations employed by Yices. It has, however, not been possible to verify this due to the lack of detailed documentation of the Yices tool in general and its MAXSMT features in particular.

The experiments shows that the MAXSMT approach is only capable of handling relatively small problems. One thing that must be kept in mind though is that the efficiency of the solver is very much dependent on how the effect of the average case optimizations and, more importantly, on minimizing the search space by leaning from conflicts. Since the choice of implementation for the channels in the problems of the experiment does not have any interdependencies little can be learned and thus the search space cannot be minimized. In this sense, the problems of the experiment could be considered the worst possible problems in regards to the expected execution time of the solver. Again, this is purely speculative as a description of the MAXSMT features of Yices is not available.

The conclusion is that Yices MAXSMT is not a practical way of solving the optimization problem as it scales horribly. All of the experiments used the cost per implementation scheme encoding. The alternative, cost per possible implementation, has significantly more weighted assertions and other experiments (not included here) show that this causes a significant increase in the execution time. A more practical approach to solving the optimization problem could be to employ the use of meta heuristic methods such as simulated annealing [59], tabu search [41] or evolutionary algorithms [14]. Using such methods the computation time can be bounded at the cost a getting a potentially sub-optimal result. This will most likely be acceptable for the purpose of the procedure.

4.5 Case Study: MJPEG

To demonstrate the procedure we will use a realistic, but still relatively simple, application. The application is an MJPEG encoder that has been used extensively in the literature for demonstrating similar and related procedures, tools and methodologies involving KPN’s [63, 78, 34, 91]. The MJPEG application will be mapped to a custom multicore hardware platform which will exercise many of the implementations schemes described earlier. It is important to note that the focus of this case study is to show the versatility of our procedure emphasizing the analysis capabilities of the Service Relation Model. This focus has motivated the choice of platform instance and mapping rather than attempting

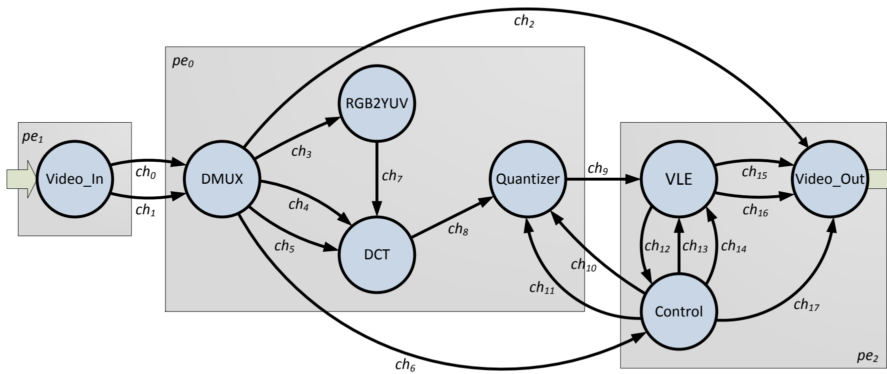


Figure 4.12 – The MJPEG encoder process network. The mapping of the processes to the processors of the platform depicted in figure 4.13 is indicated using the shaded regions.

Ch.	Source	Sink	C/C++ Type	Size	Depth
0	Video_in	DMUX	`TBlockData`	256	2
1	Video_in	DMUX	`THeaderInfo`	16	2
2	DMUX	Video_out	`TFrameSize`	8	10
3	DMUX	RGB2YUV	`TBlockData`	256	10
4	DMUX	DCT	`TBlockData`	256	10
5	DMUX	DCT	`TBlockType`	4	10
6	DMUX	Control	`TNumOfBlocks`	4	10
7	RGB2YUV	DCT	`TBlockData`	256	10
8	DCT	Quantizer	`TBlockData`	256	10
9	Quantizer	VLE	`TBlockData`	256	10
10	Control	Quantizer	`TCommand`	4	10
11	Control	Quantizer	`TQTables`	256	10
12	VLE	Control	`TStatistics`	2060	2
13	Control	VLE	`TCommand`	4	10
14	Control	VLE	`THuffTables`	4112	2
15	VLE	Video_out	`TBitStreamPacket`	4	10
16	VLE	Video_out	`TPacketFlag`	4	10
17	Control	Video_out	`TTablesInfo`	2448	2

Table 4.1 – Channels of the MJPEG application.

to optimize the implementation in terms of performance.

Figure 4.12 shows a graphical representation of the MJPEG application. The application consists of 8 processes and 18 channels. Table 4.1 shows additional information about the channels.

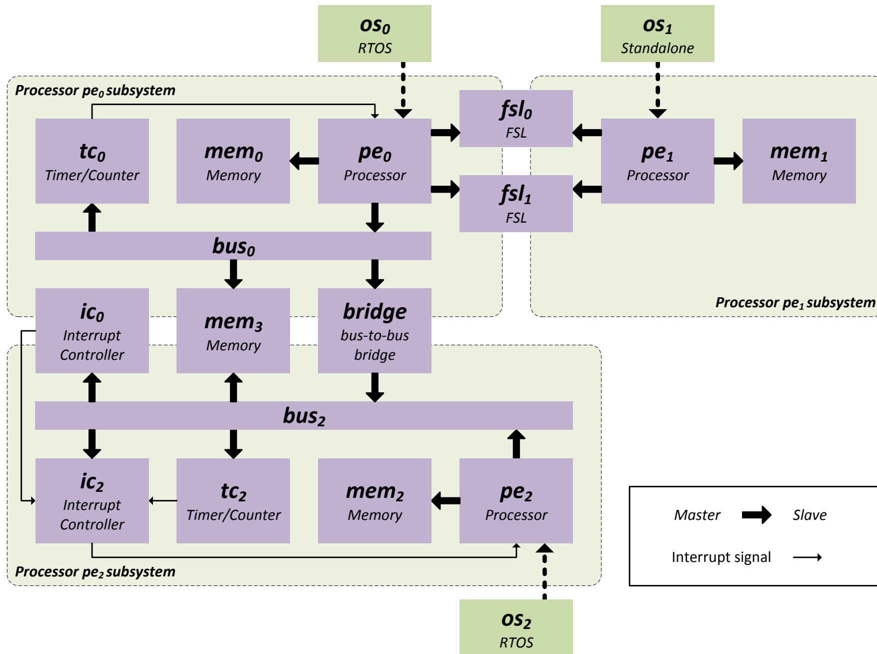


Figure 4.13 – Schematic of the MJPEG multicore platform.

4.5.1 The Platform

Hardware. Figure 4.13 shows a schematic of the target platform, created using Xilinx Platform Studio, we want to map the MJPEG application to. The schematic only contains the most important components, several interface controller have been omitted. The platform is divided into three interconnected sub-systems (ss_0, ss_1, ss_2) each of which contains a single Microblaze 32-bit RISC processor (pe_0, pe_1, pe_2). Each processor is connected to a local memory (mem_0, mem_1, mem_2) that cannot be accessed from anywhere else. Besides the processor and the local memory, sub-system ss_0 also contains a bus bus_0 and a bus-mounted timer/counter tc_0 that may generate interrupts for the processor pe_0. Sub-system ss_2 contains a bus bus_2, a timer/counter tc_2 and two interrupt controllers ic_0 and ic_2 who's purpose will be explained momentarily.

The processors of sub-systems ss_0 and ss_1 are connected point-to-point by means of two fast simplex links (fsl_0, fsl_1). Both links are configured so that pe_1 acts as master and pe_0 as slave meaning that it is only possible to transfer data in one direction from pe_1 to pe_0. Sub-systems ss_0 and ss_2 are connected by

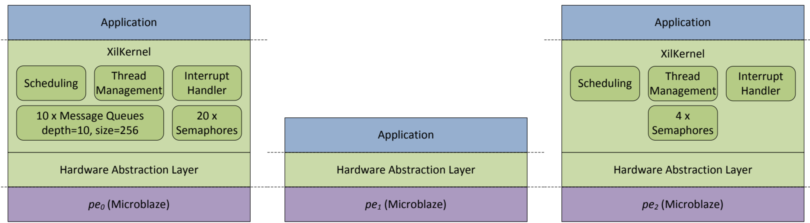


Figure 4.14 – The software stacks of the three processors pe_0, pe_1 and pe_2 in the MJEPG platform. The operating system of pe_0 is configured with support for 20 semaphore objects and 10 x message queues with a depth of 10 and token size of up to 256 bytes. The operating system of pe_2 is configured with support for 4 semaphore objects.

means of a shared memory (connected to both bus_0 and bus_2) and a bus-to-bus bridge enabling processor pe_0 to access bus_2 and its peripherals.

The purpose of the two interrupt controllers ic_0 and ic_2 is to provide processor pe_0 with the ability to generate interrupts on processor pe_2. Each interrupt controller consists of a number of interrupt inputs and a handful of control registers accessible through the bus slave interface. One of these control registers can be used to signal interrupts via software (i.e. by asserting the bit that corresponds to the interrupt source) provided that the controller is in a special ”software test” mode. In this mode, all the actual (physical) interrupt inputs are, however, ignored. Using two cascaded interrupt controller we are able to have both software generated and physical interrupts. The controller ic_0 is in the ”software test” mode and can be used to generate interrupts on one of the inputs of ic_2 which is in ”normal mode”. Physical interrupts are needed because tc_2 must be able to interrupt processor pe_2. The number of interrupt inputs on the interrupt controller ic_0 has been limited to 4.

Software. The software part of the platform is depicted in Figure 4.14. All the software stacks include a low-level hardware abstraction layer in the form of a set of C files providing easier access to the underlying processor and connected peripherals. This hardware abstraction layer is automatically generated by the Xilinx tool Libgen (”Library Generator”) for each processor in the platform instance. For two of the processors, pe_0 and pe_2, the `Xilkernel` operating system is used which provides the service of basic time-sliced multitasking. In addition to multitasking, each operating system can be configured to also include several other services such as message queues and semaphores. The operating system of processor pe_0 is configured with both message queues and semaphores where as the operating system of processor pe_2 is only configured with semaphores.

The number of such resources provided by each operating system is also shown in figure 4.14. The processor pe_1 runs "standalone" meaning that it does not have an operating system and thus can only host a single process.

Mapping. The mapping of the processes to the points-of-contact's in the platform was shown in Figure 4.12. Notice that the chosen mapping is feasible meaning that each channel should have at least one possible implementation according to one of the implementation schemes and that multiple processes are only mapped to points-of-contacts that can actually host more than one process.

The table below shows the cost associated with choosing an implementation based on each of the implementation schemes:

Scheme	Cost
$\mathtt{IS}_0$ Synthesized, Polling only	50
$\mathtt{IS}_1$ Synthesized, Two-way notification via interrupt	25
$\mathtt{IS}_2$ Synthesized, Polling+notification	30
$\mathtt{IS}_3$ Synthesized, Polling+notification	30
$\mathtt{IS}_4$ Synthesized, Two-way notification via function invocation	20
$\mathtt{IS}_5$ Platform-provided, FIFO, 32-bit, No Blocking	10
$\mathtt{IS}_6$ Platform-provided, FIFO, Unlimited, Blocking	15

The cost associated with each of the implementation schemes is to be considered an abstract measure that could refer to a concrete cost in terms of a metric or a combination of metrics such as energy, memory and/or performance. As far as the procedure and the case study is concerned, the cost is used solely for prioritization.

4.5.2 Results

Table 4.2 shows the outcome of the analysis portion of the procedure. For each channel, the number of possible implementations based on each of the seven implementations schemes are listed. All channels with the exception of channels ch_0 and ch_1 have at least one possible implementation using the synthesized memory channel scheme based on polling $\mathtt{IS}_0$. The channels with two possible $\mathtt{IS}_0$ implementations are all channels where both the reader and writer processes are located on one of the pe_0 or pe_1 processors. Both processor have, in addition to their local memories, also access to a shared memory. No channels have any possible implementations using implementation scheme $\mathtt{IS}_4$ (two-way notification via interrupts). Channels ch_0 and ch_1 can only be implemented using

	ch_0	ch_1	ch_2	ch_3	ch_4	ch_5	ch_6	ch_7	ch_8	ch_9	ch_{10}	ch_{11}	ch_{12}	ch_{13}	ch_{14}	ch_{15}	ch_{16}	ch_{17}
$\texttt{IS}_0$	0	0	1	2	2	2	1	2	2	1	1	1	2	2	2	2	2	2
$\texttt{IS}_1$	0	0	0	2	2	2	0	2	2	0	0	0	2	2	2	2	2	2
$\texttt{IS}_2$	0	0	0	0	0	0	0	0	0	0	4	4	8	8	8	8	8	8
$\texttt{IS}_3$	0	0	4	0	0	0	4	0	0	4	0	0	8	8	8	8	8	8
$\texttt{IS}_4$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\texttt{IS}_5$	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\texttt{IS}_6$	0	0	0	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0
total	1	2	5	5	5	5	5	5	5	5	5	5	20	20	20	20	20	20

Table 4.2 – The number of possible implementations per implementation scheme of each of the channels in the MJPEG application. The choice of the actual implementation of each channel is indicated by the shaded cells.

Resource	Type	Channel (usage)	Used/Available
$fsl_0.Queue$	`RCQueue<512,4>`	$ch_0(1)$	1/1
$fsl_1.Queue$	`RCQueue<32,4>`	$ch_1(1)$	1/1
$OS_0.MessageQueue$	`RCQueue<10,256>`	$ch_3(1)$, $ch_4(1)$, $ch_5(1)$, $ch_7(1)$, $ch_8(1)$	5/10
$mem_0.Memory$	`RCMemory`	*none*	0/8192
$mem_1.Memory$	`RCMemory`	*none*	0/8192
$mem_2.Memory$	`RCMemory`	$ch_{12}(6188)$, $ch_{13}(52)$, $ch_{14}(12344)$, $ch_{15}(52)$, $ch_{16}(52)$, $ch_{17}(4904)$	23588/32768
$mem_3.Memory$	`RCMemory`	$ch_2(88)$, $ch_6(48)$, $ch_9(2568)$, $ch_{10}(52)$, $ch_{11}(2824)$	5580/8192
$OS_0.Semaphore$	`RCSemaphore`	$ch_2(1)$, $ch_6(1)$, $ch_9(1)$	3/20
$OS_2.Semaphore$	`RCSemaphore`	$ch_{16}(2)$, $ch_{17}(2)$	4/4
$poc_2.IHandler_0.IHandler$	IHandler	$ch_2(1)$	1/1
$poc_2.IHandler_1.IHandler$	IHandler	$ch_6(1)$	1/1
$poc_2.IHandler_2.IHandler$	IHandler	$ch_9(1)$	1/1
$poc_2.IHandler_3.IHandler$	IHandler	$ch_{12}(1)$	1/1

Table 4.3 – Resource Usage

the platform provided implementation scheme $\texttt{IS}_5$. Channel ch_1 can be implemented using both of the FSL FIFO's where as ch_0 can only be implemented using one of them because the other one does not meet its minimum size/depth requirements.

The number of possible channels in the example have deliberately been kept low to ensure that the resulting optimization problem can be solved relatively quickly.

The choice of the actual implementation of each channel is also shown in Figure 4.2. The shaded cells shows which implementation scheme the chosen actual implementation of each channel belongs to. Table 4.3 shows the distribution of resources of the platform between the chosen actual channel implementations.

There are a total of six channels between processes mapped to pe_2. Four of these channels (ch_{12}, ch_{13}, ch_{14}, ch_{15}) have been chosen for implementation using the memory-backed polling implementation scheme $\texttt{IS}_0$ and the other two (ch_{16}, ch_{17}) using the two-way notification via function invocation implementation scheme $\texttt{IS}_4$. The limited number of available semaphores at pe_2 means that at most two channel can be implemented using $\texttt{IS}_4$. The buffer and control of all six channels have been placed in the local memory mem_2 of pe_2. This makes good sense as the alternative, placing them in the shared memory mem_3, would, all things equal, lead to a less optimal solution. This is purely incidental, however, as the alternative would also have been a valid solution.

The optimization problem used to determine the actual implementation has 18 channels and 173 possible implementations. Solving the optimization problem took 22 minutes and 40 seconds (1340 seconds). Notice that Yices required significantly less time for solving this optimization problem compared to the closest of the synthetic problems of the experiment from section 4.4.2.3. This strengthens our belief that the average case running time of Yices is substantially less than what the experiment with the synthetic examples showed.

The case study demonstrates the capabilities of the proposed procedure and shows the usefulness of the Service Relation Model as a means for analyzing complex systems.

4.6 Related Work

In this section, we will briefly compare our procedure to other approaches to automated code and abstraction layer generation described in the literature.

In [94] the authors present a programming model called Task Transaction Level (TTL). TTL is based on a process network MoC where tasks (processes) communicate with each other using bounded FIFO buffers (channels). In their approach, the API implementing the TTL interface is manually implemented bottoms-up on a per-platform basis. The channel implementation scheme used by our tool is inspired by the scheme used in the TTL implementation described in [94]. TTL includes a number of different communication and synchronization

primitives in addition to the (simple) blocking read and write supported by our tool. Implementing TTL for a platform is essentially a bottoms-up manual undertaking where as our procedure is a top-down automatic approach. TTL is mention here because it has served as a source of inspiration for our work.

In [27] the authors propose a design flow for implementing applications given as KPN's onto heterogeneous multi-processor systems based on the use of the Metropolis [13] tool. The proposed design flow is divided into four steps. In the first step, reconfiguration is applied to the platform and the application. For the platform reconfiguration means choosing a concrete platform from a set of available platforms. For the application reconfiguration means applying clustering to the process network. In the second step, the application is mapped to the platform. This is achieved by solving an optimization problem. In the third step, memory and buffer resources are allocated to the channels of the process network. In the final step, potential run-time deadlocks are addressed using runtime detection and resolution strategies. As we understand it, the proposed design flow is hugely a manual undertaking. Metropolis supports the design flow by providing a formal representation of both application and platform and by providing simulation data used to make decisions.

The aim of the *SynDEx* tool [80] is to generate optimized implementations of dataflow-based applications from descriptions of an algorithm and an architecture (platform). SynDEx supports a particular methodology for distributed real-time processing called the *Adequation Algorithm Architecture* (AAA) methodology, [86, 42]. Adequation means "efficient mapping" which illustrates the primary focus of the tool: to determine an optimal static or offline mapping and scheduling of an algorithm onto a platform. The result of applying the tool is a file for each processing element, called an executive, containing macro code that may be expanded into code suitable for that particular processing element and its interconnects. The executive of a processing element consists partly of macros describing the part of the application mapped to that processing element and partly of reusable macros for doing computation and communication on the processing element. Semaphores are used to ensure that the implementation is guaranteed to be dead-lock free. The model of architectures used by SynDEx are much more detailed than the Service Relation Model descriptions used by our tool. The extra detail is primarily needed to support reasoning about non-function aspects – something which our procedure does not consider. The SynDEx tool does not support as wide a range of different implementations as our procedure.

In [44] Guerin et al. proposes a design flow and associated tool set for generating application software for heterogeneous multi-processor systems. The flow takes as input a SimuLink application and a mapping and produces C code for each processor in the platform. The tools are based on a component-based

approach where an API is used to de-couple the application from a library of different operating systems, communication schemes and hardware abstraction layers. The proposed design flow does not employ analysis to determine the capabilities of the platform. Instead, the choice of operating systems and communication schemes and other design decisions must be made by the designer and is part of the input. The components of the library are dependent on the platform.

The *StepNP* system-level exploration platform for network processors [75] and its associated MultiFlex multi-processor programming environment [76] is an example of a combination of platform and tool that generates abstraction layers. The tool is capable of generating abstraction layers suitable for a symmetric multi-processing (SMP) model using shared memory and a distributed system object (DSOC) message passing programming model. This approach is limited by the underlying assumption that all processors in the system are connected to the same central interconnect. Also, it relies on platform-provided services in the form of hardware accelerators for message passing and task scheduling.

The tool *Embedded System-level Platform synthesis and Application Mapping* (ESPAM), part of the Daedalus design flow [90, 71], spans the entire design process from application and platform specification to FPGA implementation [70]. The tools very elaborate front-end allows applications to be specified in a restricted subset of the C programming language. ESPAM can extract possible parallelism, in the form of a KPN, from a sequential C program. Given the KPN, the tool is capable of determining an upper bound on the buffer size for each channel in the network such that deadlocks will be avoided without the use of detection and recovery mechanisms. The resulting process network can be mapped to multi-processor platforms also created using ESPAM's platform specification and generation tools. The tool allows the designer to assemble multi-processor platforms using a fairly small set of predefined components. Moreover, the tool only supports a couple of different platform topologies. ESPAM is an excellent example of a tool where limitations in the supported topologies and components are used to justify assumptions needed to simplify software synthesis tasks.

In [6] the authors presents a model based design methodology for software synthesis using a tool set called Embedded Systems Environment (ESE). The methodology starts with an application given as a set of C processes communicating via abstract channels mapped to a platform given as a netlist of system-level IP cores. A transaction level model is generated to validate the communication between processes mapped to different cores. This model is further refined into a Pin-Cycle accurate model which includes the C code realizing the communication. The authors reports a productivity gain of over 300 percent as a result of using automated software synthesis. This approach is in many ways

similar to ours but there are still a few notable differences. The model of a platform used is somewhat more limited than the Service Relation Model and, as a consequence, cannot be used to express as broad a range of different platforms. Another difference is that they do not support the concept of platform provided channels.

4.7 Discussion & Summary

In this chapter, we have presented our procedure for generating an abstraction layer implementing the communication infrastructure of an application modeled as a process network. At the heart of our procedure is the Service Relation Model and its associated analysis method for analyzing the flow and availability of services in systems composed of components. We have shown how information retrieved from analyzing a platform described using the model can be used for deriving concrete channel implementations based on abstract implementation schemes.

The procedure presented in this chapter is a vastly improved version of the procedure presented in a previous publication of ours [89]. The procedure of the publication supported only the memory-backed polling implementation scheme and thus avoided many of the problems associated with resources. One contribution of that paper, that has not been mentioned here, is a method for dealing with a mapping that does not have an immediate realization because at least one channel does not have an actual implementation. This will be the case if the two processors to which a pair of communicating processes are mapped cannot communicate or if there are insufficient with resources available. If this is the case then it may be possible to transform the process network by adding additional, appropriately mapped, processes and channels that implements the problematic channels indirectly via other processors. The necessary transformation can be done automatically using analysis information gathered using the Service Relation Model and information about how channels can be implemented (i.e. implementation schemes). We see no principle difficulties in extending the method to support the improved procedure presented here but it has, for the time being, been left as future work.

The procedure only considers processes implemented as software. An obvious extension would be to allow processes to be mapped to dedicated hardware as well. This would require adding another component type for representing hardware points of contact and new implementation schemes.

CHAPTER 5

Automated Design Generation

In chapter 3, we showed how the service relation model could be used to check the consistency of a platform. In this case the problem was to determine if a given network of components were consistent. In this chapter, we will investigate the possibility of reversing the problem and asking: given an inconsistent model what needs to be done to make it consistent? An inconsistent model can possibly be made consistent by instantiating, configuring and connecting new components to the inconsistent design.

In this chapter, we will present a procedure for automatically constructing a consistent design on the basis of an inconsistent design. The procedure is based on the use of an SMT solver capable of evaluating MAXSMT problems.

5.1 Introduction

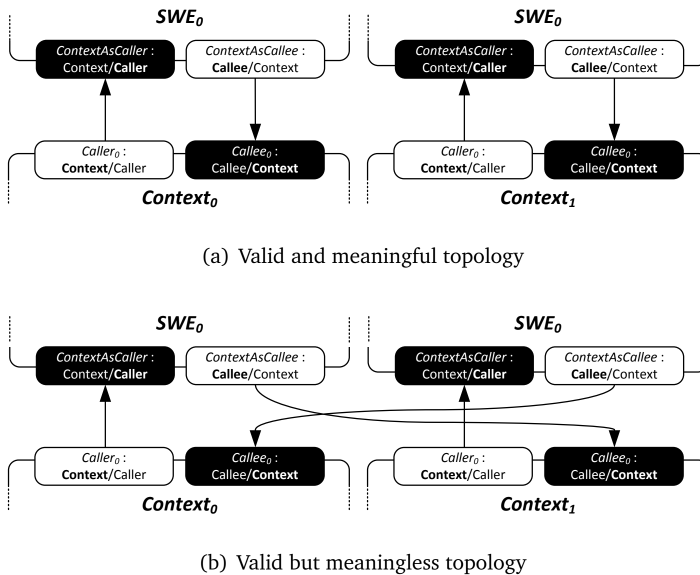
Embedded systems are typically organized as a stack of layers. Each layer in such a stack provides services to the upper layers and uses services from the lower layers. At the bottom we typically have the hardware and on top of that the various software layers. The application layer, at the top of the stack, is unique in that it does not provide any services to the other layers. The role

of the application layer is to impose requirements on the lower layers – the lower layers must provide the application layer with the services it requires. The lowest layer is characterized in that it is self-contained and does not require any services provided by other layers.

In chapter 4, a three layer view was presented consisting of 1) the platform, 2) an abstraction layer and 3) an application modeled as a process network. Here the application layer required the abstraction layer to provide the communication infrastructure of the application. The platform provided the upper layers with services such as memory access, multi-processing and synchronization. In chapter 4, the platform was assumed to be given and non-configurable and the architecture of the abstraction layer only allowed for a small degree of customization. In most cases, the platform does provide some configuration possibilities. This is especially true if the platform also contains software middelware. In order to make use of this potential for customization a more system-oriented approach must be adopted where all layers are considered simultaneously. This adds extra dimensions to the complexity of the problem and introduces a number of new challenges – which are the focus of this chapter.

In the Service Relation Model, a layer can be represented as a partial design (i.e. a design containing unconnected interfaces). A design representing an entire stack can be created by properly connecting the unconnected interfaces of the partial designs representing the individual layers of the stack to each other. There is no explicit representation of layers in the Service Relation Model. The partial design representing the top level application layer is characterized by being non-configurable and inconsistent with respect to the Service Relation Model since it will require services provided by the layers beneath it. The design representing the whole stack must, however, be consistent and thus the problem addressed in this chapter can be summarized as: Given a partial and inconsistent design what must be added to the design to make it consistent? Here, the input design is not limited to be a representation of the application alone, it may contain descriptions of non-optional parts of other layers as well. In the context of the service relation model, a solution to this problem consists of **a**) an allocation of (additional) components from a library, **b**) a configuration for each component and **c**) a topology relating the given design and the allocated components to each other. The input design may contain parts that are configurable in which case a configuration for the design is also part of the solution.

In order to automatically construct a solution to the problem presented above, we present a procedure based on the use of a MAXSMT solver supporting the theories of arithmetics and bit vectors.



(a) Valid and meaningful topology

(b) Valid but meaningless topology

Figure 5.1 – Example showing the need for topology restrictions.

5.1.1 Interface Groups

A shortcoming of the Service Relation Model with respect to automated design generation is that it does not provide a way to restrict the topology beyond ensuring that interfaces are properly connected. In some cases, a component has multiple interfaces that must be connect in a specific way in order for the model to make sense. For example, consider the simple model shown in Figure 5.1(a). The model contains two context components representing the execution context of two different processors and two software entities connected to each of the contexts. A software entity and a context is connected by means of two service exchange relations (`Context/Caller` and `Callee/Context`) used for importing and exporting services representing functions. In order to make sense, the two interfaces of a software entity must be mapped to the *same* context. This restriction, however, is not formally captured in the service relation model and, consequently, the model shown in figure 5.1(b) is also valid although it makes no sense.

To avoid having the procedure produce useless results like that of Figure 5.1(b) a way to restrict the connectivity of components is needed. To accomplish this we introduce the simple concept of *interface groups*. An interface group is a set of interfaces of a component that must be connected to the same component.

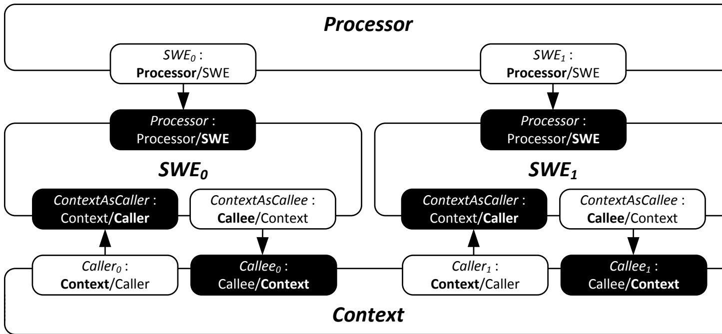


Figure 5.2 – Example of a topology that cannot be expressed using interface groups.

Definition 5.0 (Interface Group, g**,** G**)** An interface group g is a set $g = \{i_0, i_1, \ldots i_n\}$ of interfaces belonging to the same component. An interface can at most belong to one interface group. □

By placing the two interfaces of the software entities in the same group we can ensure that they are always connected to the same context and thus exclude solutions like that of figure 5.1(b). In a future version of the Service Relation Model, it may make sense to replace the concept of interface groups with a similar, but more initutive, concept of composite interfaces.

Interface groups is not an ideal solution to the problem because it can only be used to specify restrictions on the immediate connectivity of a component. As an example of a restriction that cannot be specified using interface groups consider the model depicted in Figure 5.2 of two software entities mapped to the same processor. Both of the software entities are related to the processor and a context components. Because the context components represents the execution context on a specific processor it must be the case that all software entities related to it must also be related to the *same* processor. This kind of restrictions cannot be expressed using the concept of interface groups.

5.2 Procedure Overview

It is the purpose of this section to give an overview of the design generation procedure. The details of the procedure will be presented in the following section.

The inputs to the procedure consists of 1) a model representing the a partial

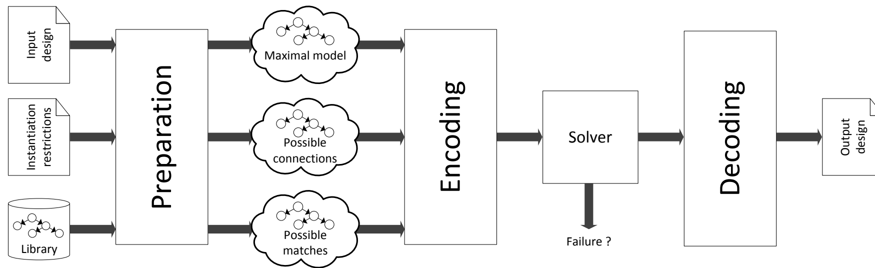


Figure 5.3 – Overview of the design generation procedure.

and inconsistent design, 2) a library of component templates and 3) a set of instantiation restrictions. The instantiation restrictions are used to place bounds on the solution space which may otherwise be infinite. Instantiation restrictions will be properly introduced and discussed later.

The idea behind the procedure is to formulate an optimization problem on the basis of the concepts of the service relation model, the input design and the library. The free variables in the problem are 1) the number and types of components added to the design, 2) the number and types of options included in each component, 3) the values of parameters and, last but not least, 4) the choice of connections between unconnected interfaces. The objective of the optimization is to minimize costs associated with adding new components and options to the design. The optimization problem is encoded as a MAXSMT problem targeting the Yices SMT solver [32]. The result of applying the procedure is either a consistent design containing the input design or failure. The procedure may fail if one or more of the inconsistencies of the input design cannot be satisfied by any combination of components instantiated from the templates of the input library.

The procedure is carried out on a modified representation of models that includes several of the concepts found in the library and design domains. More specifically, the used model representation explicitly represents the configurable portions of a design such as options and parameter values that are not be part of the model domain as presented in section 2.3.1. This is needed in order for the procedure to reason about the configurable portions of a design.

The procedure can be broken into three steps, as shown in figure 5.3, each of which will be described in detail later in sections 5.3, 5.4 and 5.5.

Step 1 (Preparation) The purpose of the preparation step is to compute information needed by the encoding into MAXSMT. The information computed

in the preparation step consists of a service relation model called the *maximal model*, a set of so called *possible connections* and another set of *possible matches* similar to those used previously in section 3.5.

The *maximal model* is a preliminary representation of the final result that includes as many components and options as allowed by the instantiation restrictions. The maximal model can be thought of as the largest possible model, in terms of the number of component and option instances included, that may be returned by the procedure. The maximal model contains relations corresponding to those found in the input model but does not contain any information about how the added components are connected and, as a result, the maximal model will contain a number of unconnected interfaces. The set of *possible connections* is an enumeration of the connections (relations) that can be made between the unconnected interfaces of the maximal model taking relations and roles of the interfaces into account.

Step 2 (Encoding) In the next step, a MAXSMT problem is formulated on the basis of the maximal model, the set of possible connections and the set of possible matches. The problem contains a number of variables that describes the free dimensions of the problem. The aim of the solver is to find a satisfiable valuation of these variables that will minimize the cost associated with falsifying assertions representing the inclusion of components and options. The encoding of the problem is described in more detail later in section 5.4.

Step 3 (Solving & Decoding) In the last step of the procedure, the result produced by the solver is decoded. In principle, the solver may return one of two results: unsatisfiable or satisfiable. If the solver determines the problem to be unsatisfiable then the input design could not be made consistent with the components specified in the instantiation restrictions and the procedure terminates with error. If the solver determines the problem to be satisfiable then a solution that minimizes the total cost of the problem has been found. In this case, both the cost and the associated valuation of the variables may be extracted. Using the valuation of the variables associated with a satisfiable solution the desired result (i.e. the allocation, topology and configuration representing the consistent result) is constructed.

5.3 Preparation

5.3.1 Procedure Input

The inputs to the procedure consists of a partial (and inconsistent) Service Relation Model design, called the *input design*, a set of instantiation restrictions and a library of components.

The Library. The library contains descriptions of the component templates that can be used by the procedure. The part of a library that is considered by the procedure is limited by means of the instantiation restrictions.

The Input Design. The input design is a *design* specifying a partial and inconsistent service relation model of some system. Notice that the input is a design rather than a model. The design may have parts that are configurable by means of options. Such configurablity is defined using instantiation restrictions.

Instantiation Restrictions. There are two kinds of instantiation restrictions: component and option restrictions. Component instantiation restrictions are used to limit the type and number of components that will be considered by the procedure and, similarly, the option instantiation restrictions are used to limit the type and number of options that will be considered. Formally, the two kinds of instantiation restrictions are defined as:

Definition 5.1 (Component Instantiation Restriction, *cir*, *CIR***)** An instantiation restriction is a triple $ir = \langle t, max, cost \rangle$ where t is a component template in the library, $max \in \mathbb{N}_0$ the maximum number of instances of the component template that can be in the model and $cost \in \mathbb{N}$ is the cost of adding an instance of the template to the design. □

Definition 5.2 (Option Instantiation Restriction, *oir*, *OIR***)** An instantiation restriction is a triple $ir = \left\langle o^{\ell}, max, cost \right\rangle$ where o^{ℓ} is an option in the library, $max \in \mathbb{N}_0$ the maximum number of instances of the option that any component in the model can have and $cost \in \mathbb{N}$ is the cost of adding an instance of the option to a component. □

In the following, let $CIR = \{cir_0, cir_1, \ldots, cir_n\}$ denote the set of component instantiation restrictions and let $OIR = \{oir_0, oir_1, \ldots, oir_m\}$ denote the set of option instantiation restrictions passed as input to the procedure.

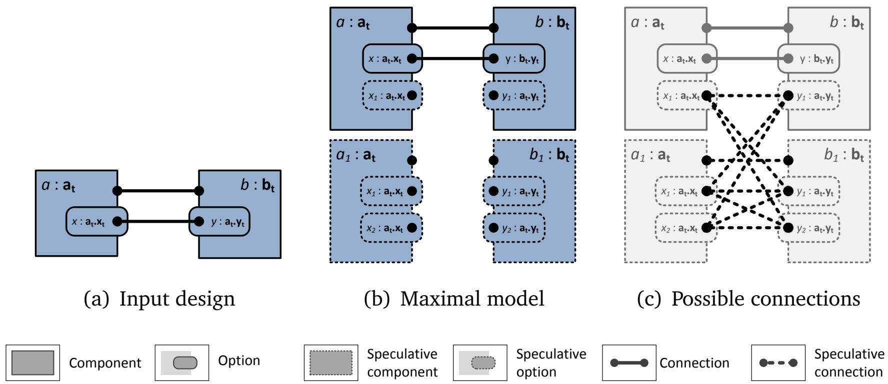
The cost associated with adding components and options to a design is primarily used to ensure that the result produced by the procedure does not contains any unnecessary components. If no cost is associated with adding components then results may be produced that contain unused components. A secondary use of cost is to prioritize between different choices of components and options that can be used to resolve the inconsistencies of the static model. In this case, the cost can be considered an abstract measure of some metric associated with including the component or option.

5.3.2 The Maximal Model

The first step in the preparation is to create the maximal model. The maximal model consists of the input design and a set of, so called, *speculative* components and options. The maximal model represents the largest possible model in terms of the number of components and option instances that will be considered. A speculative component or option is a entity that *may* be part of the final design.

The maximal model is created on the basis of the input design and the instantiation restrictions. Given a model representing the input design, the first step in creating the maximal model is to add additional speculative components. For each component instantiation restriction, a number of components corresponding to the maximum number instances allowed by the restriction are added to the model. These components are marked as being speculative. Speculative components are not connected to any other components in the model as determining their connectivity is part of the decision making handled by the solver.

Next, options are considered. The set of options included in each component is compared to the options mentioned in the option instantiation restrictions. If the template of a component contains an option mentioned in an option instantiation restriction then the number of included instances of the option in the component is compared to the maximum number of allowed instances in the restriction. If the number of included instances is less than the number of allowed instances then additional options are included in the component so that number of included instances matches the maximum number of allowed instances. All options added this way are marked as speculative.



(a) Input design (b) Maximal model (c) Possible connections

Figure 5.4 – Preparation. **a**: maximal model corresponding to the input design shown in **b** and the instantiation restrictions $CIR = \{\langle a_t, 10, 2\rangle, \langle b_t, 20, 2\rangle\}$ and $OIR = \{\langle a_t.x_t, 5, 2\rangle, \langle b_t.y_t, 3, 2\rangle\}$. **c**: set of possible connections.

Example 5.1 As an example of how the maximal model related to an input model and a set of instantiation restrictions, consider the simple input design of figure 5.4(a). The input design and the instantiation restrictions are given relative to a library containing two templates a_t and b_t. The mandatory option of template a and b both contains a single interfaces that are compatible with each other. Template a_t is configurable by means of a single option x_t and template b_t by means of the option y_t. As was the case for the mandatory options, both option x_t and y_t contains a single interface compatible with the interface of the other option.

The input design contains two components – one instance of template a_t called a and one instance of template b_t called b. A single instance of the option x_t is included in component a and, similarly, a single instance of option y_t is included in component b. As shown in Figure 5.4(a) the interfaces of the mandatory options of the two component instances are connected and so is the interfaces of the two non-mandatory options.

Figure 5.4(b) shows the maximal model corresponding to the input design and the instantiations restrictions:

$$CIR = \{\langle a_t, 10, 2\rangle, \langle b_t, 20, 2\rangle\}$$

$$OIR = \{\langle a_t.x_t, 5, 2\rangle, \langle b_t.y_t, 3, 2\rangle\}$$

■

5.3.3 Possible Connections

The next step of the preparations is to determine the possible connections between the unconnected interfaces of the components in the model. A possible connection is given as:

Definition 5.3 (Possible Connection, pc, PC) A possible connection pc is a pair $pc = \langle i_0, i_1 \rangle$ where $i_0, i_1 \in I_*$. A possible connection is valid only if $i_0.rel \neq i_1.rel$ and $i_0.role = i_1.role$ and if both i_0 and i_1 are unconnected. □

Computing the set PC_* for a maximal model m is simply a question of comparing the unconnected interfaces. Note that computing the set of possible connections can be computationally expensive for large models.

Example 5.2 The set of possible connections belonging to the maximal model of the previous example is depicted in Figure 5.4(a). ■

In the following, let $PC_* = \{pc_0, pc_1, \ldots, pc_n\}$ be the set of possible connections in the maximal model. Furthermore, let $PC : I_* \rightarrow \mathcal{P}(pc)$ be a function that maps an interface to the set of possible connections in PC_* that contains a reference to the interface.

5.4 Encoding

In this section, the encoding of the problem in MAXSMT is presented. The presentation will not cover all details of the encoding but only focus on the main ideas. For a more detailed presentation, we refer to the source code of the tool used for experimental purposes.

The encoding makes use of a number of variables organized into groups. Table 5.1 shows the most interesting groups. Again, a map-index notation will be used to refer to the actual variables of the groups. For example, $at[s]$ refers to the bit vector variable representing the set of services available at the service s.

The presentation of the encoding will be done as a presentation of a function $\mathcal{E}$ that maps concepts in the service relation model into SMT assertions. Some concepts give rise to multiple assertions in different parts of the encoding. For

Group	Type	Description
$con[i_0, i_1]$	boolean	Interfaces i_0 and i_1 connected
$con[i]$	boolean	Interface i is connected (not unconnected)
$inc[c]$	boolean	Component c is included in the result
$inc[o]$	boolean	Option o is included in the result
$at[s]$	bit vector	The set of services available at service s
$at[i]$	bit vector	The set of services available at interface i
$tid[i]$	integer	Target component id of interface i
$val_p[p]$	integer	The value of parameter p
$val_{rp}[rp, res]$	integer	The value of resource parameter p of resource res
$val_q[q]$	integer	The value of the quantity q
$\vec{u}[res, cl]$	integer	Quantity of resource allocated from res to cl

Table 5.1 – Groups of variables used in the encoding of the design generation problem in MAXSMT.

such a concept x, we will use the special notation $\mathcal{E}(x) := \ldots$ **assert** SMT_{expr} to show that the definition of $\mathcal{E}$ for the concept x is only partial.

5.4.1 Service Flow

The encoding of service availablity in the service flow is implemented using bit vectors. This is possible because the number of services in the maximal model is finite. The usual operations on sets (e.g. union, intersection, complement) can easily be expressed using bit vector operations. For clarity, however, the following presentation will use sets in place of bit vectors.

To ease the presentation of the encoding we will make use of a function $To : S_* \cup I_* \rightarrow \mathcal{P}(S_* \cup I_*)$ that given a service or an interface returns the set of services or interfaces that has the input service or interface as a predecessor node in the service flow graph of the model. Another function $Parent$: SRM concept $\rightarrow C \cup O$ is used to retrieve the immediate parent of concepts such as assertions, resources and resource claims.

Services. The set of services available at a service is simply the union of the service itself and the set of services available at each of the interfaces imported into the service:

$$\mathcal{E}(s) := \textbf{assert } at[s] = \{s\} \cup \bigcup_{i \in To(s)} at[i]$$

Interfaces. The encoding of interfaces is slightly more involved as the set of available services at an interface is dependent on which, if any, remote interface it is connected to. The encoding of an interface is split into two different encodings depending on whether or not the interface has any services exporting to it:

$$\mathcal{E}(i) := \ldots \begin{cases} \mathcal{E}_{i,1}(i) & \text{if } To(i) \neq \emptyset \\ \mathcal{E}_{i,2}(i) & \text{if } To(i) = \emptyset \end{cases}$$

The first of the two encodings deals with the case where at least one service exports to the interface. This implies that the interface is an object interface as service export is only allowed to have object interfaces as targets. In this case, the set of available services at the interface is the union of the services available at each of the services importing to the interface moderated by a requirement that the interface must be connected and the parent component must be included:

$$\mathcal{E}_{i,1}(i) := \textbf{assert} \texttt{ if } \neg con[i] \vee \neg inc[i] \texttt{ then } at[i] = \emptyset \texttt{ else } at[i] = \bigcup_{s \in To(i)} s$$

The next encoding deals with the case where no services exports to an interface. If the interface is an object then the set of services available must necessarily be the empty set because the flow of service through an interface is unidirectional from the object to the subject. If the interface is a subject interface then its service availablity set is the empty set if the interface is not connected or equal to the service availablity set of the remote interface to which the interface is connected:

$$\mathcal{E}_{i,2}(i) := \begin{cases} \textbf{assert } at[i] = \emptyset & \text{if } i.role = object \\ \textbf{assert } \neg con[i] \rightarrow at[i] = \emptyset & \text{if } i.role = subject \end{cases}$$

5.4.2 Topology

The encoding of the topology is divided into two parts: one part dealing with the relations in the maximal model corresponding to the relations found in the input design and one part dealing with the possible connections between unconnected interfaces.

Relations. For each relation r in the maximal model we simply assert that the set of services available at each of the connected interfaces must be equal:

$$\mathcal{E}_r(r) := \textbf{assert } at[r.i_1] = at[r.i_2]$$

Possible Connections. The connectivity of the unconnected interfaces in the maximal model are encoded using the set of possible connections that were computed previously in the procedure. Each possible connection pc is associated with a boolean variable $sel[pc]$ stating whether or not the connection has been *selected* as an actual connection (i.e. should be in the output design). For each possible connection, we assert that if the connection is chosen then the services available at each of the two interfaces represented by the possible connection must be equal:

$$\mathcal{E}_{pc}(pc) := \ldots \textbf{ assert } sel[pc] \rightarrow at[pc.i_0] = at[pc.i_1]$$

Unconnected Interfaces. For each interface i in the model, we assert that the interface is either connected or exactly one of the possible connections that i can participate in is selected:

$$\mathcal{E}_i(i) := \ldots \textbf{ assert } \neg con[i] \oplus \bigoplus_{pc \in PC(i)} sel[pc]$$

here $con[i]$ is a variable associated with an interface i stating whether or not the interface is connected.

Interface Groups. The encoding of interface groups is based on the use of unique integer id's associated with the components of the maximal model. In the encoding, each interface i in the model is associated with a variable $tid[i]$ that contains the id of the component to which it is connected. The special id zero is used for unconnected interfaces and does not identify any component in the model.

To ensure that the values of the target id's properly reflects the topology additional assertions regarding the possible connections and interfaces of the model are needed. For each possible connection, an assertion is added stating that if the possible connection is selected then the target id of each interface in the possible connection equals the id of the parent component of the other:

$$\mathcal{E}(pc) := \ldots \textbf{ assert } sel[pc] \rightarrow (tid[pc.i_0] = id(pc.i_1) \wedge id(pc.i_0) = tid[pc.i_1])$$

here $id : I_* \rightarrow \mathbb{N}_{>0}$ is a convenience function for retrieving the id associated with the parent component of an interface. For each interface i in the model, an assertion stating that if the interface is unconnected then its target id is zero is also added.

$$\mathcal{E}(i) := \ldots \ \textbf{assert} \ \neg con[i] \rightarrow tid[i] = 0$$

Using the target id's, the encoding of an an interface group g is encoded straightforward by an assertion stating that the target id's of all interfaces in the group must be the same:

$$\mathcal{E}(g) := \textbf{assert} \bigwedge_{i_0, i_1 \in g} tid[i_0] = tid[i_1]$$

5.4.3 Parameters

A parameter p is encoded using an integer variable $val_p[p]$. If the parameter belongs to a non-speculative option then its value is defined by means of an assertion:

$$\mathcal{E}_p(p) := \textbf{assert} \ val_p[p] = p.value$$

Components & Options. The components and options of the maximal model are treated differently in the encoding depending on whether or not the component or option is marked as speculative. Associated with each component and option is a boolean variable $inc[x]$ stating whether or not the component or option is part of the result. For a component or an option that is part of the static model, the value of the $inc[x]$ variable is asserted to be true meaning it is (must be) included in the result. For a speculative component or option, the value of the $inc[x]$ variable if left for the solver to decide. This is accomplished by means of a weighted assertion stating that if $val[x]$ is true of a component or option x then the cost specified in the instantiation restriction associated with the component or option is incurred.

$$\mathcal{E}(c) := \ldots \begin{cases} \textbf{assert_w} \ (\neg inc[c], cost(c)) & \text{if } c \text{ is speculative} \\ \textbf{assert} \ inc[c] & \text{otherwise} \end{cases}$$

$$\mathcal{E}(o) := \ldots \begin{cases} \textbf{assert_w} \ (\neg inc[o], cost(o)) & \text{if } o \text{ is speculative} \\ \textbf{assert} \ inc[o] & \text{otherwise} \end{cases}$$

$\mathcal{E}_\alpha : \textbf{Expr}_\alpha \rightarrow SMT$		
$\mathcal{E}_\alpha(e_0 \textbf{ and } e_1)$	:=	$\mathcal{E}_\alpha(e_0) \wedge \mathcal{E}_\alpha(e_1)$
$\mathcal{E}_\alpha(e_0 \textbf{ or } e_1)$	:=	$\mathcal{E}_\alpha(e_0) \vee \mathcal{E}_\alpha(e_1)$
$\mathcal{E}_\alpha(e_0 \textbf{ implies } e_1)$	:=	$\mathcal{E}_\alpha(e_0) \rightarrow \mathcal{E}_\alpha(e_1)$
$\mathcal{E}_\alpha(\textbf{not } e_0)$	:=	$\neg\, \mathcal{E}_\alpha(e_0)$
$\mathcal{E}_\alpha(sc \texttt{ @ } i)$	:=	$sc.S \cup at[i] \neq \emptyset$
$\mathcal{E}_\alpha(sc \texttt{ @ } s)$	:=	$sc.S \cup at[s] \neq \emptyset$

Table 5.2 – Encoding of alpha expressions

If a component is not included then all of its options are also not included:

$$\mathcal{E}(c) := \ldots \textbf{ assert } \neg inc[c] \rightarrow \bigwedge_{o \in c.O} \neg inc[o]$$

If an option is not included then all of its interfaces must be unconnected:

$$\mathcal{E}(o) := \ldots \textbf{ assert } \neg inc[o] \rightarrow \bigwedge_{i \in o.I} \neg con[i]$$

5.4.4 Assertions

Due to the expressiveness of MAXSMT, the encoding of Service Relation Model assertions in quite simple. A Service Relation Model assertion a is encoded by means of a single MAXSMT assertion:

$$\mathcal{E}(a) := \textbf{assert } inc[parent(a)] \rightarrow \mathcal{E}_\alpha(a)$$

The MAXSMT assertion is constructed by means of the function given in Table 5.2 for mapping alpha expressions into MAXSMT expressions. The implication ensures that the service relation model assertion should only be considered if the parent component or option of a is included.

5.4.5 Resources & Resource Claims

The encoding of resources and resource claims resembles the encoding used for the consistency checking procedure of section 3.5. Similar to the approach of section 3.5, the encoding of resources and claims makes use of flow variables

$\vec{u}[res, cl]$ representing the assignment of resource from a given resource *res* to a given claim *cl* and a pre-computed set of *possible matches*. In comparison with the encoding used for consistency checking, the computation of the possible matches only considers resource classes and does not take the where-expression of the claim into account. This leads to a slightly different definition of a possible match than the one used previously for consistency checking:

Definition 5.4 (Possible Match, pm, PM) A possible match pm is a pair $pm = \langle res, cl \rangle$ where $res \in RES_*$ a resource and $cl \in CL_*$ is a resource claim so that $res.rc \sqsubseteq cl.rc$. □

In the following, let $PM_* = \{pm_0, pm_1, \ldots, pm_n\}$ be the set of all possible matches in the maximal model. This set is easily computed by comparing the resource classes claimed by the resource claims in the model with the resource classes provided by the resources. Furthermore, let P_{claim} be a function for retrieving the set of claims that may be satisfied by a resource

$$P_{claim}(res) = \{cl : \langle res, cl \rangle \in PM_*\}$$

and P_{res} be a similar function for retrieving the set of resources that may satisfy a given claim.

$$P_{res}(cl) = \{res : \langle res, cl \rangle \in PM_*\}$$

Quantity Expressions. Both resources and claims include quantity expressions. In resources, quantity expressions are used to define the quantity of resource being provided and to define the value of any resource class parameters of the associated resource class. In resource claims, quantity expressions are used to define the quantity of resource being claimed. A quantity expression q is encoded using a integer variable $val_q[q]$ and an assertion defining its value:

$$\mathcal{E}(q) := \textbf{assert}\ val_q[q] = \mathcal{E}_{\text{qty}}(q)$$

The assertion is constructed by means of the function given Table 5.3 for encoding the expression. As can be seen, the encoding of the expression itself is rather trivial. Note that Yices only supports multiplication and division for pairs of operands where at least one of them is a constant.

Resources. A resource *res* is encoded using several assertions. The first assertion states that if the parent component or option of the resource is included

$\mathcal{E}_{\text{qty}} : \mathbf{Expr}_{\text{qty}} \rightarrow SMT$		
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ + } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) + \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ - } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) - \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ * } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) \times \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ / } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) / \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ > } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) > \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ >= } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) \geq \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ < } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) < \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ <= } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) \leq \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ = } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) = \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ != } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) \neq \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ and } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) \wedge \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ or } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) \vee \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(e_0 \texttt{ implies } e_1)$	=	$\mathcal{E}_{\text{qty}}(e_0) \rightarrow \mathcal{E}_{\text{qty}}(e_1)$
$\mathcal{E}_{\text{qty}}(\texttt{not } e)$	=	$\neg \mathcal{E}_{\text{qty}}(e)$
$\mathcal{E}_{\text{qty}}(\texttt{- } e)$	=	$-\mathcal{E}_{\text{qty}}(e)$
$\mathcal{E}_{\text{qty}}(parameter)$	=	$val[parameter]$
$\mathcal{E}_{\text{qty}}(constant)$	=	$constant$

Table 5.3 – Encoding of quantity expressions.

then sum of resource assigned from *res* to any claims must be less than or equal to the quantity of resource provided:

$$\mathcal{E}(res) := \ldots \ \textbf{assert}\ inc[parent(res)] \rightarrow val_q[res.q] \geq \sum_{cl \in P_{claim}(res)} \vec{u}[res, cl]$$

The next assertion states that if the parent component or option of the resource is *not* included then amount of resource from *res* assigned to any claims must be zero:

$$\mathcal{E}(res) := \ldots \ \textbf{assert}\ \neg inc[parent(res)] \rightarrow \bigwedge_{cl \in P_{claim}(res)} \vec{u}[res, cl] = 0$$

If the resource class of the resource is associated with any resource parameters the value of these are defined by means of assertions of the following form

$$\mathcal{E}(res, rp) := \textbf{assert}\ val_{rp}(res, rp) := val_q[rp.val]$$

Resource Claims. A resource claim cl is encoded differently depending on whether the claim can be satisfied by many resources or must be satisfied by exactly one resource. If cl can be satisfied by multiple resources ($cl.mp = many$) then we assert that the sum of the amount of resource assigned to cl from each resource equals the value of the quantity expression $cl.q$ representing the quantity of resource required by cl:

$$\mathcal{E}(cl) := \ldots\ \textbf{assert}\ inc[parent(cl)] \rightarrow val_q[cl.q] = \sum_{res \in P_{res}(cl)} \vec{u}[res, cl]$$

If, on the other hand, the claim must be satisfied by a single resource ($cl.mp = one$) then we assert that the amount of resource assigned to cl from each of the different resources with the exception of one that must provide exactly the amount of resource specified by the value of the quantity expression $cl.q$:

$$\mathcal{E}(cl) := \ldots \textbf{assert}\ inc[parent(cl)] \rightarrow \bigoplus_{res \in P_{res}(cl)} \left(val_q[cl.q] = \vec{u}[res, cl] \wedge \bigwedge_{z \in P_{res}(cl) \setminus res} \vec{u}[z, cl] = 0 \right)$$

The pre-computed set PM_* of possible matches used for the encoding does not take the where expressions of resource claims into account. The reason for this is that the value of a where expression depends on the topology of the model and because the defining the topology of the model is part of the procedure then it is not possible to pre-compute its value. The influence of the where expressions on the assignment of resource from resources to resource claims are encoded by means of a single assertion per possible match:

$$\mathcal{E}(pm) := \textbf{assert}\ \neg\mathcal{E}_\omega(pm.cl, pm.res) \rightarrow \vec{u}[pm.res, pm.cl] = 0$$

here $\mathcal{E}_\omega$ is the function given in table 5.4 for encoding the where expression. The assertion states that if the where expression is not satisfied for a given possible match $pm = \langle res, cl \rangle$ then the amount of resource from res assigned to cl must be zero.

Note that the meaning of the two is-available-through expressions in Table 5.4 differs from the meaning previously defined in section 3.3.2. The expressions used here are slightly more limited because it only takes the services the resource is directly exported to into account. In other words, these expressions can only be used to make tests on the types of the services $res.S$ and not on the set of services where res is available. The reason for this limitation is that it simplified the encoding slightly because we do not need to encode the available-at information of services. Adding the available-at information and making a proper encoding of the two is-available-through expressions is not difficult but will most likely have a negative effect on the time required to solve the problem.

$\mathcal{E}_\omega : \mathbf{Expr}_\omega \times RES \to SMT$		
$\mathcal{E}_\omega(e_0 \texttt{ + } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) + \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ - } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) - \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ * } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) \times \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ / } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) / \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ > } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) > \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ >= } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) \geq \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ < } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) < \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ <= } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) \leq \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ = } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) = \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ != } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) \neq \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ and } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) \wedge \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ or } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) \vee \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(e_0 \texttt{ implies } e_1, res)$	=	$\mathcal{E}_\omega(e_0, res) \to \mathcal{E}_\omega(e_1, res)$
$\mathcal{E}_\omega(\texttt{not } e, res)$	=	$\neg\mathcal{E}_\omega(e, res)$
$\mathcal{E}_\omega(\texttt{- } e, res)$	=	$-\mathcal{E}_\omega(e, res)$
$\mathcal{E}_\omega(sc \texttt{ @ } s)$	=	$sc.S \cap at[s] \neq \emptyset$
$\mathcal{E}_\omega(sc \texttt{ @ } i)$	=	$sc.S \cap at[i] \neq \emptyset$
$\mathcal{E}_\omega(\texttt{->}(sc \texttt{ @ } s), res)$	=	$res.S \cap sc.S \cap at[s] \neq \emptyset$
$\mathcal{E}_\omega(\texttt{->}(sc \texttt{ @ } i), res)$	=	$res.S \cap sc.S \cap at[i] \neq \emptyset$
$\mathcal{E}_\omega(resource_parameter, res)$	=	$val[resource_parameter, res]$
$\mathcal{E}_\omega(parameter, res)$	=	$val[parameter]$
$\mathcal{E}_\omega(constant, res)$	=	$constant$

Table 5.4 – Encoding of where expressions.

5.4.6 Cyclic Designs

The procedure does not support models with cycles in the service flow graph. The reason for this is that it may invalidate the encoding of the service flow. Fortunately, cycles rarely shows up in practice. Figure 5.5 shows an example of a model and a corresponding service flow graph containing a cycle. In the model, two buses are connected to each other via two bus-to-bus bridges. If the address ranges mapped by the two bridges overlap a cycle will emerge through the bridges and buses. Attempting to invoke a service that will exercise this cycle will most likely cause a run-time error. Note that cyclic infrastructure components such as ring buses does not give rise to cycles.

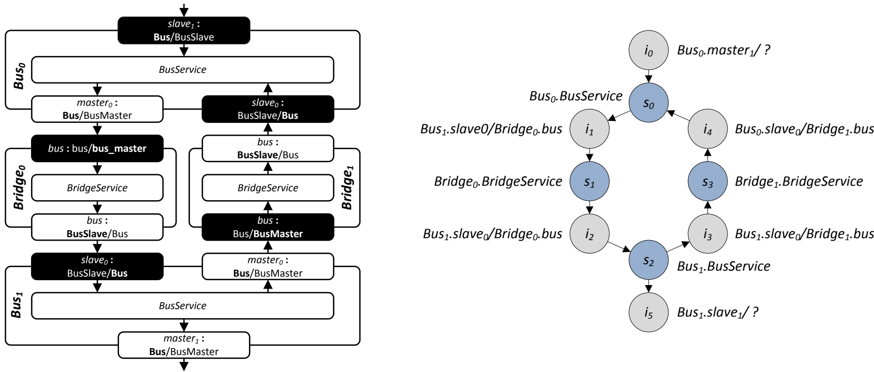


Figure 5.5 – SRM cycle example. Two buses connected to each other using two bus-to-bus bridges.

To see why cycles causes problems consider the service flow graph of figure 5.5. The encoding of the service flow can compactly be written as:

$$
\begin{array}{lll}
at[s_0] = \{s_0\} \cup at[i_0] \cup at[i_4] & at[i_0] = x & at[i_4] = at[s_3] \\
at[s_1] = \{s_1\} \cup at[i_1] & at[i_1] = at[s_0] & at[i_5] = at[s_2] \\
at[s_2] = \{s_2\} \cup at[i_2] & at[i_2] = at[s_1] & \\
at[s_3] = \{s_3\} \cup at[i_3] & at[i_3] = at[s_2] &
\end{array}
$$

where x is the set of services available at i_0. Let s_x be a service that is *not* available at i_0 ($s_x \notin x$). The following is a valid solution despite the fact that s_x is not in the service flow.

$$
\begin{array}{l}
at[s_0] = at[s_1] = at[s_2] = at[s_3] = \{s_0, s_1, s_2, s_3, s_x\} \\
at[i_1] = at[i_2] = at[i_3] = at[i_4] = at[i_5] = \{s_0, s_1, s_2, s_3, s_x\} \\
at[i_0] = x
\end{array}
$$

The underlying problem is that the solver is the solver does not construct a solution bottoms-up like the worklist-based analysis algorithm presented earlier. In general, there are many possible solutions to a set of flow equations but only the least solution corresponds to the service availablity information. By disallowing cycles the only solution to the flow equations is the least solution and, consequently, any satisfiable valuation of the variables will correspond to the correct solution.

5.5 Decoding

The process of decoding a satisfiable result produced by the solver, in the form of a valuation of the variables given in table 5.1, into an output design is rather straightforward and will only be outlined here.

For each speculative component or option x in the maximal model the value of the corresponding $inc[x]$ is checked. If the value of $inc[x]$ is true then the component or options is part of the output design and otherwise not. The encoding ensures that if an option is included then so is its parent component. For included options, the values of any parameters are immediately available in the $val_p[p]$ set of variables. Similarly, the quantities of resources and resource claims with an included speculative option can be retrieved by inspecting the $val_q[q]$ set of variables.

The topology of the output design is decoded from the values of the $sel[pc]$ variables associated with the set of possible connections. A service exchange relation of the appropriate type is added to the output design for each possible connection $pc \in PC_*$ where $sel[pc]$ is true. The encoding ensures that the resulting topology is consistent with the well-formedness rules of the service relation model (i.e. no interface is connected to more than one service exchange relation and so on).

5.6 Experiments

To show that generating designs on the basis of consistency information was indeed possible a couple of small experiments were carried out. Because the procedure for generating a consistent design is based on MAXSAT we expect that it will only be able to handle smaller problems in reasonable time. For this reason, the complexity of the experiments presented in this section has been kept low.

The experiment is an attempt at generalizing the procedure for automatically generating an abstraction layer implementing the communication infrastructure of a process network presented in the previous chapter. Besides choosing implementations for each channel in the process network we also want to determine the values of some configurable parts of the platform.

Figure 5.6 shows an overview of the tool, based on the `xSRM` framework, used for the experiments. The tool takes as input a pair of MHS and MSS files speci-

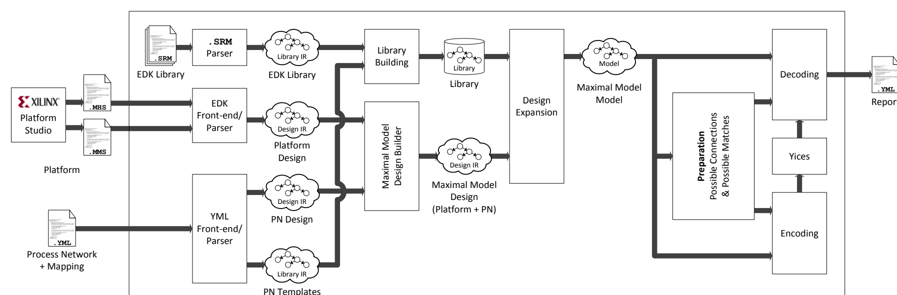


Figure 5.6 – Overview of the tool used for experimental purposes.

fying a platform and a YML file containing a specification of a process network application and a mapping of its processes onto the processors of the platform. The platform and the process network are combined into a design representing the maximal model and then this design is expanded into a proper model. The resulting model is used as the basis for computing the possible connections and the possible matches and for the encoding and decoding of the MAXSMT problem. The output produced by the tool is a report containing information about the generated design and other information useful for debugging purposes.

5.6.1 Platform

The same platform will be used for all of the experiments presented here. The platform was created using the Xilinx EDK tool and imported into the `xSRM` framework using the EDK front-end previously described in section 3.4. Figure 5.7 shows a schematic of the platform used in the experiments. The hardware platform consists of a processor local bus (PLB) bus, two Microblaze processors, two (local) BRAM memories, two timer/counters and a number of interfacing components for connecting the processors with the memories. The microblaze debug module, connected to the PLB bus as a slave, was inserted automatically by the EDK when the platform was generated but has no real use for our purposes. Each of the processors are configured with an instance of the Xilkernel operating system. Notice that the platform does not support any means for inter-processor communication.

Ideally, the platform should also include an interrupt sub-system for connecting the timer/counters with the processors. This, however, has been left out to keep the complexity of the model down.

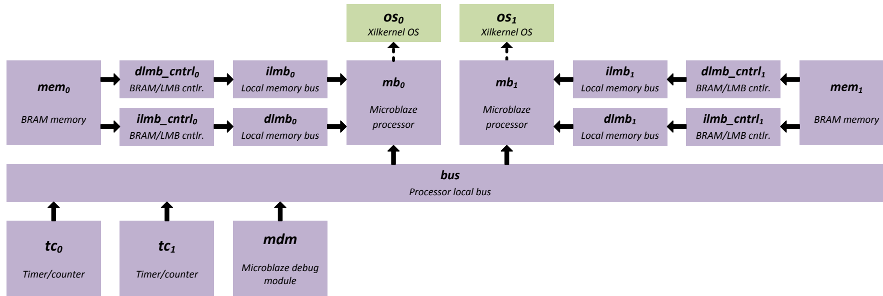


Figure 5.7 – Schematic of the platform used for the experiments.

In comparison with the platforms used in the previous chapter on automated programming, this platform provides some degree of customizatibility. More specifically, the platform allows for additional hardware components to be connected to the processor local bus and for the Xilkernel operating systems to include support for message queues. Both kinds of customizability are modeled as options that can be included in the processor local bus and Xilkernel components.

Listing 5.1 shows parts of the template used for the Xilkernel. As can be seen, the template provides an option for adding message queue support in the kernel. The message queue can be used for inter-process communication between processes mapped to the same processor. The option can be included multiple times allowing the same kernel to provide multiple message queues with different depths and sizes. The maximum number of message queues allowed in an instance of the kernel is defined using the instantiation restrictions later on.

Because of the limitations of the interface group concept discussed previously in section 5.1.1 it has been necessary to change the way the relationship between a software entity and a processor is modelled. Instead of relating a processor with each of its associated software entities through an instance of the `Processor/SWE` service exchange relation, the processor is instead related to the context component and the services offered by the processor imported into the context service. A software entity may then access the services provided by the processor through a Callee/Context interface. Although we would prefer to use the other approach to modeling the relations between processors and software entities there is nothing ”wrong” with doing it this way.

```
library {
  namespace Xilinx.EDK {
    template Xilkernel {
      interface ContextAsCaller : sub(ContextToCaller);
      interface ContextAsCallee : obj(CalleeToContext);
      ...

      option MessageQueue {
        parameter Depth { min = 1; max = 10000; }
        parameter Size  { min = 1; max = 10000; }

        service Enqueue : SCEnqueue { export(ContextAsCallee); }
        service Dequeue : SCDequeue { export(ContextAsCallee); }

        resource QueueResource : RCQueue<D,S> {
          quantity = 1;
          set D = Depth;  set S = Size;
          export(Enqueue, Dequeue);
        }
      }
    }
  }
}
```

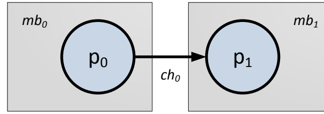
Listing 5.1 – Xilkernel Template

5.6.2 Application

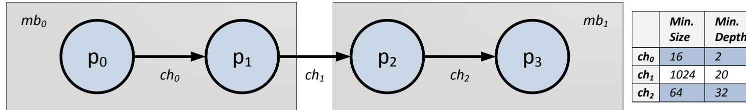
The experiments will use three different applications of varying complexity. Figure 5.8 shows the process networks for the applications. The mappings of the processes to the two processors of the platform are also shown. The process networks for application 1 and 2 are very simply and does not represent any real applications. The third application, the MJPEG encoder that was the subject of the case study in the previous chapter, is more complicated and does represent a real application.

The process networks comprising the three applications are captured in the YML format. The YML description of an application is transformed into a partial Service Relation Model description by means of the YML front-end to the `xSRM` framework. The generated description of an application is quite simple and consists of only a single component. The component has one interface of type `CallerToContext` per process. When combined with a platform, the interfaces of the component are connected to the context components so that the interface associated with a given process is connected to the context of the processor to which the process is mapped. Figure 5.9 shows the component generated for Application 2. In addition to the interfaces, the component also contains one resource claim per channel.

Application 1



Application 2



	Min. Size	Min. Depth
ch_0	16	2
ch_1	1024	20
ch_2	64	32

MJPEG Application

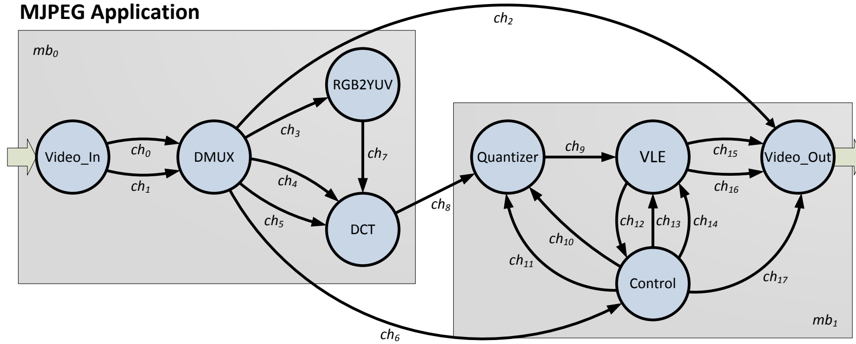


Figure 5.8 – The three example applications

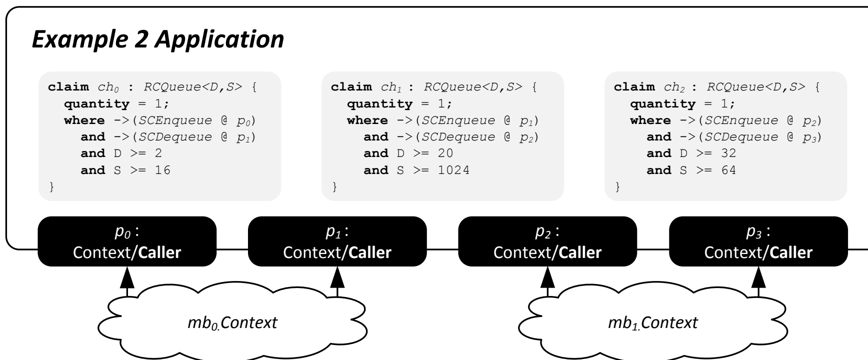


Figure 5.9 – Component representing Application 2.

```
library {
  namespace Xilinx.EDK {
    template BramMemory {
      parameter Size { min = 1; max = 0xFFFFFFFF; }

      interface PortA : obj(BramToBramController);
      interface PortB : obj(BramToBramController);

      service MemoryService : SCMemoryService {
        export(PortA, PortB);
      }

      resource QueueResource : RCMemory {
        quantity = Size;
        export(MemoryService);
      }
    }
  }
}
```

Listing 5.2 – BRAM Memory Template

5.6.3 Library Components

The library of components that the procedure can used to make the design consistent consists of a software queue, a bus-to-memory (processor local bus/BRAM memory) controller and a BRAM memory. The bus-to-memory controller is used to connect a BRAM memory to a PLB bus. The component representing the bus-to-memory controller has a trivial description in the Service Relation Model and will not be discussed in detail here.

The BRAM memory is a dual port memory whose size is configurable. The Service Relation Model description of the memory is shown in listing 5.2. Each of the two ports of the memory are modelled using an interface. A service belonging to the `SCMemoryService` service class is exported through both interfaces. Besides the two interfaces and the service, the component also provides a resource of type `RCMemory`. The quantity of resource (bytes of memory) provided is configurable by means of the parameter `Size`.

The software queue resembles the queues produced by the $\mathtt{IS}_0$ implementation scheme presented in the previous chapter. The queue is implemented as a circular buffer placed in a memory and synchronization between the writer and reader is done using polling. The Service Relation Model description of the software queue, shown in listing 5.3, has two pairs of interfaces used for relating it with the context(s) where from where data may be enqueued and dequeued. The two interfaces of each pair of interfaces are grouped into an interface group of their own meaning that they must be connected to the same component (context in this case). If both the reader and the writer of the queue

```
library {
  namespace Components {
    template SoftwareQueue {
      parameter Depth { min = 1; max = 100000; }
      parameter Size  { min = 1; max = 100000; }

      interface EnqueueContextAsCallee : obj(CalleeToContext) { group = 0; };
      interface EnqueueContextAsCaller : sub(ContextToCaller) { group = 1; };

      interface DequeueContextAsCallee : obj(CalleeToContext) { group = 0; };
      interface DequeueContextAsCaller : sub(ContextToCaller) { group = 1; };

      service Enqueue : SCEnqueue { export(EnqueueContextAsCallee); }
      service Dequeue : SCDequeue { export(DequeueContextAsCallee); }

      resource QueueResource : RCQueue<D,S> {
        quantity = 1;
        set D = Depth;
        set S = Size;
        export(Enqueue, Dequeue);
      }

      claim MemoryRequirement : RCMemory<D,S> {
        quantity = Depth * 100 + 8;
        where ->(CSMemoryService @ EnqueueContextAsCaller)
          and ->(SCMemoryService @ DequeueContextAsCaller);
      }
    }
  }
}
```

Listing 5.3 – Software Queue Template

are hosted on the same processor the two pairs of interfaces may be connected to the same context. The component exports two services called `Enqueue` and `Dequeue`. Both services are specializations of a pair of general versions. A resource of type `RCQueue<D,S>` is exported through these two services. The size (S) and depth (D) of the resource is configurable by means of two parameters (Depth and Size). Finally, the component has a resource claim for claiming memory required for implementing the circular buffer. The quantity associated with the claim is set by means of a quantity expression to `Depth * 100 + 8`. Ideally, this should have been `Depth * Size + 8` but because multiplication in Yices is restricted to the case where at least one operand is a constant the size has been statically set at 100.

5.6.4 Setup

The platform and the three applications are combined to form three different designs that are used as input to the procedure. The purpose of the experiments is partly to demonstrate that automated design generation is in fact possible

Type	Name	Cost	Max. Instances		
Experiment			**Ex. 1**	**Ex. 2**	**MJPEG**
Component	Components.SoftwareQueue	10	1	2	3
Component	Xilinx.EDK.XpsBramIfCntlr	100	1	1	1
Component	Xilinx.EDK.BramBlock	20	3	3	3
Option	Xilinx.EDK.Plb_v46.PlbSlave	4	4	4	4
Option	Xilinx.EDK.Xilkernel.MessageQueue	5	3	3	10
Option	Components.Context.Callee	1	10	10	15
Option	Components.Context.Caller	1	10	10	15

Table 5.5 – Instantiation restrictions for the three experiments. Recall that the maximum number of instance specified in the instantiation restrictions *includes* the instances in the input design.

and partly to determine how complex problems can be handled. An experiment consists of an input design and a set of instantiation restrictions. Table 5.5 shows the instantiation restrictions used for the three experiments.

Note the cost associated with the different instantiation restrictions does not correspond to any "real" cost and have been chosen only for the purpose of prioritizing. For all of the three experiments, the maximum number of instances defined by the instantiation restrictions is close to the minimum of what is needed to ensure that a consistent design can be generated. Again, this is done to keep the complexity down.

5.6.5 Results

Before reporting on the execution time of the procedure, we will give a more detailed presentation of the result produced using the design based on application 2 and the instantiation restrictions given below:

Figure 5.10 is an attempt at visualizing the maximal model created on the basis of the input design and the instantiation restrictions. There are three inconsistencies in the model in the form of the three resource claims associated with the channels of the application. The three resource claims cannot be satisfied by the platform because no queue resources (`RCQueue`) are provided. Notice the rather large number of possible connections between the interfaces of the two software queue components and the context components.

Figure 5.11 shows the result produced by the procedure. Three components have been added to the design: a software queue, a PLB/BRAM interface con-

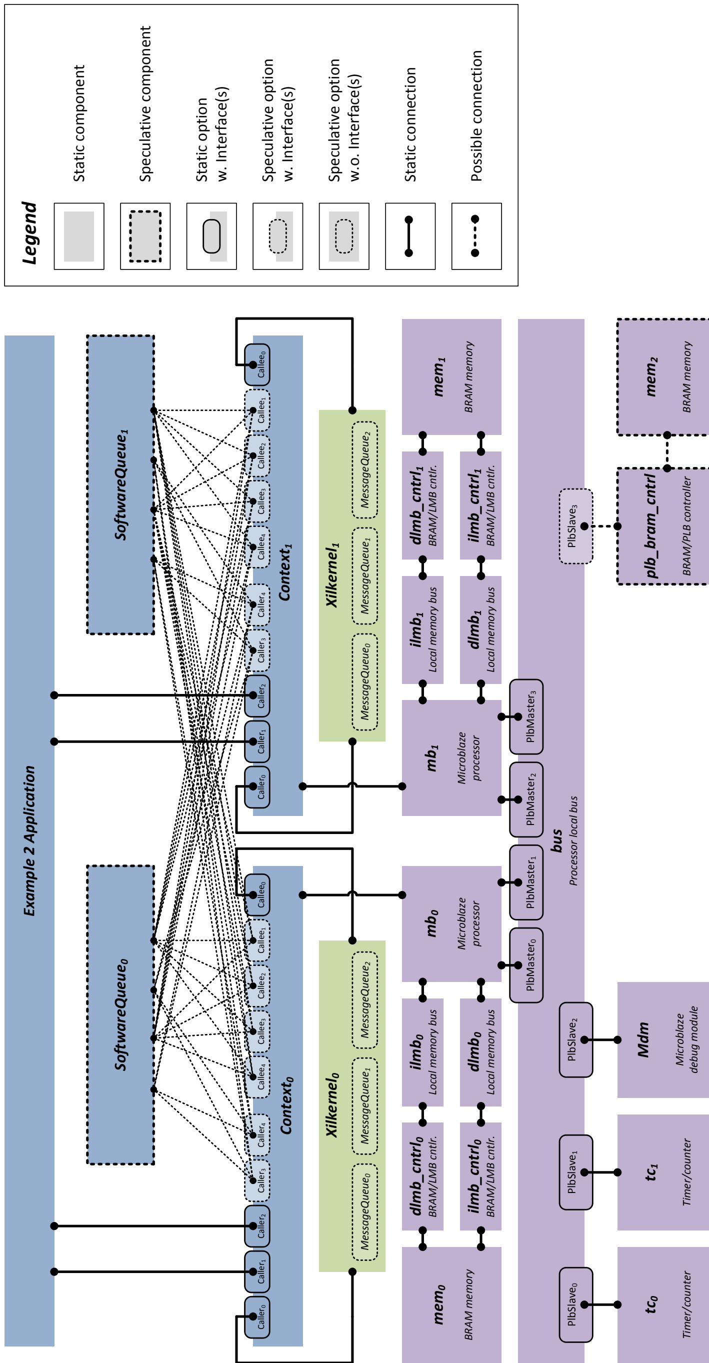


Figure 5.10 – Maximal model of experiment 2.

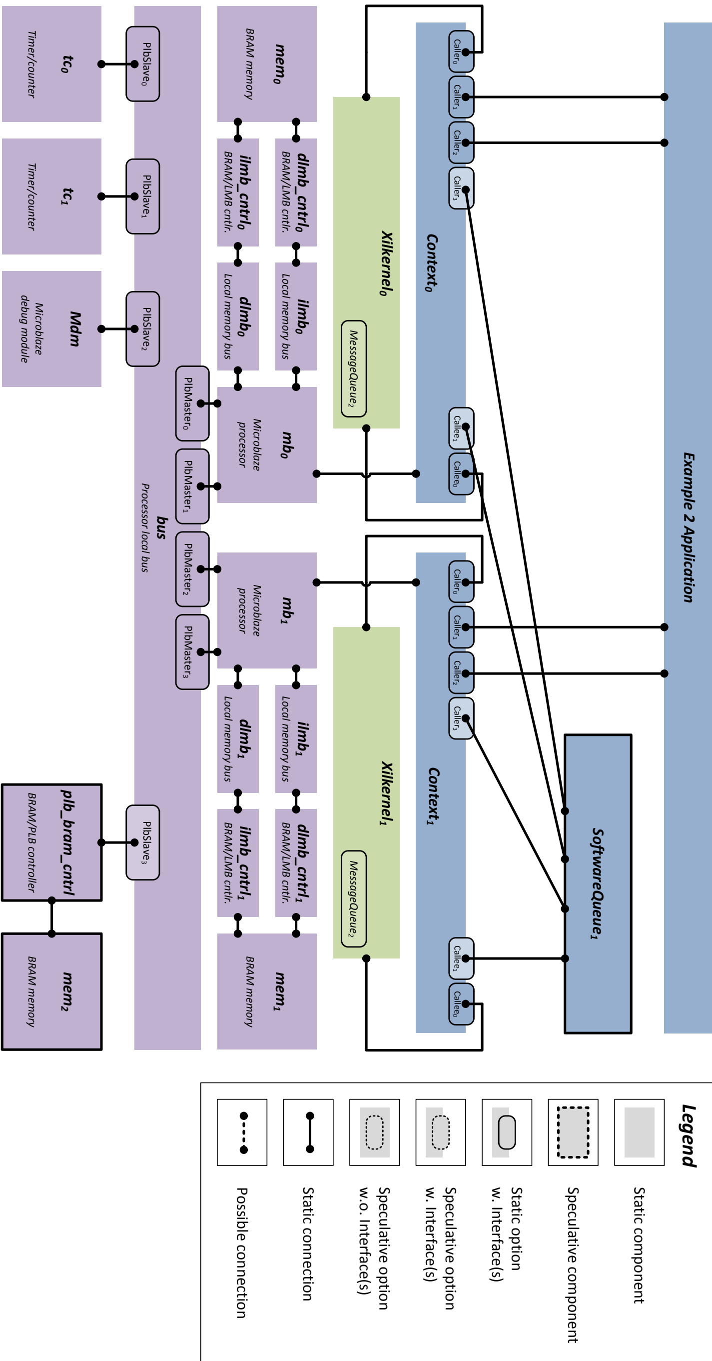


Figure 5.11 – Result produced for experiment 2.

Parameter	**Value**
SoftwareQueue$_1$.*Depth*	20
SoftwareQueue$_1$.*Size*	1024
mem$_2$.*Size*	4016
Xilkernel$_0$.MessageQueue$_2$.*Depth*	2
Xilkernel$_0$.MessageQueue$_2$.*Size*	16
Xilkernel$_1$.MessageQueue$_2$.*Depth*	32
Xilkernel$_1$.MessageQueue$_2$.*Size*	64

Table 5.6 – Parameter values computed by the procedure for the components and options added to experiment 2.

troller and a BRAM memory. Besides the components, a pair of Callee and Caller options have been added to both of the contexts for connecting the added software queue and a number of message queue options have been added to the Xilkernel components. The inconsistency of the input design caused by the resource claim associated with channel ch_0 has been resolved by including a message queue option in the component Xilkernel$_0$. Similarly, the inconsistency caused by the resource claim associated with channel ch_2 has been resolved by the inclusion of a message queue option in Xilkernel$_1$. The last inconsistency, caused by the resource claim associated with channel ch_1, has been resolved by adding a software queue to the design. Adding this queue, however, introduces a new inconsistency because the platform does not include a memory reachable from both processors. This inconsistency is handled by connecting a BRAM memory block through a PLB/BRAM interface controller to the bus.

Besides adding new components and options to the design, the procedure has also determined the values of their parameters. Table 5.6 shows the values computed for the add components and options. Six of the seven parameters are used for specifying the dimensions of the three queues added. Their values match the minimum values specified for the three channels in the YML file of experiment 2 (see Figure 5.8). The value of the size parameter for the memory has been set at 4016 even though it only needs to be 2008 ($20\times100+8$) in order to satisfy the resource claim of the software queue. This illustrates a general shortcoming of the procedure: it is not possible to associate a "cost" with the values of parameters and, as a consequence, results like this may show up. This is unfortunate as parameters are likely to be used to describe dimensions of components whose chosen value is proportional to a real cost as is the case with the memory here.

Although simple, the experiment demonstrates that the procedure works and shows that it can handle inconsistencies that are not completely trivial. The

Experiment	Variables	Expressions	Time (hh:mm:ss)
Experiment 1	597	2536	00:00:08
Experiment 2	690	3602	00:03:18
MJPEG	1457	18059	n/a

Table 5.7 – Complexity in terms of variables and expressions in the problem and execution time of Yices for the three experiments. No result was obtained for the MJPEG example. The experiments were run on an Intel Core2 Quad CPU at 2.40 GHz running Microsoft Windows 7.

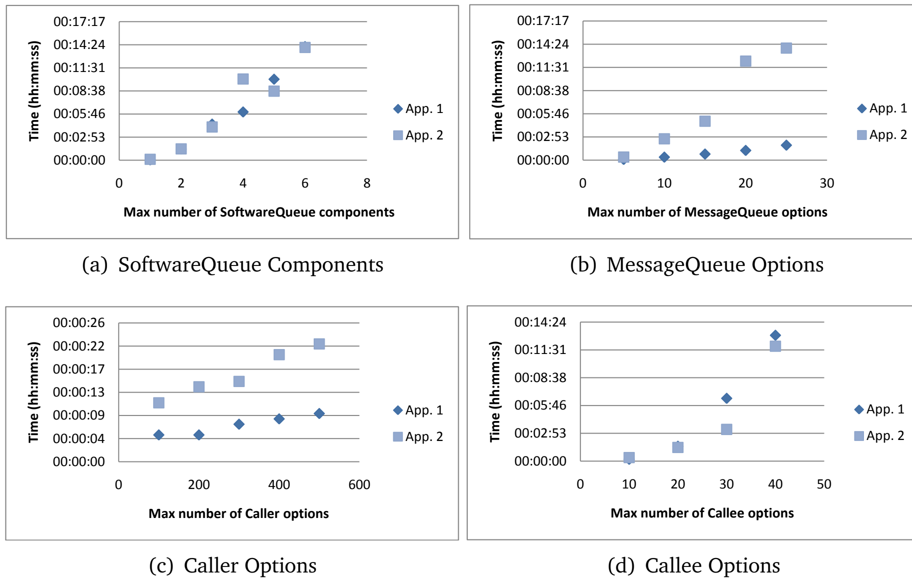
inconsistency caused by channel ch_1 cannot be resolved by just adding one component or including an option but requires a rather non-obvious combination of several components

5.6.5.1 Timing Results

Table 5.7 shows the time required by Yices to solve the MAXSMT problem for the three experiments. As can be seen, experiment 1 ran for about 8 seconds before Yices came up with a solution. The marginally more complex experiment 2 required 3 minutes and 18 seconds before a solution was found. No result has been obtained for the MJPEG experiment. The MJPEG experiment was aborted after having been running for 24 hours without producing a result. Whether it would take days, weeks, months or perhaps years to come up with a result remains unknown.

For reference, the size of the SMT problems for the three experiments are also shown in Table 5.7. The size is given as the number of variables and expressions in the problem. The number of expressions are not completely accurate as some expressions are expanded into multiple expressions as part of the encoding. In general, there is not a strong correlation between the number of variables and expressions in a problem and the time required to solve it. This observation is supported by the data enclosed in appendix E.

The results show that the procedure is capable of handling small problems quite fast but cannot handle even medium sized problems such as the MJPEG example. The logical explanation for this is that the complexity of the MAXSMT problem, and thus time time required to find a solution, increases exponentially – and at a rather extreme rate.



(a) SoftwareQueue Components

(b) MessageQueue Options

(c) Caller Options

(d) Callee Options

Figure 5.12 – Varying the number of speculative components and options. The data supporting the plots of Figure 5.12 can be found in appendix E.

To investigate what causes this sudden explosion in complexity a number of other experiments have been carried out. The experiments are all based on experiments 1 and 2 and consists of varying the maximal number of instances set in the instantiation restrictions. Figure 5.12 shows the time required by Yices for finding a (satisfiable) solution for both Experiments 1 and 2 when the maximum number of instances of a single instantiation restriction is increased. The data supporting the plots of Figure 5.12 can be found in appendix E.

As can be seen, increasing the number of `SoftwareQueue` components causes the execution time of Yices to increase rather dramatically. Both experiment 1 and 2 requires a single software queue and adding extra instances does not give rise to a different solution. Increasing the maximum number of `MessageQueue` options for experiment 2 is also associated with a significant increase in execution time although not as dramatic as adding software queues. For Experiment 1, increasing the maximum number of `MessageQueue` options is associated with an, in this context, insignificant increase in execution time. The reason for this is that they are not used in Experiment 1 and, consequently, does not have an impact on the satisfiability of the problem. Figures 5.12(c) and 5.12(d) shows the effect on the execution time for increasing the maximum number of Caller and Callee options. The Caller and Callee options influences the number of different

ways the software queue components can be connected to the two context components and thus the number of possible connections in the problem. As can be seen, increasing the maximum number of Callee options is more demanding than increasing the maximum number of Caller options. Why this is the case we do not know but it may have to do with the fact that the interface contains within a Callee interface is a subject interface and the services available at a such is used for defining the services available at a larger part of the model than what is the case for the object interface contains within a Caller option.

In these experiments, the maximum number of instances for all but one of the instantiation restrictions were kept constant. When the maximum number of instances for several of the instantiation restrictions are increased then the execution time rises even more dramatically. This is, of cause, a natural consequence of the complexity of the MAXSMT problem which is known to be NP.

The results clearly show that the procedure is not practically useful because of the overwhelming computational requirements associated with evaluating all but the smallest problems. Although we did initially expect to have problems with the scalability, we did not expect for it to be this extreme. However, considering the size of the solution space and and using the timing result obtained in chapter 3 for checking the consistency of a model as a benchmark for the time required to test one possible solution it is obvious that the problem is indeed very complex and that the results were to be expected. A more feasible approach to tackle the optimization problem could be to use meta-heuristic methods supported by the consistency checking procedure presented earlier in chapter 3.

5.7 Optimizations

The performance of the procedure is rather poor and not well suited for practical applications. In this section, we will briefly discuss two possible optimizations that may improve the performance of the procedure.

Flow Merging. A significant part of the complexity is due to the encoding of the service flow. The bit-vector operations used to encode service availablity propagation are quite expensive. By collapsing the encoding of the service flow in components and options some computation can be saved. Also, the components and the options of the input design could be merged.

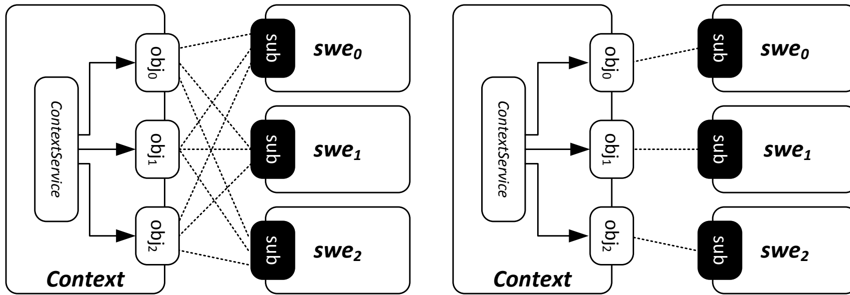


Figure 5.13 – Reducing the set of possible connections from 9 to 3

Connection Reduction. A major source of complexity is the encoding of the possible connections between the components. In the presented encoding, each (free) interface can be connected to all other (free) interfaces of a compatible type. This means that for a design with x object and y subject interfaces of a given service exchange relation there are $x \times y$ possible connections. It is, however, often the case that a significant number of these possible connections are equivalent meaning that regardless of which, if any, of them is chosen as an actual connection the service flow of the resulting model is effectively identical.

As an example, consider the service exchange relation `Context/Caller` used to relate software entities relations with an execution context. The context component provides a number of completely identical interfaces each of which are used to relate it to a single software entity. It does not matter which of these interfaces a given software entity is connected to because the interfaces are identical both regarding the type (relation and role) and the set of services available at them. For a design with three software entities and a single context there are a total of 9 different possible connections (see Figure 5.13). This can be reduced to 3 if we are able to determine that the three interfaces on the processor are in fact equivalent (Figure 5.13).

While this optimization would most likely do wonders for the three simple problems considered, it is not a silver bullet because it the number of possible connections would still grow rapidly as more components are added.

5.8 Discussion & Summary

In this chapter, we have presented an approach to generate consistent designs on the basis of an inconsistent design and a library of components. The approach attempts to transform the inconsistent design into a consistent one by adding components from the library.

Besides the problem with the overwhelming computational requirements, the presented procedure has a number of shortcoming and limitations that must be addressed before the procedure can be put to practical use.

A serious shortcoming of the procedure is that it can only deal with costs associated with the inclusion of components and parameters. Choosing the parameter values associated with a speculative component or option is not associated with a cost and may be chosen freely by the solver. For example, the cost of including the message queue option in the Xilkernel component is the same regardless of the size and depth of the queue.

Another shortcoming is the lack of a ”compositional rules” concept that can be used to restrict how components are connected. The concept of interface groups is not sufficient to properly handle the kinds of restrictions that one would like to make in practice. A more capable solution would probably involve a ”topology constraint” language of sorts.

In theory, the procedure can also be used to generate a design from scratch given only a Service Relation Model description of the top (application) layer. To be practical in such a scenario, however, the procedure will have to be extended with the ability to reason about non-functional properties and constraints. As it is, the procedure will return the smallest possible consistent design. How to add support for non-functional properties and constraints is an open question. Experimentation will be difficult without a faster way to solve the optimization problem and we believe that this problem should be addressed either before or in tandem with the question of how to approach the inclusion of non-functional properties.

Many of these shortcoming could probably be addressed one way or the other but once the problem with the computational requirements were discovered work on addressing the other problems was put to a halt. The reader may ask why MAXSMT was used when the results of chapter 4 clearly showed that it was computationally expensive and only feasible for relatively small problems. The reason for this is that most of the work presented in this chapter was carried out prior to the work presented in chapter 4.

CHAPTER 6

Conclusion

6.1 Contributions

Design automation tools supporting the platform-based design approach must be able to reason about the capabilities of platforms. By imposing restrictions on the supported platforms it is often possible to replace the need for analysis by a set of assumptions. Such restrictions can help simplifying the tool makers job at the cost of limiting the design space covered by the tool.

This dissertation has presented an attempt at devising a formalism and an associated analysis method for automated analysis of platform capabilities. The result has been named the Service Relation Model emphasizing its focus on relations between abstract services. In the Service Relation Model, a platform is viewed as a network of components providing and forwarding services.

Three different applications of the Service Relation Model have been presented in this dissertation:

In the first application, the Service Relation Model was used to check the consistency of designs with respect to service and resource availablity. For each component of a design we are able to determine whether or not it has access to the necessary services and resources that it will need to function properly. The

consistency check is implemented by adding additional concepts to the core Service Relation Model. Compared to other approaches, the consistency check does not require that the dependencies amongst the components have been explicitly asserted in the input nor does it assume the existence of a component framework. Our experiments shows that the procedure is capable of handling rather large problems in a reasonable time frame.

In the second application, information retrieved from analyzing a Service Relation Model description of a platform was used to generate an abstraction layer implementing the communication infrastructure of a process network application. The presented procedure uses analysis information for determining the set of possible channel implementations based on a set of abstract channel implementation schemes. An optimization problem, in the form of a MAXSMT problem, is formulated and solved to determine, for each channel, which of its possible implementations should be used in the final implementation. The back-end of the procedure uses service invocation synthesis – an approach to code generation based on the Service Relation Model. Compared to other similar procedures and approaches, the presented procedure is completely independent of the platform and can easily be extended with new platform components and implementation schemes. Experiments show that MAXSMT is capable of handling small to medium sized designs reasonably fast but does not scale well and, as a consequence, is not practical for larger designs.

In the third and final application, the concepts of the Service Relation Model and the concept of consistency is used as the foundation for a procedure that, given an inconsistent design, attempts to generate a new consistent design containing the given inconsistent design. The procedure generalizes parts of the previously presented procedure for implementing the communication infrastructure of process networks. At the heart of the procedure is a non-linear optimization problem. An attempt at formulating and solving the problem as a MAXSMT problem has been undertaken. Experiments show that only very small problems can be handled in reasonable time.

Collectively, the three applications demonstrates the worth of the Service Relation Model and the associated service availablity analysis. Using abstract re-usable descriptions of the components of platforms, we have shown how information about its capabilities can be extracted and used for diverse purposes such as consistency checking and automated programming.

6.2 Discussion

The real contribution of this work is the concept of service aggregation upon which the service relation model is based. In our view, it is so simple and obvious that others before us surely must have used it for similar purposes. Despite our best efforts, however, we have been unable to find any prior work based on the same or a similar concept.

Two of the presented applications of the service relation model requires an optimization problem to be solved. In this dissertation, these problems have been solved using the MAXSMT feature of the Yices solver. Unfortunately, the exponential running time of MAXSMT means that both of the presented procedures can only handle problems of a limited size in reasonable time. In general, we conclude that the use of MAXSMT for solving the kind of optimization problems encountered in this work is not practical due to the overwhelming computational complexity. Alternative ways of dealing with the optimization problems must be investigated. We believe that an approach based on the use of meta-heuristics may be appropriate in both cases.

Although the performance results of the two procedures proved to be unsatisfactory, we still believe that the two procedures served their intended purpose of demonstrating the usefulness and versatility of the Service Relation Model.

6.3 Future Work

The Service Relation Model, as presented in this thesis, focuses exclusively on functional aspects. In practice, the non-functional properties of systems cannot be ignored. Extending the Service Relation Model so that it may be used to analyze different non-functional properties is an important next step. We theorize that many of the ideas employed by the Metropolis framework for analyzing non-functional properties can also be used with the Service Relation Model.

Another shortcoming of the Service Relation Model, that could be addressed, is that it can only be used for modeling static platforms and systems. Whether or not this is worth pursuing, however, depends on the intended application of the model. Within the context of embedded systems, there are plenty of areas where the lack of means to model dynamic behavior is not an issue.

The expressibility of the presented concepts of assertions and resource claims used for capturing component dependencies is somewhat limited. It could be

interesting to explore the possibility of replacing both concepts with a more general ”constraint language” that supports more advanced reasoning with respect to the structure and meaning of the model and its constituent parts.

In this thesis, we have presented three different uses of the Service Relation Model. We believe that the model has more interesting uses that could be explored. In this work, we have focused solely on applications given using the process networks MoC. It would be interesting to see if the methods and principles used for process networks could be used for other MoC’s as well.

APPENDIX A

Alternative Analysis Algorithm

This appendix contains the pseudo code for the alternative analysis algorithm mentioned in section 2.4. The algorithm differs from algorithm 1 from section 2.4 in that it computes the availablity of interfaces in addition to the availablity of services.

Algorithm 2 Available Services - Worklist Algorithm

```
 1: procedure SOLVE(S, I, E)
 2:     worklist := E                                          ▷ Initialization
 3:     for each x ∈ S ∪ I do
 4:         available[x] := { x }
 5:     end for
 6:     while worklist ≠ ∅ do                                  ▷ Main loop
 7:         e := DEQUEUE(worklist)
 8:         t := GETTARGETNODE(e)
 9:         s := GETSOURCENODE(e)
10:         if ¬ (available[t] ⊆ available[s]) then
11:             available[t] := available[t] ∪ available[s]
12:             for each o ∈ GETSUCCESSOREDGES(t) do
13:                 ENQUEUE(o)
14:             end for
15:         end if
16:     end while
17: end procedure
```

Appendix B

Evaluation Functions

$\Omega_e : \mathbf{Expr}_\omega \times RES \times (S \times I \rightarrow \mathcal{P}(S)) \rightarrow \{\texttt{true}, \texttt{false}\})$		
$\Omega_e(e_0 \texttt{ + } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) + \Omega_e(e_1, res, sa)$
$\Omega_e(e_0 \texttt{ - } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) - \Omega_e(e_1, res, sa)$
$\Omega_e(e_0 \texttt{ * } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) \times \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ / } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) / \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ > } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) > \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ >= } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) \geq \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ < } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) < \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ <= } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) \leq \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ = } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) = \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ != } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, sa) \neq \Omega_e(e_1, res, p, sa)$
$\Omega_e(e_0 \texttt{ and } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, p, sa) \wedge \Omega_e(e_1, res, sa)$
$\Omega_e(e_0 \texttt{ or } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, p, sa) \vee \Omega_e(e_1, res, sa)$
$\Omega_e(e_0 \texttt{ implies } e_1, res, sa)$	$=$	$\Omega_e(e_0, res, p, sa) \rightarrow \Omega_e(e_1, res, sa)$
$\Omega_e(\texttt{not } e, res, sa)$	$=$	$\neg\Omega_e(e, res, sa)$
$\Omega_e(\texttt{- } e, res, sa)$	$=$	$-\Omega_e(e, res, sa)$
$\Omega_e(sc \texttt{ @ } s)$	$=$	$sc.S \cap sa[s] \neq \emptyset$
$\Omega_e(sc \texttt{ @ } i)$	$=$	$sc.S \cap sa[i] \neq \emptyset$
$\Omega_e(\texttt{->(}sc \texttt{ @ } s\texttt{)}, res, sa)$	$=$	$\overline{RA}[res] \cap sc.S \cap sa[s] \neq \emptyset$
$\Omega_e(\texttt{->(}sc \texttt{ @ } i\texttt{)}, res, sa)$	$=$	$\overline{RA}[res] \cap sc.S \cap sa[i] \neq \emptyset$
$\Omega_e(resource_parameter, res, sa)$	$=$	$res.RP_v[resource_parameter]$
$\Omega_e(constant, res, sa)$	$=$	$constant$

Table B.1 – Evaluation of where expressions. It is assumed that the input expression is well-formed with respect to types.

$Q_e : \mathbf{Expr}_{\mathrm{qty}} \times (SYM \to \mathbb{Z} \cup \bot) \to \{\mathtt{true}, \mathtt{false}\})$		
$Q_e(e_0 \texttt{ + } e_1, p)$	$=$	$Q_e(e_0, p) + Q_e(e_1, p)$
$Q_e(e_0 \texttt{ - } e_1, p)$	$=$	$Q_e(e_0, p) - Q_e(e_1, p)$
$Q_e(e_0 \texttt{ * } e_1, p)$	$=$	$Q_e(e_0, p) \times Q_e(e_1, p)$
$Q_e(e_0 \texttt{ / } e_1, p)$	$=$	$Q_e(e_0, p) / Q_e(e_1, p)$
$Q_e(e_0 \texttt{ > } e_1, p)$	$=$	$Q_e(e_0, p) > Q_e(e_1, p)$
$Q_e(e_0 \texttt{ >= } e_1, p)$	$=$	$Q_e(e_0, p) \geq Q_e(e_1, p)$
$Q_e(e_0 \texttt{ < } e_1, p)$	$=$	$Q_e(e_0, p) < Q_e(e_1, p)$
$Q_e(e_0 \texttt{ <= } e_1, p)$	$=$	$Q_e(e_0, p) \leq Q_e(e_1, p)$
$Q_e(e_0 \texttt{ = } e_1, p)$	$=$	$Q_e(e_0, p) = Q_e(e_1, p)$
$Q_e(e_0 \texttt{ != } e_1, p)$	$=$	$Q_e(e_0, p) \neq Q_e(e_1, p)$
$Q_e(e_0 \textbf{ and } e_1, p)$	$=$	$Q_e(e_0, p) \wedge Q_e(e_1, p)$
$Q_e(e_0 \textbf{ or } e_1, p)$	$=$	$Q_e(e_0, p) \vee Q_e(e_1, p)$
$Q_e(e_0 \textbf{ implies } e_1, p)$	$=$	$Q_e(e_0, p) \to Q_e(e_1, p)$
$Q_e(\textbf{not } e, p)$	$=$	$\neg Q_e(e, p)$
$Q_e(\texttt{- } e, p)$	$=$	$-Q_e(e, p)$
$Q_e(parameter, p)$	$=$	$p[parameter]$
$Q_e(constant, p)$	$=$	$constant$

Table B.2 – Evaluation of quantity expressions. It is assumed that the input expression is well-formed with respect to types.

APPENDIX C

Implementation Scheme C Templates

This appendix contains the C templates used for generating implementations for the access modules of the implementation schemes that were not presented with the main text in section 4.4.1.

```
#include <shared.c>
void chx_write_init
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemInit s1>() {
  initialize_memory<r0,s0>();
  [s1(⌊r1⌋, CH_DEPTH)]; /* initialize semaphore to CH_DEPTH */
}

void chx_write
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemWait s2>(void* data) {
  [s2(⌊r1⌋)]; /* wait */
  copy_to_token<r0,s0>(data);
  inc_wr_idx<r0,s0>();
}

void chx_write_handler(RCSemaphore r1, SCSemPost s3) {
  [s3(⌊r1⌋)]; /* post */
}
```

Listing C.1 – Template for the write access module of $\mathtt{IS}_2$.

```
#include <shared.c>
void chx_read_init() { /* empty */ }

void chx_read<RCMemory rs, SCMemoryService s, SCIHandler i>(void* data) {
  while(is_empty<rs,s>());
  copy_from_token<rs,s>(data);
  inc_rd_idx<rs,s>();
  [i();] /* notify */
}

void chx_read_handler() { /* empty */ }
```

Listing C.2 – Template for the read access module of $\mathtt{IS}_2$.

```
void chx_write_init<RCMemory r0, SCMemoryService s0>() {
  initialize_memory<r0,s0>();
}


void chx_write
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemWait s2,
   SCIHandler i1>(void* data) {
    while(is_full<r0,s9>());
    copy_to_token<r0,s0>(data);
    inc_wr_idx<r0,s0>();
    [i0();] /* notify */
  }

void chx_write_handler() {
  /* empty */
}
```

Listing C.3 – Template for the write access module of $\mathtt{IS}_3$.

```
void chx_read_init<RCSemaphore r1, SCSemInit s1>() {
  [s1(⌊r1⌋, 0)]; /* initialize semaphore to 0 */
}

void chx_read
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemWait s2,
   SCIHandler i1>(void* data) {
    [s2(⌊r1⌋)]; /* wait */
    copy_from_token<r0,s0>(data);
    inc_rd_idx<r0,s0>();
  }

void chx_read_handler<RCSemaphore r1, SCSemPost s3>() {
  [s3(⌊r1⌋)]; /* post */
}
```

Listing C.4 – Template for the read access module of $\texttt{IS}_3$.

```
void chx_write_init
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemInit s1>() {
    initialize_memory<r0,s0>();
    [s1(⌊r1⌋, CH_DEPTH)]; /* initialize semaphore to CH_DEPTH */
  }


void chx_write
  <RCMemory r0, RCSemaphore r1, SCMemoryService s0, SCSemWait s2,
   SCIHandler i1>(void* data) {
    [s2(⌊r1⌋)]; /* wait */
    copy_to_token<r0,s0>(data);
    inc_wr_idx<r0,s0>();
    [i0();] /* notify */
  }

void chx_write_handler<RCSemaphore r1, SCSemPost s3>() {
  [s3(⌊r1⌋)]; /* post */
}
```

Listing C.5 – Template for the write access module of $\texttt{IS}_4$.

```
void chx_read_init<RCSemaphore r2, SCSemInit s4>() {
  [s4(⌊r1⌋, 0)]; /* initialize semaphore to 0 */
}

void chx_read
  <RCMemory r0, RCSemaphore r2, SCMemoryService s0, SCSemWait s5,
   SCIHandler i1>(void* data) {
    [s5(⌊r2⌋)]; /* wait */
    copy_from_token<r0,s0>(data);
    inc_rd_idx<r0,s0>();
    [i1();] /* notify */
  }

void chx_read_handler<RCSemaphore r1, SCSemPost s6>() {
  [s6(⌊r2⌋)]; /* post */
}
```

Listing C.6 – Template for the read access module of $\texttt{IS}_4$.

Appendix D

Automated Programming Timing Measurements

This appendix contains the measured execution times for the experiments of section 4.4.2.3.

		Possible Implementations									
		10	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
Channels	**10**	1,204	1,016	0,928	1,116	1,340	1,747	1,608	1,844	1,921	2,153
	11	4,605	2,871	2,548	2,971	3,547	4,533	4,270	4,259	5,937	6,440
	12	8,053	8,586	8,705	10,114	10,120	11,375	12,705	13,464	14,866	16,055
	13	13,095	14,799	16,235	17,183	29,641	20,625	27,225	30,619	37,325	27,781
	14	35,471	93,643	94,017	91,167	66,594	83,464	80,004	98,477	81,980	107,126
	15	173,258	640,039	305,584	427,352	447,273	535,248	443,380	361,172	480,316	515,537
	16	620,546	1305,917	1646,053	2169,154	1660,172	1796,504	1660,121	2036,722	1594,032	1728,816
	17	709,392	1336,494	1363,059	1514,732	1612,444	1551,698	1396,173	1543,090	1521,742	1742,375

Table D.1 – Results – Cost per implementation scheme. All problems with 16 and 17 channels returned ”undefined”. All values are given in seconds.

Appendix E

Design Generation Timing Measurements

Increaing number of SoftwareQueues components						
Component Instantiation Restrictions						
Components.SoftwareQueue	1	2	3	4	5	6
Xilinx.EDK.XpsBramIfCntlr	1	1	1	1	1	1
Xilinx.EDK.BramBlock	3	3	3	3	3	1
Option Instantiation Restrictions						
Xilinx.EDK.Plb_v46.PlbSlave	4	4	4	4	4	4
Xilinx.EDK.Xilkernel.MessageQueue	3	3	3	3	3	3
Components.Context.Callee	5	5	5	5	5	5
Components.Context.Caller	5	5	5	5	5	5
Complexity						
Ex. 1 (variables)	497	560	615	670	725	780
Ex. 1 (expressions)	2056	2544	2966	3388	3810	4232
Ex. 2 (variables)	521	578	631	684	737	790
Ex. 2 (expressions)	2536	3022	3476	3930	4384	4838
Execution time (hh:mm:ss)						
Ex. 1	00:00:02	00:01:23	00:04:30	00:06:01	00:10:06	00:14:10
Ex. 2	00:00:07	00:01:25	00:04:09	00:10:08	00:08:37	00:14:04

Figure E.1 – Complexity and measured execution time of Yices for increasing number of SoftwareQueue components.

Increaing number of MessageQueue options					
Component Instantiation Restrictions					
Components.SoftwareQueue	1	1	1	1	1
Xilinx.EDK.XpsBramIfCntlr	1	1	1	1	1
Xilinx.EDK.BramBlock	3	3	3	3	3
Option Instantiation Restrictions					
Xilinx.EDK.Plb_v46.PlbSlave	4	4	4	4	4
Xilinx.EDK.Xilkernel.MessageQueue	5	10	15	20	25
Components.Context.Callee	5	5	5	5	5
Components.Context.Caller	5	5	5	5	5
Complexity					
Ex. 1 (variables)	541	651	761	871	981
Ex. 1 (expressions)	2340	3050	3760	4470	5180
Ex. 2 (variables)	573	703	833	963	1093
Ex. 2 (expressions)	3076	4426	5776	7126	8476
Execution time (hh:mm:ss)					
Ex. 1	00:00:05	00:00:22	00:00:45	00:01:12	00:01:51
Ex. 2	00:00:23	00:02:39	00:04:50	00:12:18	00:13:55

Figure E.2 – Complexity and measured execution time of Yices for increasing number of MessageQueue options.

Increaing number of Caller options					
Component Instantiation Restrictions					
Components.SoftwareQueue	1	1	1	1	1
Xilinx.EDK.XpsBramIfCntlr	1	1	1	1	1
Xilinx.EDK.BramBlock	3	3	3	3	3
Option Instantiation Restrictions					
Xilinx.EDK.Plb_v46.PlbSlave	4	4	4	4	4
Xilinx.EDK.Xilkernel.MessageQueue	3	3	3	3	3
Components.Context.Callee	5	5	5	5	5
Components.Context.Caller	100	200	300	400	500
Complexity					
Ex. 1 (variables)	1257	2057	2857	3657	4457
Ex. 1 (expressions)	4716	7516	10316	13116	15916
Ex. 2 (variables)	1281	2081	2881	3681	4481
Ex. 2 (expressions)	5196	7996	10796	13596	16396
Execution time (hh:mm:ss)					
Ex. 1	00:00:05	00:00:05	00:00:07	00:00:08	00:00:09
Ex. 2	00:00:11	00:00:14	00:00:15	00:00:20	00:00:22

Figure E.3 – Complexity and measured execution time of Yices for increasing number of Caller options.

Increaing number of Callee options				
Component Instantiation Restrictions				
Components.SoftwareQueue	1	1	1	1
Xilinx.EDK.XpsBramIfCntlr	1	1	1	1
Xilinx.EDK.BramBlock	3	3	3	3
Option Instantiation Restrictions				
Xilinx.EDK.Plb_v46.PlbSlave	4	4	4	4
Xilinx.EDK.Xilkernel.MessageQueue	3	3	3	3
Components.Context.Callee	10	20	30	40
Components.Context.Caller	5	5	5	5
Complexity				
Ex. 1 (variables)	557	677	797	917
Ex. 1 (expressions)	2396	3076	3756	4436
Ex. 2 (variables)	581	701	821	941
Ex. 2 (expressions)	2876	3556	4236	4916
Execution time (hh:mm:ss)				
Ex. 1	00:00:11	00:01:32	00:06:30	00:13:02
Ex. 2	00:00:22	00:01:25	00:03:17	00:11:54

Figure E.4 – Complexity and measured execution time of Yices for increasing number of Callee options.

Bibliography

[1] http://www.openmp.org/.

[2] Enterprise JavaBeans Specification. http://java.sun.com/products/ejb/docs.html.

[3] JavaBeans Specification. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.

[4] *The Future of Embedded Systems Technology*, 2005/06. BCC Research, Report G-229R.

[5] Nieuwland A., Kang J., Gangwal O.P., Sethuraman R., Busá N., Goossens K., Llopis R.P., and Lippens P. C-HEAP: A Heterogeneous Multi-Processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems. *Design Automation for Embedded Systems*, 7:233–270, 2002.

[6] S. Abdi, D. D. Gajski, and I. Viskic. Model Based Synthesis of Embedded Software. *Journal of Software*, 7:717–727, 2009.

[7] Accellera. SystemVerilog. http://www.systemverilog.org.

[8] R. Allen, S. Vestal, D. Cornhill, and B. Lewis. Using an architecture description language for quantitative analysis of real-time systems. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 203–210, New York, NY, USA, 2002. ACM.

[9] Altera. Quartus II Development Software Handbook: SOPC Builder. http://www.altera.com/literature/lit-sop.jsp.

[10] ARM. ARM AMBA. http://www.arm.com.

[11] Artemis Strategic Research Agenda Working Group. *Artemis Strategic Research Agenda*, 2006.

[12] V. Asokan. Dual Processor Reference Design Suite. http://www.xilinx.com/support/documentation/application_notes/xapp996.pdf, 2008.

[13] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, 2003.

[14] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

[15] R. Bergamaschi, W.R. Lee, D. Richardson, S. Bhattacharya, M. Muhlada, R. Wagner, A. Weiner, and F. White. Coral – automating the design of systems-on-chip using cores. In *Proceedings of the IEEE 2000 Custom Integrated Circuits Conference, 2000. CICC.*, pages 109–112. IEEE, 2000.

[16] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. 1996.

[17] G. Booch, J. Rumbaugh, and I. Jacobsen. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[18] L. Cai and D.D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM.

[19] W.O. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, Sungjoo Yoo, A.A. Jerraya, L. Gauthier, and M. Diaz-Nava. Multiprocessor SoC platforms: a component-based design approach. *Design and Test of Computers, IEEE*, 19:52–63, 2002.

[20] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabaey. UML and platform-based design. pages 107–126, 2003.

[21] P. C. Clements. A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[22] J. E. Coffland and A. D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 666–671, New York, NY, USA, 2003. ACM.

[23] S. Cook, G. Jones, S. Kent, and A. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[24] IBM Corperation. IBM CoreConnect. https://www-01.ibm.com/ chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture.

[25] A. Davare. Automated Mapping for Heterogeneous Multiprocessor Embedded Systems. Technical report, University of California at Berkeley, 2007.

[26] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A Next-Generation Design Framework for Platform-based Design. In *Design and Verification Conference (DVCON07)*, 2007.

[27] A. Davare, Q. Zhu, and A. L. Sangiovanni-Vincentelli. A Platform-based Design Flow for Kahn Process Networks. Technical report, University of California at Berkeley, 2006.

[28] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and Essink G. YAPI: application modeling for signal processing systems. In *Proceedings of the 37th Annual Design Automation Conference*, pages 402–405. ACM, 2000.

[29] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[30] Rainer Dömer. *System-level Modeling and Design with the SpecC Language*. PhD thesis, Universität Dortmund, 2000.

[31] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta. BALBOA: A Component-Based Design Environment for System Models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22:1597–1612, 2003.

[32] B. Dutertre and L. De Moura. The YICES SMT Solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[33] W. Ecker and M. Hofmeister. The design cube: a new model for VHDL designflow representation. In *EURO-DAC '92: Proceedings of the conference on European design automation*, pages 752–757, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[34] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel. Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design. *IEEE Transactions on Evolutionary Computation*, 10:358–374, 2006.

[35] M. Fränzle and C. Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.

[36] Martin G. Overview of the MPSoC design challenge. In *Proceedings of the 43rd annual conference on Design automation*, pages 274–279. ACM, 2006.

[37] D. D. Gajski and A. Gerstlauer. System-Level Abstraction Semantics. *System Synthesis, International Symposium on*, 0:231–236, 2002.

[38] D. D. Gajski and R. H. Kuhn. Guest Editors' Introduction: New VLSI Tools. *Computer*, 16:11–14, 1983.

[39] D. Garlan, R. T. Monroe, and D. Wile. *Foundations of component-based systems*, chapter Acme: architectural description of component-based systems, pages 47–67. Cambridge University Press.

[40] T. Genssler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich. Components for embedded software: the pecos approach. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26, New York, NY, USA, 2002. ACM.

[41] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[42] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the seventh international workshop on Hardware/software codesign*, pages 74–78, New York, NY, USA, 1999. ACM.

[43] J. Grode and J. Madsen. A Unified Component Modeling Approach for Performance Estimation in Hardware/Software Codesign. In *EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO*, page 10065, Washington, DC, USA, 1998. IEEE Computer Society.

[44] X. Guerin, K. Popovici, W. Youssef, F. Rousseau, and A.A. Jerraya. Flexible Application Software Generation for Heterogeneous Multi-Processor System-on-Chip. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 279–286, Washington, DC, USA, 2007. IEEE Computer Society.

[45] A. Hansson, K. Goossens, and A. Rădulescu. A unified approach to constrained mapping and routing on network-on-chip architectures. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2005. ACM.

[46] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. In *IEEE Proceedings – Computers and Digital Techniques*, volume 152, pages 148–166, 2005.

[47] D. V. Hung and P. H. Thai. Towards a Template Language for Component-based Programming. Technical report, The United Nations University, International Institute for Software Technology, 2007.

[48] Microsoft Inc. Distributed Component Object Model, DCOM. http://msdn.microsoft.com/library/cc201989.aspx.

[49] Object Management Group Inc. Common Object Request Broker Architecture, CORBA. http://www.corba.org/.

[50] Open SystemC Initiative. SystemC. http://www.systemc.org.

[51] ITRS. *International Technology Roadmap for Semiconductors (Design)*, 2007. http://www.public.itrs.net.

[52] Xu. J. and Wolf. W. Platform-based design and the first generation dilemma. In *9th IEEE/DATC Electronic Design Processes Workshop*. IEEE, 2002.

[53] A. Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[54] A. Jantsch, S. Kumar, and A. Hemani. The rugby model: a conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 54, New York, NY, USA, 1999. ACM.

[55] A. A. Jerraya, A. Bouchhima, and P. Frédéric. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *Proceedings of the 43rd Annual Design Automation Conference*, pages 280–285. ACM, 2006.

[56] B. Jonsson, E. Brinksma, G. Coulson, S. Graf, I. Crnkovic, S. Gérard, H. Hermanns, J.-M. Jezequel, A. Ravn, Ph. Schnoebelen, F. Terrier, and A. Votintseva. Roadmap: Component based design and Integration platforms. In *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *LNCS*. Springer, 2005.

[57] G. Kahn. The semantics of a simple language for parallel programming. *IFIP Congress 74*, 1974.

[58] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns andplatform-based design. In *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, volume 19, pages 1523–1543. IEEE, 2000.

[59] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[60] L. Kriaa, A. Bouchhima, M.-W. Youssef, F. Petrot, A.-M. Fouillart, and A.A. Jerraya. Service based component design approach for flexible hardware/software interface modeling. In *Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping, 2006.*, pages 156–162. IEEE, 2006.

[61] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[62] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17:1217–1229, 1998.

[63] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. System level design with spade: an M-JPEG case study. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 31–38, Piscataway, NJ, USA, 2001. IEEE Press.

[64] S. Mahadevan, K. Virk, and J. Madsen. ARTS: A SystemC-based framework for multiprocessor Systems-on-Chip modelling. *Design Automation for Embedded Systems*, 11(4):285–311, 2007.

[65] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 60–76, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[66] G.E. Moore. Multidimentional synchronous dataflow. *Electronics*, 38, 1965.

[67] Kathy Dang Nguyen, Zhenxin Sun, P. S. Thiagarajan, and Weng-Fai Wong. Model-driven soc design via executable uml to systemc. *Real-Time Systems Symposium, IEEE International*, 0:459–468, 2004.

[68] F. Nielson, R. H Nielson, and C Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.

[69] R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. *Theory and Applications of Satisfiability Testing, SAT 2006, Lecture Notes in Computer Science*, 4121:159–169, 2006.

[70] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and Automated Multiprocessor System Design, Programming, and Implementation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 27, pages 242–555. IEEE, 2008.

[71] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 574–579, New York, NY, USA, 2008. ACM.

[72] C. Norström, K. Sandström, J. Mäki-Turja, M. Gustafsson, and N.-E. Bånkestad. Experiences from introducing state-of-the-art real-time techniques in the automotive industry. *IEEE International Conference on the Engineering of Computer-Based Systems*, 0, 2001.

[73] E. Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Acta Polytechnica Scandinavica, 1995.

[74] J. M. Paul. Programmers' views of SoCs. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2003, CODES+ISSS '03*, pages 156–181. ACM, 2003.

[75] P. G. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A System-Level Exploration Platform for Network Processors. *IEEE Design & Test*, 19(6):17–26, 2002.

[76] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, and G. Nicolescu. Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 48–53, New York, NY, USA, 2004. ACM.

[77] A. D. Piementel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and F. E. Deprettere. Exploring Embedded Systems Architectures with Artemis. *Computer*, pages 57–63, 2001.

[78] A. D. Pimentel, C. Erbas, and S. Polstra. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Transactions on Computers*, 55:99–112, 2006.

[79] Paul Pop, Petru Eles, Zebo Peng, and Traian Pop. Analysis and optimization of distributed real-time embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 11(3):593–625, 2006.

[80] M. Raulet, F. Urban, J.-F. Nezan, O. Deforges, and Y Sorel. Rapid Prototyping for Heterogeneous Multicomponent Systems: An MPEG-4 Stream over a UMTS Communication Link. *EURASIP Journal on Applied Signal Processing*, pages 1–13, 2006.

[81] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A UML 2.0 profile for SystemC: toward high-level SoC design. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 138–141, New York, NY, USA, 2005. ACM.

[82] I. Sander. Platform-Based Design of Heterogeneous Embedded Systems, 2009. Presentation, Real-Time in Sweden (RTiS 2009).

[83] Alberto Sangiovanni-Vincentelli. Defining Platform-based Design. *EEDesign of EETimes*, February 2002.

[84] A. Sarmento, L. Kriaa, A. Grasset, M.-W. Youssef, A. Bouchhima, F. Rousseau, W. Cesario, and A. A. Jerraya. Service dependency graph: an efficient model for hardware/software interfaces modeling and generation for SoC design. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 261–266, New York, NY, USA, 2005. ACM.

[85] G. Smith. Platform based design: does it answer the entire SoC challenge? In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 407–407, New York, NY, USA, 2004. ACM.

[86] Y. Sorel. Real-time embedded image processing applications using the A^3 methodology. In *International Conference on Image Processing, Proceedings*, pages 145–148, 1996.

[87] P. V. B. Sørensen and J. Madsen. Component-based Service Availability Checking. In *Proceedings of the Nordic Workshop and Doctoral Symposium on Dependability and Security*. Institute of Cybernetics at Tallinn University of Technology, 2008.

[88] P. V. B. Sørensen and J. Madsen. Consistency Check for Component-Based Design of Embedded Systems using SAT-Solving. In *Proceedings of the 20th Nordic Workshop on Programming Theory*, pages 93–96. Institute of Cybernetics at Tallinn University of Technology, 2008.

[89] P. V. B. Sørensen and J. Madsen. Generating Process Network Communication Infrastructure for Custom Multi-Core Platforms. *International*

Journal of Embedded and Real-Time Communication Systems (IJERTCS), page To appear, 2010.

[90] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 9–14, New York, NY, USA, 2007. ACM.

[91] S. Todor, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using Khan process networks: the Compaan/Laura approach. In *Design, Automation and Test in Europe Conference and Exhibition, 2004, Proceedings*, volume 1, pages 340–345. IEEE, 2004.

[92] A. S. Tranberg-Hansen and J. Madsen. A Service Based Component Model for Composing and Exploring MPSoC Platforms. In *First International Symposium on Applied Sciences on Biomedical and Communication Technologies, 2008. ISABEL '08.*, pages 1–5, 2008.

[93] W.-T. Tsai, X. Wei, R. Paul, J.-Y. Chung, Q. Huang, and Y. Chen. Service-oriented system engineering (SOSE) and its applications to embedded system development. *Service Oriented Computing and Applications*, 1:3–17, 2007.

[94] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 206–217, New York, NY, USA, 2004. ACM.

[95] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33:78–85, 2000.

[96] C. Xi, L. JianHua, Z. ZuCheng, and S. YaoHui. Modeling SystemC design in UML and automatic code generation. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 932–935, New York, NY, USA, 2005. ACM.

[97] Xilinx. Xilinx Platform Studio and Embedded Development Kit. http://www.xilinx.com/tools/embedded.htm.

[98] Xilinx. Xilinx System Generator for DSP. http://www.xilinx.com/ tools/sysgen.htm.

[99] Xilinx. PowerPC Processor Reference Guide, UG011 (v1.2). http://www.xilinx.com/support/documentation/user_guides/ug011.pdf, 2007.

[100] Xilinx. Embedded System Tools Reference Manual, Embedded Development Kit, EDK 10.1, Service Pack 3. http://www.xilinx.com /support/-documentation/sw_manuals/edk10_est_rm.pdf, 2008.

[101] Xilinx. Microblaze Processor Reference Guide, Embedded Development Kit, EDK 10.1i. http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf, 2008.

[102] Xilinx. Xilkernel (v4.00.a). http://www.xilinx.com/support/docu mentation/sw_manuals/xilinx11/oslib_rm.pdf, 2009.