ESWeek Workshop
Tampere, Finland, October 7th, 2012

MeCoES -
Metamodelling and Code Generation for Embedded Systems

Proceedings

Editors:
Wolfgang Mueller, University of Paderborn, Germany
Wolfgang Ecker, Infineon Technologies, Germany
# Table of Content

**Metamodeling and Code Generation - The Infineon Approach**  
Wolfgang Ecker, Michael Velten, Leily Zafari, Ajay Goyal,  
*Infineon Technologies AG, DE* ......................................................................................1

**Framework for Content Engineering of HW IP in Renesas Mobile**  
Tommi Mäkeläinen, Arto Hanikäinen, Anssi Marttila, Joumi Siirtola, Mika Naula,  
*Renesas Mobile Corporation, Tampere, FI* ...................................................................5

**Domain-Specific Modeling Languages for Embedded System Development**  
Juha-Pekka Tolvanen, Steven Kelly, *MetaCase, FI* ........................................................11

**Kactus2: Extended IP-XACT Metadata Based Embedded System Design Environment**  
Antti Kamppi, Joni-Matti Määätä, Lauri Matilainen, Erno Salminen, Timo D. Hämäläinen;  
*Tampere University of Technology, FI* ........................................................................17

**An IP-XACT-TO-SystemC Model Generator for Mutation Analysis**  
Tao Xie, Wolfgang Mueller, *University of Paderborn/C-LAB, DE* .................................23

**Combining Various Meta-Models for the Generation of Control Structures in Hardware**  
Rainer Findening¹, Thomas Steininger², Thomas Leitner³;  
¹*Upper Austrian University, AT*; ²*Intel Mobile Communication, DE*; ³*Danube Mobile Communications Engineering, AT* ........................................................................29

**Meta-Modelling the SystemC Standard for Component-Based Embedded System Design**  
Andreas Burger¹, Martin Küster¹, Alexander Viehl¹, Oliver Bringmann¹², Wolfgang Rosenstiel¹²;  
¹*FZI, DE*; ²*University of Tübingen, DE* ........................................................................35

**Code Generation for QEMU/SystemC Cosimulation from SysML**  
Fabian Mischkalla, Wolfgang Mueller; *University of Paderborn/C-LAB, DE* ...................41

**Code Generation for Component-Based Applications**  
Ansgar Rademacher, Chokri Mraidha, *CEA-LIST, FR* ..................................................46.

**UML/MARTE Methodology for High-Level System Concurrency Exploration and Synthesis**  
Hector Posadas, Pablo Penil, Alejandro Nicolas, Eugenio Villar;  
*Universidad de Cantabria, ES* .......................................................................................52

**Meta Constraints for Consistency Checks of Embedded System Specifications**  
Andreas Burger¹, Wolfgang Ecker², Oliver Bringmann¹², Wolfgang Rosenstiel¹²;  
¹*FZI, DE*; ²*University of Tübingen, DE*; ³*Infineon Technologies AG, DE* .................58

**Verification of Embedded Systems Using Modeling and Implementation Languages**  
Mathias Soeken¹², Heinz Riener¹, Robert Wille¹, Görschwin Fey¹³, Rolf Drechsler¹²;  
¹*University of Bremen, DE*; ²*DFKI, DE*; ³*DLR, DE* ..................................................67
ABSTRACT
This paper presents current application of Metamodeling and Code generation at Infineon. The technique helped to save up 95% time for specific design tasks and up to 60% in the implementation of a complete chip. Key for the success is the fast prototyping of a first generator chain, the open framework, and the adaptability to specific formats and styles.

Categories and Subject Descriptors
J.6 [Computer-Aided Engineering]: Computer-aided Design

General Terms
Standardization, Languages, Design

Keywords
Metamodel, Template, Code Generation

1. INTRODUCTION
Metamodeling and code generation has a long history and evolution at Infineon. Several visions, ideas, concepts, and approaches lead to Infineon’s metamodeling and code generation framework called “Metagen”.

One root of metamodeling is the migration of concepts of a semiconductor or car fab to design, so to say to establish a design factory. Taking a look at a car fab, the first productivity boost came from Ford’s invention of an assembly line. Instead of covering the complete construction process by one person or a handset of people, the construction process was split in many small steps, each requiring only a small amount of knowledge. Specialization found its way to manufacturing. The second productivity boost was achieved by using robots, so to say by automation. – Chip design followed similar trends. The layout of early chips was handcrafted and verification was done by visual inspection. Chip designs were done by a small number of people. Since more and more people got involved in design, specialization found its path to chip design: Digital hardware, analog hardware, firmware, software, concept, architecture, verification and many more tasks were split to different people and later even to different teams. Also automation, the so called EDA – electronic design automation – found its way to chip design. Huge productivity boosts were established however, since the introduction of RTL synthesis and partially IP reuse, no technique came up that improved design productivity in a wide range of application (as RTL synthesis did). Only very specialized tools such as high level synthesis or ASIP synthesizer came up which brought further productivity in special fields. But these techniques are too tailored to serve a wide field of applications and too rigid to interact with other tools. More adoptable and interoperable point tools would have been needed but they would conflict with the initial idea of EDA industry: making one solution that fits many customers, which leaded them to replace many in-house tools about 25 years ago:

Second driver is a new trend in design. Figure 1 shows the classical “miniaturization” trend of Moore’s law but also a second trend “diversification” – named as “More than Moore”. Even more specific automation solutions are needed for products in the field of analog RF, high voltage, or low power. But most of the EDAs still serve the “More Moore” trend.

As Infineon focuses on energy efficiency, mobility, and security products with target markets automotive, industrial power control, power management and multimarket as well as chip card and security, the demand for automation supporting the diversification trend is quite high.

Finally, in SW domain, early automation tools – called CASE tools - had to fight with comparable problems. The generated code did not fit the needs, the initial descriptions didn’t provide sufficient descriptive power as well as the descriptions, tools, and generated code didn’t fit together. But solutions for the problems were worked out under the umbrella of OMG [2]; UML as a standard language [3], MDA as a flexible process incorporating flexible code generation [4], and metamodeling to allow creation of owns formalisms [5]. Instead of providing one solution that fits all, domain specific solutions come in focus.
2. Terminology

The term “Meta” is often associated with something esoteric and therefore the term “Metamodeling” leads to some skeptics when designers are faced with. This is a pity and unjustified. The term "Meta" is derived from the Greek language and simply means "beyond". As shown in Figure 2, a “Metamodel” describes a model which is beyond – or more abstract - than a model. In other words, a Metamodel defines the structure of a model. A Metamodel, hence defines a so called abstract DSL (domain specific language) – a concrete DSL would be a computer language that describes an abstract DSL in a textual way. XML schema [6] is such a language as well as SystemRTL [7] in the context of registers. Editors, GUIs, or even concrete DSLs can be automatically derived from an abstract DSL. Also APIs – a structured and systematic way to access a model – can be generated from an abstract DSL.

Since a Metamodel describes objects, their attributes, relations, and properties, it can be captured in a class like diagram. For example UML class diagrams can be used but there are better fitting specific formalisms as ECORE [8] or MOF [9].

![Figure 2 - Abstracted OVM testbench](image)

Using the generated API, the metamodel can be filled. Consequently an instance of the metamodel is built. This instance of is called a model. In turn a model follows the structure of the metamodel. A GUI but also specification formats as Framemaker or EXCEL can be used to build such a model.

The model on the other hand defines the content of one or more views in an abstract way. From that model, different views such as documentation (HTML, ...), hardware description (VHDL, RTL, ...), verification items (SVA, ...), software code (C-header) or other documents can be generated.

It becomes clear now, that the term model is used in the context of metamodeling slightly differently than in hardware design. In hardware design, a model – as an RTL- or TLM-model - is an abstraction of hardware but in metamodeling, RTL- or TLM-models are treated as views. The metamodeling aspect of model definition is used further in this paper.

3. Incremental Introduction of Metamodeling at Infineon

Generation of code in the context of registers is in place at Infineon for over a decade or even longer. To provide a company wide solution, an alignment was started to define a core metamodel and an infrastructure to handle that metamodel. IP-XACT – a standard supporting electronic IP databooks and IP integration - was considered as one candidate for that metamodel but it had several deficiencies: No common RTL/TLM view, no own type system, structural changes via versions, and most important, no information about internal details. But, an IP-XACT import and export to and from our metamodel was implemented to have the best of both worlds: standard IP description and powerful generation capabilities. The framework was developed with Java and JXBY. Also full EDA solutions were considered but they did not have sufficient capabilities and their generated code was incompatible with our coding styles.

In parallel, a vcd toolset was developed and enhanced with the Mako template engine [10] for flexible code generation of verification environments. The template engine was then also linked with the previously mentioned core model via JAVA/Python interface. Code generators for different formats and different coding styles could be built very efficiently.

As a next step, the template engine was linked to a hand coded API of a model (in the sense of metamodeling) of an ASIP. The API implemented in Python mimicked a hand drawn metamodel upfront drafted on a piece of paper. The model was filled via API from an EXCEL specification. Various views as RTL code, formal properties, documentation, behavioral model and many more were generated. Depending on the view, savings up to 80% could be achieved considering the effort of making the API and the generators on the one hand side and the manual effort that would have been needed to code on the other side. In addition, several specification errors were detected, since reading the EXCEL and filling the model required the coding and execution of several consistency checks as well. Metamodeling had proven its effectiveness in HW design even since a hand crafted metamodel was used as a starting point.

But the direction was clear and in two days, a prototype framework was implemented that transformed a Metamodel specified in UML syntax (but with additional semantic) in an API. Also here Python and Mako were used.

4. The Metagen Environment

4.1 Metamodeling Framework

The metamodeling idea changed the way of design – at least in several applications. Instead of typing the target code, a Metamodel is designed, specification readers (=model fillers) are implemented, and generators are coded. The manual work “typing” is so to say replaced by a specially designed “typing robot”.

Besides providing an API for model handling, the framework also includes tools for gluing generated GUIs, readers and generators together.
4.2 Implementation Aspects
As already mentioned, UML was used to capture the metamodel. We initially used the tool Bouml [11] but interfaced the API Generator via XMI (XML Metadata Interchange), so that other UML tools such as enterprise architect [12] can be used to capture the metamodel as well.

We used Bouml in its open source version since it was scriptable (i.e. XMI and the internal formats could be automatically in sync) and it was available for all our development platforms (windows, cygwin, linux, solaris).

As already mentioned before, the template engine MAKO is mainly used for code generation. MAKO was selected since it showed to be the most powerful template engine in our benchmark. Special features like template inheritance or decorators were convincing. Another benefit was the ability to use full Python for computations in the context of code generation.

When speaking about Metamodelling, the focus lies more on concepts than representations meaning languages. Nevertheless the selection of Python as core language of the environment showed as very beneficial. When ramping up metamodels, we estimate to be at least a factor of 2x faster compared to classical OO languages as Java or C++. Reason is the interpreted nature and the polymorphic object approach. Class inheritance hierarchies to establish polymorphism, introspection, or extension as needed in OO languages mentioned above are often not necessary. Further on Python is a small language (e.g. compared to PERL) but with rich base types as strings (very important for generation), lists, dicts, ordered sets and many others and on top a huge number of powerful libraries such as regular expressions, XML handling, parsing, and others. Last but not least, Python has a helpful documentation freely available on the internet. Googling “python” and a list of items describing the problem often gives the result on the first page.

5. Application Example
5.1 Basic Framework
Metamodelling is used in several applications amongst the presented application for ASIPs also for OVM/UVM testbench generation or power modeling.

Nevertheless, we use the classical register description as application example since it is best known.

5.2 Filling the MetaModel
Code generation starts with filling the metamodel. In our case, specification is mostly done in Framemaker or EXCEL. A set of document formats is supported, but the open nature of the Python based framework allows to adopt to own styles as well. This is especially important if specifications are given by the customers. Of course IP-XACT register descriptions can be read as well. A plugin mechanism supports on the fly replacement of the single readers.

Similarly, different writers can be linked to the generation engine but mostly MAKO templates are used to describe the intended outputs. Of course, template hierarchies and template modules are used to structure code generation. However, best benefit can be taken from templates if existing code of a view shall be statically inserted in the generated code, or starting from an existing view code, a generic set of code pieces shall be developed.

Also here the flexibility pays off. So, we were able to implement a generator prototype for a customer specific register format in less than two hours.

5.3 Component Model
A sample of a component model is shown in Figure 5. Amongst others, a component has parameters (that map e.g. in SystemC to templates and in VHDL to generics) and interfaces (that map e.g. in SystemC to TLM interfaces or in RTL to bundles of
signals). Registers are associated with interfaces, bitfields with registers, and so on.

The generated API provides now setting and getting of attributes together with (not shown in Figure 5) more advanced query and sort functionality. Based on this interface, basic checks can be formulated.

When the metamodel is populated with the data, checks can be carried out on the model. For example, as shown in Figure 5, it should be checked if offset + value > datawidth. The advantage of carrying out these checks on the model is that they can be implemented independent of the spec format, which increases the reusability of our framework.

5.4 Code Generation
Also code generation makes use of the generated API. First, the root node or another node is passed to a template. In the template, iterative and alternative rendering constructs are used to establish proper code generation.

6. Summary and Outline
We presented metamodeling and code generation methodologies and the Metagen framework as used at Infineon. Savings up to 80% for specific design tasks have been already established using a very rudimental version of the framework. Up to 95% of savings for specific tasks have been reported in follow up applications.

Considering the complete implementation (specification ready to RTL freeze), up to 40% savings could be achieved in a first project and up to 60% in a second project utilizing and enhancing the generator chains of the first project.

Also quality increase could be achieved by metamodeling. But this is more a qualitative than a quantitative statement.

Key for success of metamodeling technique is the rapid establishment of a first generator chain (often in hours) thanks to the framework, Python and MAKO and the possibility to adapt to existing flows and coding styles.

But, metamodeling is not a competition to todays EDA and EDA related standards it is an orthogonal add on allowing automation in fields, where general tools are not available.

We are convinced that metamodeling also will help in the future to be more productive since it was not introduced at one shot but evolved over several years. Many ideas and concepts found themselves in metamodeling.

But we guess that we still are at the beginning of metamodel based embedded system design. The technology has a huge potential but also a need for innovation. Verification of metamodels or automated backtracking from code to specification are only some aspects.

7. REFERENCES

ACKNOWLEDGMENTS
Many thanks to the many colleagues all over Infineon that contributed to the work by following the Metamodeling idea and by applying the techniques in various applications. Many thanks also also to our partners in the SANITAS project for their open discussions and technical contributions to the Metamodeling techniques. The Project SANITAS (01M3088) by the Federal Ministry of Education and Research funded parts of this work.
Kactus2: Extended IP-XACT metadata based embedded system design environment

Antti Kamppi¹, Joni-Matti Määttä¹, Lauri Matilainen¹, Erno Salminen¹, Timo D. Hämäläinen¹

¹ Tampere University of Technology, Department of Computer Systems, Korkeakoulunkatu 1, FI-33720 Tampere, Finland
{firstname.familyname}@tut.fi

Abstract. IP-XACT/IEEE1685 is a standard for System-on-Chip (SoC) integration. Its goals are vendor, tool and language neutral description of IP-blocks and designs in XML metadata. The standard aims to simplify IP integration and configuration as well as increase tool interoperability. Kactus2 is the first graphical open source IP-XACT tool. Kactus2 extends the standard to support SW, SW to HW mappings and application communication abstraction using, e.g. Multicore Association MCAPI. Kactus2 is implemented in C++ and QT4 framework and includes tools to packetize components and integrate them. For example VHDL, Modelsim, and Quartus files are generated automatically for implementation, HTML for documentation and later there will be generators for also SW build. With IP-XACT and our Kactus2 extensions, porting applications between HW and SW or between different platforms becomes a matter of hours compared to weeks with non-metadata based design methods.

1 Introduction

According to semiconductor manufacturers’ roadmap (ITRS), the SoC design productivity must increase 10x and the design re-use exceed the 90% limit by 2022 to keep up with the 15x logic size increase [1]. We can apply this trend also to embedded systems product development, especially for FPGA-based Multiprocessor System-on-Chip (MP-SoC) designs that include both HW and SW components in a highly reconfigurable and reusable manner.

Current System-on-Chip designs may include hundreds of Intellectual Property (IP) blocks, of which over four may be processor cores [2]. From the software point of view, stacks of hundreds of layers and APIs and thousands of programs and libraries exist. To estimate the SW complexity, Android OS has over 15M lines of code (LoC) and its development effort is estimated to be 4.677 person years by Sept. 2012 [3].

The overall complexity has been attacked by well-known platform-based design [4] as well as model driven engineering (MDE) [5] for raising the design abstraction. Ideally there is a high-enough description of both the required application functionality and available execution platforms. After that the implementations could be automated, and both the applications and platforms could be easily reused.
However, according to our experience, there are **five main obstacles** on the road to the automated, model based design:

1. There are too many paradigms and domain-specific models to choose from. Each needs long time to learn and conversion between them is too cumbersome or outright impossible. Moreover, many users fear losing control over e.g. real-time behavior when low level details are hidden, or the metamodel does not cover their needs.

2. Modeling is a separate side-task not fitting to the whole flow, for example used mainly for extra documentation in addition to “real” implementation steps. Tools with custom, often changing, and even encrypted formats further complicate creating a seamless design flow.

3. Massive legacy code base and multitude of versions and variants are not adequately accounted. Designers won’t have time and money to manually create models from the old code, but without it the model based design cannot be adopted at all.

4. One-way automation from top to bottom works only with ideal tools and limited use case. In practice, user often likes to manually modify at least some parts of auto-generated code, but this is often impossible. Propagating the changes upwards does not usually exist or work.

5. High abstraction often means graphical models, but unfortunately many visual models are difficult to create and compare. Version management is easier with textual format but is less descriptive power in structural design. Graphical editors are complicated to use and design artifacts get confused due to unclear separation of shapes for different meaning.

This paper presents our approach to solve these issues in a tool framework called Kactus2 [6]. The next section explains background, the basic ideas and motivation to utilize IEEE 1685 standard (IP-XACT) [7] as the basis of our solutions. Section 3 presents our extensions to IP-XACT and Section 4 the Kactus2, the tool implementing our approach. Status of our work and future directions are concluded in the last Section.

## 2 Our approach

We have previously successfully demonstrated the high-level design flow, in which the design entry was SDL and later UML2.0 diagrams with asynchronously communicating Finite State Machines model. Complete design flow was demonstrated in practice for a MP-SoC on FPGA [7]. Several similar proposals exist, representing the top-down approach in Figure 1, and often suffer the problems mentioned in Section 1.

For Kactus2, we focused first on intermediate metadata formats, which allow making models from different source formats and abstraction levels [9]. We also changed the primary focus from pure *auto-generation to auto-assembly* as depicted in Figure 1. Currently, we first perform structural modeling using metadata, which is neutral to implementation language, tool, and IP vendor. Later we can include behavioral modeling by embedding other models like UML/MARTE and timing diagram markup lan-
Our approach is based on IEEE1685/IP-XACT standard, which is originally meant for hardware IP-block integration with XML-metadata descriptions.

Components, bus definitions and designs are basic independent objects that have \{Vendor, Library, Name, Version\} identifier (VLNV) and can be referenced to build up the systems. Metadata for an IP component includes, for example, the associated files, interfaces, possibly the internal structure, and assisting generator scripts.

For example, ST Ericsson has published an IP-XACT case study on a SoC IP called IZARN-IP that includes an ARM CPU and is targeted to SoC for mobile phones [10]. The objective was to speed-up reuse of this IP in future design integration that normally take several months. The first time delivery with IP-XACT took 6.5 person months that is 2.5 months more than the traditional method. However, after that the libraries can be reused and next changes and integration is carried out in a week. After the initial setup, the speedup was 24x. As a specific example, the effort to make an RTL change to bus matrix structure was reduced about 50% with IPXACT.

Xilinx has recently adopted IP-XACT in all of their IP libraries, which expands use of IP-XACT greatly from ASIC vendors to embedded systems. Unfortunately, Xilinx has also introduced over 100 vendor extensions, which is a standardized method to add new properties to IP-XACT.

Our approach uses basic IP-XACT for HW integration for documenting the product structure, e.g. circuit boards. Moreover, we consider the SW and system domains as well. Figure 2 depicts our new objects with the VLNV identity and how they extend the original IP-XACT in three aspects [11]. The most important is inclusion of SW in the “implementation” axis that follows the platform layer model depicted in Figure 3.

An SW component encapsulates files, language and tool directives and settings like HW components. SW design is collections of SW components and can be hierarchical just like the HW designs. SW component can have API definitions which are similar to bus and bus definitions in HW. They used in three ways: a) validate API between SW components b) validate source code against API definition and c) assist code writing in Kactus2 editors.
There are also COM interfaces which describe which application components communicate. We can use, for example, Multicore Association MCAPI communication abstraction where each component has MCAPI endpoints that are connected via MCAPI channels. Our specialty is that MCAPI can be included in both SW and HW components. Visually, connecting API and COM interfaces resembles the basic IP-XACT HW design. In summary, a SW component may have ports for both API and COM, but their use is different (function calls vs. inter-process communication).

Kactus2 System design is a hierarchical IP-XACT view of the system’s HW/SW components. SW architecture includes application, application-independent SW platform, and optionally the MCAPI endpoints and channels. This is stored to the topmost HW design as a new system view. If the HW design is hierarchical, Kactus2 automatically seeks all and shows all execution units, such as processors cores, as potential placeholders for mapping the SW components.
3 Kactus2 tool

Kactus2 combines component packetizing editors, designers to integrate the components, and generators to create documentation and codes for simulation and implementation. Kactus2 attempts to solve the earlier mentioned model-based design problems as follows.

Problems 1 & 2. Kactus2 uses IP-XACT and our extensions for the abstraction that focuses on structural modeling. We heavily use hierarchy and encapsulation of information to simple looking graphical models. We allow also incomplete designs by including draft components together with fully packetized ones. This way Kactus2 can also be used for sketching, but there is still a direct path to implementation.

Problem 3. Creating IP-XACT component descriptions for legacy code must be simple. Such tools are freely available and Kactus2 can include them as plugin. Note that metadata-based design can be adopted gradually by first using it as only documenting the product structure using quickly incomplete components. The more content is added, the more benefits are gained.

Problem 4. Kactus2 can create application code from MCAPI endpoint and channel definitions and user adds his code to the stub. If either generated and manually created part changes later, the potential inconsistency problems are highlighted. This way the graphical and textual models remain in synchrony. Nevertheless, this problem requires more work to be solved.

Problem 5. Kactus2 uses guided layout for the graphical elements. The column-wise layout with expansion downwards helps visualizing the differences in two designs. This feature has been only partially implemented so far.

The roadmap for Kactus2 is depicted in Figure 4 to summarize the solutions to the problems. Significant leap was taken when the extensions in this paper were implemented, thus we moved directly from version 1.4 to 2.0.

![Figure 4](image_url)

Figure 4 Implemented (green) and planned Kactus2 milestones.

Kactus2 targets also the best user experience among ESL tools. We address it by three aspects. 1) It is human friendly: it uses context to help user to never get lost and
on the other hand hides all possible design paths and shows only information relevant to the stage at hand. 2) It is wrist friendly: it minimizes mouse clicks and deep menus combined. 3) It is “brain friendly”: it hides as much as possible awkward IP-XACT terms and XML tree structure from users and assists in dialogs and code editors to fill in the right information.

Kactus2 is implemented in C++ and QT4 framework. Special attention has been paid on very easy installation, since many other tools require separate compilation, environment setup or virtualization to run on user’s computer.

4 Conclusions

IP-XACT standard includes 265 XML tags with 64 unique attributes. Our extensions add 70 tags and 28 attributes, which we consider reasonable low for avoiding bloating of the metamodel. Still we are able to cover all essential aspects. In a study we showed how Kactus2 can speeded-up the porting of a video encoder application between HW and SW, between PC and embedded system, and between different FPGA vendors chips [11]. Such tasks are matter of hours with extended IP-XACT metamodel.

References

Domain-Specific Modeling Languages for Embedded System Development

Juha-Pekka Tolvanen, Steven Kelly

MetaCase
Ylistönmäentie 31, FI-40500 Jyväskylä, Finland
{jpt, stevek}@metacase.com

Abstract. Metamodels and generators are typically applied to define development environments that produce code and other software related artifacts. Based on our experience they can also be used to produce other non-software related artifacts, enable software and hardware co-design, support early level design space exploration, and automate testing activities. In this paper we describe – based on implementations done by users of MetaEdit+ tool – various kinds of embedded system development situations that benefit from metamodeling and generators.

1 Introduction

Metamodeling and code generators are typically applied to improve the productivity and quality of software development [13]. A metamodel defines one or more modeling languages for a specific area of interest raising the level of abstraction from the solution domain, code, world to the problem domain. Code generators provide then automation by reading the models created with the language to produce various kinds of artifacts like code, configuration, test data and documentation. Not all metamodels, though, lend themselves similarly to code generation as languages that do not focus on a specific domain being general purpose, such as UML and SysML, can not raise the level of abstraction and can’t guarantee that the models created are complete and correct to enable code generation. Instead, the closer the language, and its metamodel, maps to the domain of interest, the more likely the models describe the system correctly and generators can then be applied efficiently. Often the most powerful modeling languages for code generation are those applied within a single company only.

The creation of domain-specific languages for modeling has become more widely used because modern metamodeling tools, for a review see [3], have removed the burden of tool creation and maintenance from the language creator: they provide fully functional modeling environments for a given language without any particular tool development activities. Industry-strength tools also enable experimentation with the language as it is built and can update models based on metamodel change: Work is not lost when moving to newer language versions.
Since metamodels allow defining domain-specific languages for various kind of situations we have seen the approach applied in many domains, including consumer electronics [5, 12], industrial automation [11], railway systems [8], mobile devices [1], telecommunication [1], and home automation [1, 9]. Not all these cases deal with code generation only. We start by describing the benefits that metamodels and code generators have provided – as reported by the users of MetaEdit+ [6] tool. Then we move beyond code generation and present other typical areas that benefit from metamodels and generators too.

2 Industry experiences

Raise in the level of abstraction provides many benefits and among the industry experience reports improved productivity is perhaps the most often claimed. While many companies report or provide anecdotes about productivity improvements, luckily some companies have also reported publicly their data. Kieburtz et al. [3] compared the use of domain-specific languages and code generators in experiments covering more than 130 development tasks in the domain of message translation and validation. Comparison to manual coding practices showed 300% increase in productivity. In the study the differences in the average performance of the subjects are statistically significant at confidence levels exceeding 99%. Usually most companies do not have resources and time to conduct as comprehensive studies and need to rely on more practical approaches. Panasonic [12] compared languages and generators they build for developing touch screen applications against manual programming by building most of the functions of the same touch screen device twice. The comparison to manual programming showed that modeling and code generation was over 400% faster. Polar [5] implemented languages for their sports computer applications along with code generators. In a laboratory study 6 engineers build individually a typical sports computer feature. After recording the development time they measured in average 900% increase in productivity. After extending the comparison by building a large portion of the sports computer functionality, similarly to [12], productivity increase was measured to be over 1000%.

Metamodels do not only capture domain concepts, but they may also cover domain rules. This allows languages to detect errors in the models and even prevent errors to occur in the first place. Since the impact of development approaches to the quality is harder and more laborious to measure than productivity, companies often rely on their engineers’ opinions: at EADS it was seen that the quality of the generated code is clearly better because the modeling language rules out errors, eliminating them already in the design stage [1]. At Polar, programmers compared the generated code to the available manually written code and found the generated code having clearly better quality [5]. Kieburtz et al. [3] studied the quality of the code by analyzing errors found in generated code and in manually written code. Their statistically significant data showed that generated code had 50% less errors.

While other desirable characteristics have also been reported, like easier understanding of models [1, 8] and easier introduction of new developers [5], we focus in the following to the use of generators for producing other than plain code.
Chapter 3: Beyond Plain Code Generation

The use of metamodels is obviously not limited to automating software development tasks, but can also be applied for automating and supporting various system development tasks. We describe below some of these as been implemented in MetaEdit+ metamodeling and generator environment.

**Generation of non-software artifacts.** One obvious approach has been to extend the domain-specific languages to cover also parts of the non-software system. For example, in cases of developing industrial automation systems (e.g. fishing farm [11]) and home automation systems (heating system [14], Figure 1), the generators also produce other than software related artifacts, like hardware mappings, network device configuration, wiring plans, material usage, installation guidelines and even labels for a wiring cabinet. Generation of these artifacts from the same and single source brings major benefits, including already mentioned productivity and quality improvements.

![Model of the system structure (right), model of the application behavior (left) [14]](image)

The metamodels are in these cases extended to cover the system characteristics of interest. As an example, in Figure 1 the characteristics of the pipes and instruments of the heating system (right) are specified in addition to the behavioral models (left) used for code generation. These two models are not made with separate languages but the
languages share the same concepts and allow sharing the same model elements since they are based on common metamodel. Also domain rules are checked with the combined models.

Generators may also use the combined models to analyze the system in more detail. A particularly important part in embedded system development is generation of simulations as it enables addressing situations that would not be practically even possible to reach with the real system – being for example too dangerous or expensive to realize. Also analysis of safety characteristic and techniques like failure analysis can be addressed easier as generators decrease the cost for analysis and simulation.

**Hardware and software co-design.** Concurrent development of both hardware and software has always been of great importance. Metamodels may cover these two worlds to support co-design. Such approaches enable SW/HW allocation work and facilitate examination of different integration approaches. Figure 2 below shows an example of two different kind of architecture models from automotive: one addressing functional architecture (left) and another addressing hardware architecture (right). To support allocation among these, a dedicated language (shown as an allocation matrix in model level below) has been defined [7]. Such metamodels can cover the rules of allocation and generators can be applied to analyze different allocation options or produce the allocation information into existing analysis tools.

![Fig. 2. Functional architecture (left), hardware architecture (right), and allocation modeling (bottom)](image)

**Design space exploration.** Another area of using metamodels and generators is then defining completely new languages for the specific system development tasks. Some of the cases we have seen companies addressing deal with defining modeling languages for early-phase design space exploration, verification and validation as well as performing failure analysis. The metamodels are here defined to capture the needed characteristics of the system and generators produce the output in the format needed by a particular tool (e.g. ABSOLUT, TLA, SPIN, UPPAAL) used to perform the simulation and/or analysis.
The results of the analysis can be provided as a feedback to the models by updating the models or by annotating the models. In Figure 3, the results of failure analysis are shown by annotating the functions in the chain: failure in one port is traced and model shows how it is influencing in the whole system. In this example the analysis is performed by the generator of the modeling tool, but the models could also be used as a front-end for an existing analysis and simulation tool (as e.g. in [15]).

![Fig. 3. Annotating failure trace directly in the model.](image)

**Automating testing.** Test cases are typically written using a common scripting language to describe different aspects of the overall behavior of the system under test. These test cases share many common properties while exhibiting some degree of variation. Such items are perfect candidates for domain-specific languages which describe a set of variants with a common set of properties.

Here metamodeling is used to specify what are tested and the created models are used for test case generation. Different testing approaches exists as models created for specification can be used as a basis for test design and test case generation, models can be created to specify individual test cases, or models can specify test logic from which multiple test cases are then been generated. Industrial experiences of model-based testing (e.g. [10]) show remarkable improvements in productivity and coverage of test cases.

4 Concluding remarks

Experienced developers have always been looking how to automate repeating development tasks. Modern metamodeling tools make this task cost effective and provide industrial strength modeling tools for the engineers developing embedded systems. Typically the emphasis has been on generating the code, either for production systems or for prototypes. Metamodeling and related tools can also be applied to support other kind of development work as been presented in this paper: support design space exploration, hardware/platform and application co-design and automate testing activities. While the above mentioned approaches may require new
languages – new metamodels to define the languages - often the most straightforward approach is to extend the generators to produce various non-software related artifacts from the existing models, or from models which expression power has been extended via metamodels. Such non-software artifacts include simulation data, hardware mappings, device configuration, wiring plans, material usage calculations, installation guidelines, etc. Having a single source, models, to produce multiple different kinds of outputs brings many benefits as the design data is entered, checked and reviewed only once. The metamodels can also be defined so that the models are kept consistent and in-completeness information can be reported directly in the models.

References

Kactus2: Extended IP-XACT metadata based embedded system design environment
Antti Kamppi¹, Joni-Matti Määttä¹, Lauri Matilainen¹, Erno Salminen¹, Timo D. Hämäläinen¹,
¹Tampere University of Technology, Department of Computer Systems, Korkeakoulunkatu 1, FI-33720 Tampere, Finland
{firstname.familyname}@tut.fi

Abstract. IP-XACT/IEEE1685 is a standard for System-on-Chip (SoC) integration. Its goals are vendor, tool and language neutral description of IP-blocks and designs in XML metadata. The standard aims to simplify IP integration and configuration as well as increase tool interoperability. Kactus2 is the first graphical open source IP-XACT tool. In addition to the basic standard, it also supports SW, SW to HW mappings and application communication abstraction using, for example, MCAPI. Kactus2 is implemented in C++ and QT4 framework and includes tools to packetize components and integrate them. For example VHDL, Modelsim, Quartus files are generated automatically for implementation, HTML for documentation and later there will be generators for also SW build. With IP-XACT and our Kactus2 extensions, porting applications between HW and SW or between different platforms becomes a matter of hours compared to weeks with non-metadata based design methods.

1 Introduction

According to semiconductor manufacturers’ roadmap (ITRS), the SoC design productivity must increase 10x and the design re-use exceed the 90% limit by 2022 to keep up with the 15x logic size increase [1]. We can apply this trend also to embedded systems product development, especially for FPGA-based Multiprocessor System-on-Chip (MP-SoC) designs that include both HW and SW components in a highly reconfigurable and reusable manner.

Current System-on-Chip designs may include hundreds of Intellectual Property (IP) blocks, of which over four may be processor cores [2]. From the software point of view, stacks of hundreds of layers and APIs and thousands of programs and libraries exist. To estimate the SW complexity, Android OS has over 15M lines of code (LoC) and its development effort is estimated to be 4.677 person years by Sept. 2012 [3].

The overall complexity has been attacked by well-known platform-based design [4] as well as model driven engineering (MDE) [5] for raising the design abstraction. Ideally there is a high-enough description of both the required application functionality and available execution platforms. After that the implementations could be automated, and both the applications and platforms could be easily reused.
However, according to our experience, there are five main obstacles on the road to the automated, model based design:

1. There are too many paradigms and domain-specific models to choose from. Each needs long time to learn and conversion between them is too cumbersome or outright impossible. Moreover, many users fear losing control over e.g. real-time behavior when low level details are hidden, or the metamodel does not cover their needs.

2. Modeling is a separate side-task not fitting to the whole flow, for example used mainly for extra documentation in addition to “real” implementation steps. Tools with custom, often changing, and even encrypted formats further complicate creating a seamless design flow.

3. Massive legacy code base and multitude of versions and variants are not adequately accounted. Designers won’t have time and money to manually create models from the old code, but without it the model based design cannot be adopted at all.

4. One-way automation from top to bottom works only with ideal tools and limited use case. In practice, user often likes to manually modify at least some parts of auto-generated code, but this is often impossible. Propagating the changes upwards does not usually exist or work.

5. High abstraction often means graphical models, but unfortunately many visual models are difficult to create and compare. Version management is easier with textual format but is less descriptive power in structural design. Graphical editors are complicated to use and design artifacts get confused due to unclear separation of shapes for different meaning.

This paper presents our approach to solve these issues in a tool framework called Kactus2 [6]. The next section explains background, the basic ideas and motivation to utilize IEEE 1685 standard (IP-XACT) [7] as the basis of our solutions. Section 3 presents our extensions to IP-XACT and Section 4 the Kactus2, the tool implementing our approach. Status of our work and future directions are concluded in the last Section.

2 Our approach

We have previously successfully demonstrated the high-level design flow, in which the design entry was SDL and later UML2.0 diagrams with asynchronously communicating Finite State Machines model. Complete design flow was demonstrated in practice for a MP-SoC on FPGA [7]. Several similar proposals exist, representing the top-down approach in Figure 1, and often suffer the problems mentioned in Section 1.

For Kactus2, we focused first on intermediate metadata formats which allow making models from different source formats and abstraction levels [9]. We also changed the primary focus from pure auto-generation to auto-assembly as depicted in Figure 1. Currently, we first perform structural modeling using metadata which is neutral to implementation language, tool, and IP vendor. Note that transformation from high-level models to intermediate level is assumed manual. Later we can include behavioral
modeling by embedding other models like UML/MARTE and timing diagram markup language (TDML). Our approach is based on IEEE1685/IP-XACT standard, which is originally meant for hardware IP-block integration with XML-metadata descriptions.

Components, bus definitions and designs are basic independent objects that have \{Vendor, Library, Name, Version\} identifier (VLNV) and can be referenced to build up the systems. Metadata for an IP component includes, for example, the associated files, interfaces, possibly the internal structure, and assisting generator scripts.

For example, ST Ericsson has published an IP-XACT case study on a SoC IP called IZARN-IP that includes an ARM CPU and is targeted to SoC for mobile phones [10]. The objective was to speed-up reuse of this IP in future design integration that normally take several months. The first time delivery with IP-XACT took 6.5 person months that is 2.5 months more than the traditional method. However, after that the libraries can be reused and next changes and integration is carried out in a week. After the initial setup, the speedup was 24x. As a specific example, the effort to make an RTL change to bus matrix structure was reduced about 50% with IPXACT.

Xilinx has recently adopted IP-XACT in all of their IP libraries, which expands use of IP-XACT greatly from ASIC vendors to embedded systems. Unfortunately, Xilinx has also introduced over 100 vendor extensions, which is a standardized method to add new properties to IP-XACT, but also causes severe vendor lock-in.

Our approach uses basic IP-XACT for HW integration for documenting the product structure, e.g. circuit boards. Moreover, we consider the SW and system domains as well. Figure 2 depicts our new objects with the VLNV identity and how they extend the original IP-XACT in three aspects [11]. The most important is inclusion of SW in the “implementation” axis that follows the platform layer model depicted in Figure 3.

An SW component encapsulates files, language and tool directives and settings like HW components. SW design is collections of SW components and can be hierarchical just like the HW designs. SW component can have API definitions which are similar to bus and bus definitions in HW. They used in three ways: a) validate API between SW components b) validate source code against API definition and c) assist code writing in Kactus2 editors.
There are also COM interfaces which describe which application components communicate. We can use, for example, Multicore Association MCAPI communication abstraction where each component has MCAPI endpoints that are connected via MCAPI channels. Our specialty is that MCAPI can be included in both SW and HW components. Visually, connecting API and COM interfaces resembles the basic IP-XACT HW design. In summary, a SW component may have ports for both API and COM, but their use is different (function calls vs. inter-process communication).

Kactus2 System design is a hierarchical IP-XACT view of the system’s HW/SW components. SW architecture includes application, application-independent SW platform, and optionally the MCAPI endpoints and channels. This is stored to the topmost HW design as a new system view. If the HW design is hierarchical, Kactus2 automatically seeks all and shows all execution units, such as processors cores, as potential placeholders for mapping the SW components.

**Figure 2** Kactus2 extensions to IP-XACT.

**Figure 3** Kactus2 “implementation” extensions follow the layer model.
3 Kactus2 tool

Kactus2 combines component packetizing editors, designers to integrate the components, and generators to create documentation and codes for simulation and implementation. Kactus2 attempts to solve the earlier mentioned model-based design problems as follows.

**Problems 1 & 2.** Kactus2 uses IP-XACT and our extensions for the abstraction that focuses on structural modeling. We heavily use hierarchy and encapsulation of information to simple looking graphical models. We allow also incomplete designs by including draft components together with fully packetized ones. This way Kactus2 can also be used for sketching, but there is still a direct path to implementation.

**Problem 3.** Creating IP-XACT component descriptions for legacy code must be simple. Such tools are freely available and Kactus2 can include them as plugin. Note that metadata-based design can be adopted gradually by first using it as only documenting the product structure using quickly incomplete components. The more content is added, the more benefits are gained.

**Problem 4.** Kactus2 can create application code from MCAPI endpoint and channel definitions and user adds his code to the stub. If either generated and manually created part changes later, the potential inconsistency problems are highlighted. This way the graphical and textual models remain in synchrony. Nevertheless, this problem requires more work to be solved.

**Problem 5.** Kactus2 uses guided layout for the graphical elements. The column-wise layout with expansion downwards helps visualizing the differences in two designs. This feature has been only partially implemented so far.

The roadmap for Kactus2 is depicted in Figure 4 to summarize the solutions to the problems. Significant leap was taken when the extensions in this paper were implemented, thus we moved directly from version 1.4 to 2.0.

**Figure 4** Implemented (green) and planned Kactus2 milestones.
Kactus2 targets also the best user experience among ESL tools. We address it by three aspects. 1) It is human friendly: it uses context to help user to never get lost and on the other hand hides all possible design paths and shows only information relevant to the stage at hand. 2) It is wrist friendly: it minimizes mouse clicks and deep menus combined. 3) It is “brain friendly”: it hides as much as possible awkward IP-XACT terms and XML tree structure from users and assists in dialogs and code editors to fill in the right information.

Kactus2 is implemented in C++ and QT4 framework. Special attention has been paid on very easy installation, since many other tools require separate compilation, environment setup or virtualization to run on user’s computer.

4 Conclusions

In a previous study we showed how Kactus2 speeded-up porting of a video encoder application between HW and SW, between PC and embedded system and between different FPGA vendor chips [11]. Such tasks are matter of hours compared to weeks with non-metadata based design methods. Our future work includes management of vendor extensions to improve import and export from other tools.

References

An IP-XACT-to-SystemC Model Generator for Mutation Analysis

Tao Xie and Wolfgang Mueller

University of Paderborn /C-LAB, Fuerstenallee 11, 33102 Paderborn, Germany
{tao, wolfgang}@c-lab.de

1 Background

In the context of IP-based SoC development, IP packaging is understood as the assembling of pre-verified IP components into an integrated design for verification or synthesis. This integration includes mainly the instantiation, interconnection, and the corresponding configurations of components. In this context, IP-XACT is the IEEE Std. 1685 language/meta-model [1] for describing IP reuse data and SoC system assembly instances. It defines the meta-model as a set of XML schema.

During the last years, mutation analysis has been considered as an adequate quality metrics to measure the simulation-based verification and validation processes [2]. As such mutation analysis is language based. It models typical, potential design errors that are called mutation operators based on the syntax of the language under consideration. This derives the basic effectiveness of mutation analysis as those errors should be detected by any capable testbench.

We have defined an IP-XACT mutation analysis framework to systematically manage and enhance the verification quality of SoC designs.

1.1 IP-XACT standard

The purpose of IP-XACT [1] is to establish a unified format for IP exchange and reuse among different IP providers, system integrators and tool vendors. At the core, it defines an XML schema as the data model for describing the meta-data of IP components and their integrations as systems. Around this standard data model, a vendor-neutral IP-reuse environment can be built as shown in Figure 1.

BusDefinition and abstractionDefinition are used to specify the port constraints and other properties for a bus interface. In abstractionDefinition, ports can be described as either transactional for TLM (Transaction level Modeling) or wire for RTL (Register Transfer Level). This makes IP-XACT to cover both TLM and RTL components. With schema definition component, we package an IP core. It contains the reference to the real design files of the IP and declaration of the physical ports as in the IP. It describes the bus interfaces implemented by the IP along which the address space and register files on this interface are also defined. Here we have further the description how the bus specification ports are mapped to the IP physical ports. Moreover, an IP can also be configurable by declaring parameters, which may have default values and later resolved during system integration.
An IP-XACT design describes a SoC system, or sub-system integration. Mainly, we have the instantiation of components and their interconnections. In the instances, we configure appropriate values to component parameters. Interconnections between components can be established through their bus interfaces or in an ad-hoc manner, i.e., direct port-to-port connection.

Additionally, the schema provides the opportunity to encapsulate IP configuration and other third-party tools as Generators, which can further be weaved to form Generator Chains. A Tight Generator Interface (TGI) defines both the API for invoking the tools from a main IP-XACT design environment and a set of Web-service based interfaces that this environment should provide to the tools for accessing and manipulating its IP-XACT files.

2 IP-XACT-to-SystemC Model Generator for IP-XACT Based Mutation Analysis

We face two challenges for this goal. Since IP-XACT designs are in the form of XML files and XML is not naturally executable, to simulate an IP-XACT SoC design we need at first an execution engine for IP-XACT. It enables us to verify an IP-XACT design and debug its errors, which further gives the possibility of adding mutation analysis in a simulation-based IP-XACT verification flow. Our way of creating this IP-XACT simulation engine is to define a set of synthesis rules that map any IP-XACT design to another system model that is simulatable. The target language that we choose for this transformation is SystemC. The mapping rules imply a SystemC model generator from IP-XACT, which we also implemented as an Eclipse tool to obtain an experiment basis for IP-XACT mutation analysis.

![IP-XACT standard design environment.](figure1.png)
Second, a key to the creation of any new mutation analysis metric is the definition of a mutation operators set on the target language. Here, these are the errors that we can implant into IP-XACT XML designs. Typical syntactic changes should be collected and classified based on IP-XACT XML schema to mimic representative and important errors that one can make with IP-XACT design.

Figure 2 shows an overview of our efforts to meet these two challenges towards an IP-XACT based SoC verification quality. Once both are available, the SystemC simulation and a set of mutation operators for IP-XACT, we can seek the bugs of a SoC design with functional tests and measure the adequacy of these tests by their ability to kill system mutants. An example of mutation operators is replacement of an existing parameter configuration. Then, each measurement of mutants is to see whether it simulates differently from the original SoC design. For a short introduction and the definition of more mutation operators we refer to [3].

Figure 4 shows the working flow of the IP-XACT-to-SystemC code generator, assuming an IP database exists in the design environment. As introduced, an IP-XACT design contains simply component instances and the connections between them. The code generation engine takes such an IP-XACT design XML file as input and, based on an IP component repository, executes the following subtasks:

- It checks the validity of the involved IP-XACT descriptions, which includes the design input and the referenced component and busDefinition/abstraction Definition descriptions from the repository. Any missed reference will terminate the process.

- In addition to the schema definition, IP-XACT standard also defines a set of semantic consistency rules that an IP-XACT document should obey, for example, whether two transactional ports are compatible to be connected together. These are also checked for the design.
After the checking and gathering of necessary information, a SystemC design file is generated. In this file, we create instances of the design components with their parameter configurations. Component interconnections are established by port-interface/port-port bindings for TLM or port-wire connections for RTL.

Further, the code generator composes also a Makefile. The purpose of this Makefile is to enable a seamless launch of simulation for the IP-XACT design. Information like source files and library dependencies is collected from the IP-XACT descriptions for components. As the IP-XACT schema specifies common types of design files that can be used, such as \textit{vhdlSource}, \textit{verilogSource}, \textit{systemCsource}, \textit{swObject}, etc. this eases the task of Makefile generation.

At the end, the Makefile is ready for a direct make simulation, which will compile, if necessary, all the referred IPs and the SystemC top design and start a simulation immediately. The benefit with SystemC simulation by a simulation tool like QuestaSim from Mentor Graphics, for instance, is that it is convenient to get a co-simulation of SystemC and other HDLs and to cover both TLM and RTL IP components.

A big challenge for the code generator lies on the creation of appropriate SystemC connections between components. At RTL, as ports have a narrow range of type choices, their binding is quite straightforward. In contrast, TLM ports are bound via the SystemC interfaces that they implement or expect for communication, which are abundant through user extensions. We need to devise a mechanism for determining whether and how two IP-XACT TLM ports can be connected. Figure 5 illustrates our proposal for automatic checking of IP-XACT TLM compatibility.
First, we note that SystemC interface classes for TLM communication can be considered as a non-private inheritance tree starting from \texttt{sc\_interface} as Fig. 4 shows it on the left hand side. Then, to enable an automated determination of TLM port binding solely based on IP-XACT descriptions, we specify the following two guidelines when using IP-XACT to describe TLM components:

- For a TLM port that implements a communication interface, such as \texttt{TLM\_port\_2} in Figure 4, its IP-XACT description should declare all the interface classes on the inheritance paths down to this interface. Therefore, the IP-XACT for \texttt{TLM\_port\_2} includes \texttt{sc\_IF\_1}, \texttt{sc\_IF\_2}, \texttt{sc\_IF\_3}, \texttt{sc\_IF\_4}, and \texttt{sc\_IF\_6}, as it is indeed capable of providing all these communication services.

- For a TLM port that awaits a communication interface, for example \texttt{TLM\_port\_1}, we should declare in IP-XACT just the TLM interface that it expects for binding; here the \texttt{sc\_IF\_2}.

Based on this rule, the compatibility between a pair of IP-XACT TLM ports should be decided by seeing whether the interface description in the requires-port is included in those from the provides-port. If the provided interfaces indeed include the required interface, a corresponding SystemC TLM binding can be safely generated between the two ports.

With such a fully automated flow, we obtain an engine for simulating and testing IP-XACT designs and their foreseeable mutants, which prepares us for defining the IP-XACT based mutation.

We implemented a prototype tool that can be used for case studies. We applied the \textit{Eclipse Modeling Framework} (EMF) \cite{4} to implement an IP-XACT editor, which then incorporates both, the code generation engine and the mutant generator. EMF facilitates the building of tools and code generators with an automated process of transforming a structured meta-model to Java classes. XML schema is among the multiple meta-model formats that EMF supports. We first fed the IEEE standard version of IP-XACT schema into the EMF process and obtained a basic editor for IP-XACT XMLs. Using the Java classes generated accurately from IP-XACT, the SystemC code generator was constructed, which is able to set up an IP repository when the editor starts and transform an IP design file as described previously.
As the Java classes are mapped from IP-XACT schema, the mutation operators can be realized by implementing correspondingly the manipulation schemes on the Java classes. At runtime, a mutation, i.e., error injection is conducted by modifying the Java objects that represent an IP-XACT XML file and saving the modified objects as another XML file to be the mutant.

3 Conclusion

This article presents an IP-XACT-to-SystemC model generator and, in particular, outlines how it serves the simulation engine of an IP-XACT based mutation analysis framework. By using a SystemC code generator as the simulation engine for IP-XACT, we are able to cover the simulation of both, RTL and TLM component integrations or combinations of them. The generated Makefile makes the mutation testing process even more efficient. With IP-XACT, our definition of mutation operators maintains a focus on the integration, configuration and interaction of IP components. For the tool implementation, the use of EMF guarantees a total conformance to the IP-XACT standard schema.

References

Combining Various Meta-Models for the Generation of Control Structures in Hardware

Rainer Findenig\textsuperscript{1,2}, Thomas Steininger\textsuperscript{3}, and Thomas Leitner\textsuperscript{2}

\textsuperscript{1} Upper Austrian University o.A.S., Hagenberg, Austria
\texttt{rainer.findenig@fh-hagenberg.at}
\textsuperscript{2} Danube Mobile Communications Engineering, Linz, Austria
\textsuperscript{3} Intel Mobile Communications, Neubiberg, Germany

Abstract. We present a hardware modeling framework that combines information contained in different meta-models for the automated code generation of control structures in hardware. Integrating a component, an interface, and a data type metamodel, our approach essentially subsumes other metamodels like IP-XACT (for which we provide importers), integrates it with behavior specification using State Charts, and additionally includes information required for code generation and consistency checking. Results from the code generation show a significant increase in productivity and as well as a promising quality of results.

1 Introduction

After model-driven and metamodeling approaches have been in use in software engineering for a long time, their use, especially in combination with automatic code generation, is becoming increasingly important in hardware development, too. In this work, we present a single-source modeling and code generation framework for control structures in embedded systems. The approach supports mixing of different abstraction level in a single model: The user may describe a model on several different degrees of accuracy, ranging from transaction-level (TL) to cycle-accurate (CA) versions of a model, both for behavior and interface (IF), and select the desired abstraction level during the code generation step. It is noteworthy that, in contrast to automated high-level synthesis (HLS) solutions, our approach employs a manual refinement flow where a high-level model is stepwise refined to a low-level model by the designer, entailing additional effort but more fine-grained control for better results than achievable with HLS.

The most interesting supported backends for the code generation are SystemC for both TL and CA implementations for simulation, VHDL RTL for synthesis, and, not covered in this paper, C header files for accessing registers from the firmware as well as documentation for, e. g., register maps.

2 Related Work

In recent years, metamodeling has gained more and more use in hardware design. IP-XACT, for example, provides a standardized metamodel (MM) covering
component, register and IF definitions [1]. Our approach extends IP-XACT with additional design aspects required for code generation. Most notably, we integrate the component definition with a MM covering UML State Charts (SCs) [2].

To seamlessly integrate the IP-XACT and the UML SC MMs from a user’s point of view, our approach extends both. For example, the SC MM used in our approach includes a refinement mechanism that allows defining states and transitions on different abstraction levels. For interoperability with the IF MM, it includes support to unify this refinement approach with the abstraction levels of IFs provided in IP-XACT. Similarly, the presented component MM supports defining an internal view of registers, allowing to define access mechanisms such as read or write triggers or access restrictions on them and thereby defining how the elements can be used in the SC. Additionally, we introduce a data type MM to support target-language-agnostic data type definitions in the other MMs.

Much research has been focused on code generation from SCs. Most relevant to our work are, for example [3, 4] for SystemC, [5, 6] for hardware description languages, and [7] for both. While those approaches support varying subsets of SCs, none of those integrates with other MMs. For the behavior generation, the presented approach builds on the one of [7].

3 Proposed Approach

In the presented framework, different aspects of the model are stored in separate MMs, with their information interlinked between the MMs. Currently, our approach uses separate MMs for component, IF, data type, and behavior definition. While this separation necessitates the recombination of the MMs so that a code generator has access to all its required information, it provides several advantages over the use of a single, large MM containing all information.

First, when implementing a design flow based on metamodeling, it is imperative that it be easy to use. Users proficient in the target domains should be able to use the MM with minimal training effort; limiting the MM’s scope can significantly aid this. For example, a user might model a standard IF without concerning herself with what designs eventually might make use of the IF.

Also, many domains are too specialized to use an off-the-shelf tool, resulting in the need to develop solutions in house. However, with separated MMs, an off-the-shelf tool might be usable for one design aspect without requiring it to be able to handle other aspects of the in-house solution. Depending on the tool’s complexity, this can save both training time and effort for the users and tool development effort; for example, we employ UML SCs for the behavior specification, easing the transition from other state-based specifications for the developers, while, at the same time, allowing the use of a low-cost, standard UML entry tool.

Separate MMs can also simplify the reuse of aspects of a design. For example, especially with bus-centered designs, different design units (DUs) will often rely on the same communications IF. In our approach, the IF can be modeled in the IF MM and later linked to several DUs. Similarly, different versions of one DU
may only disagree in their implementation but not their register map. Again, both DUs can simply link to the same register specification.

Finally, the approach needs to be interoperable with preexisting solutions such as third-party or legacy MMs. Where applicable, a third-party MM can directly be used in the framework. If the third-party MM is not sufficient or unforeseen specification changes are made to it, an in-house alternative can easily be defined. Restricting this MM in its scope can simplify implementing importers for the original MM. For example, as mentioned before, IP-XACT is an industry-standard format for exchanging IF definitions; our approach provides an importer for IP-XACT into our IF and component MMs.

3.1 Metamodels in Use

Our approach uses four different MMs. The first MM allows defining the IFs of a DU. To allow reusing the IF across abstraction levels, it supports several “views” for each IF, where one view models one abstraction level. Note that there is no inherent order in the IF views, i.e. the IF MM does not have a notion of a higher or a lower abstraction level; this will be defined in the behavior MM.

A view consists of a set of ports, where each port can be a pin or a transactional port (functions for communication with modules). Since the IF definition is independent of the component and the behavior MMs, it is easy to share standard IFs between DUs or to switch a DU to a different IF.

For the specification of behavior, we employ a SCs-based MM extending UML’s definition by a refinement association, which allows defining different abstraction levels of a SC in a single model: states and transitions can be defined on a high level and, later in the design process, the user can define refinements that implement the same behavior on a lower abstraction level. Additionally, user feedback led us to include support for variables in the SC definition. As with extended state machines, the user can define variables in the state machine. Our approach, however, expands on this idea by allowing to scope variables to a given state, akin to the scoping mechanism of most programming languages.

The information in the other MMs is connected in the component MM, which allows defining a component and linking its IFs and SC. Additionally, it provides means to store information regarding configuration and status registers that are external to the DU but logically belong to it (so-called “bitfields”) and can be used by its environment to communicate with it. Moreover, a user can define minimum and maximum allowable values and access restrictions for bitfields. Note that this MM focuses on registers that are visible to the DU’s environment, i.e., which will be added to the DU’s IF during the code generation. Registers used only internally, such as state or variable storage, are not part of this MM but, as mentioned before, implicitly defined in the behavior MM. This enables the definition of the register MM’s contents independently of the behavior.

Finally, it is noteworthy that the above MMs do not include elements specific to a given target domain but allow a unified modeling style over all supported target domains. To that end, our approach includes a data type MM for defining
language-agnostic data types for use in the other MMs, substantially simplifying the process of adding a new target domain.

### 3.2 Code Generation

Fig. 1 presents the structure of a DU generated with our approach. The “interface” block implements ports and transactions as well as bitfield access signals while the “behavior” block implements the SC itself.

![Fig. 1. Structure of a generated design unit.](image)

With the MMs separated, linking their contents is a prerequisite for code generation. To gather the required information from the different MMs, the component MM is checked, which links to the IF and the behavior specification. However, it is not sufficient to simply juxtapose elements: information contained in one MM is required to decide how information contained in another will be interpreted for the code generation. This interlinking step can use explicit links provided by the MMs, for example used for defining IFs and registers for a component and IF refinements in the behavior MM, or implicit links, as for example used for interpreting bitfield accesses in our approach (see below).

**Element Access Transformation** The IF and behavior of the DU depend on constraints imposed by the environment it will be embedded in. Apart from obvious differences such as the output language, more subtle ones arise too. First, consider the implementation of the behavior. While bitfields, ports, and variables can all be accessed in the same way in the SC, the implementation of those accesses depends on the target. As presented in [7], the implementation in VHDL uses a two-process style in which the combinational logic is separated from the sequential logic. From the combinational logic, bitfields are accessed using unregistered data and enable signals so that no clock cycle is lost before the value is written. Output ports, on the other hand, are registered—while, from the developer’s point of view, both elements are accessed in the same way, the code generator needs to implement Mealy semantics for bitfields and Moore semantics for ports. To achieve sequential execution semantics, the VHDL code generator needs to map internal variables to registers (see [7]), but mirror them in VHDL variables and convert every access to use the associated variable.

In SystemC, bitfields are abstracted to C++ variables, i.e., they can be written immediately and do not require a clock edge to update. This allows using a single-process implementation where the code is only evaluated on the clock edge (or an arbitrary event for TLM), which improves the simulation performance by reducing events and context switches. Output ports are simply assigned in this
process, making them implicitly clocked. Hence, regardless of the elements accessed, the implementation uses Moore semantics. Variables are mapped to C++ variables, automatically giving the desired semantics. For accessing ports, it is considered good practice to use read/write methods in SystemC, again requiring the code generator to check the element’s type to infer its accessing method.

All of the steps above use implicit interlinking: whenever an element is accessed, its name is looked up in the behavior (for variables), the IF (for ports) and, finally, the component (for bitfields) MM, and, if found, the access is transformed accordingly. The designer does not need to explicitly reference the element from the MM it is defined in. This, on the one hand, eases the implementation, but requires the generator to include consistency checks. If two MMs describe elements with the same name, an error is raised to avoid unknowingly accessing a different element than intended.

**Access Restriction Enforcement** Regardless of the output language, the code generator will also check the accesses to bitfields and ports. While the direction of a port is usually also enforced by the target language (e.g., neither VHDL nor SystemC allows writing on input ports), this is not generally possible for bitfields, the access restrictions of which can be more complex: a DU may, for example, only be allowed to reset, but not to set a bitfield or specific min/max values may have been defined. The generator checks each bitfield (port) access in the SC against the access restrictions (direction) defined in the component (IF) MM and issues a warning or an error if an access violation occurs.

Again, this step is done using implicit, i.e., name-based, interlinking.

**Interface Generation** For generating the DU’s IF, the generator again requires information from all MMs. While the IF MM defines the views for every IF, it is sensible that the behavior MM dictates the required view of the DU’s IFs: if the behavior directly interacts with the IF, the selected view of an IF needs to match the behavior. For example, a state implementing communication with another DU using transactions can be refined by another state implementing the same communication, but pin based and cycle true. Obviously, the high-level state requires a high-level IF view, as the low-level state requires a low-level view; other combinations are not sensible. Also, the behavior may not be specified for all views described in the IF MM. Therefore, as mentioned before, our approach unifies the refinement specification: while the IF MM defines the views of an IF, the behavior MM defines which views will be used for the selected behavior.

This step uses both explicit and implicit interlinking. The behavior and the IF are explicitly linked to in the component MM, while the behavior MM itself implicitly (again, name-based) refers to the IF MM’s views for specifying their refinement structure.

As mentioned before, bitfields require different signals, depending on the target domain; obviously, the IF needs to follow the access transformations. Moreover, a DU will often access dozens of bitfields, making their inclusion in the IF a redundant and error-prone task. In our approach, therefore, for every bitfield accessed in the SC, the appropriate access signals are automatically added to the IF, again using an implicit, name-based link between the MMs.
4 Results

Results with the generation framework show significant improvements in productivity: Designers reported the turnaround time for small specification changes or bug fixes such as the relocation of a register was reduced from up to three weeks to less than an hour, largely due to the push-button approach in which a change in a MM can automatically be pushed to all relevant code by simply regenerating the code. Generated TL models have been reported to work on par with or even more efficiently than hand-written ones, which can be attributed to optimizations done by the code generator. Also, the availability of a SystemC CA model for simulation significantly eased the integration into an existing virtual prototype. Positive results were also achieved regarding the quality of the generated hardware: Due to the manual refinement of the behavior, common problems of high-level synthesis (sub-optimal area, timing, or power usage) can be avoided, resulting in resource usage very similar to hand-written implementations, again while retaining the advantages of simply regenerating a model if parts of it need to be changed.

5 Summary

We presented an approach that integrates well-known MMs such as IP-XACT or UML SCs with custom extensions and internal MMs to provide a unified code generation framework. Instead of using one MM to store all aspects of a design, we proposed using several MMs, each containing a restricted and well-defined aspect of the design. While this increases the generator development effort, it can save tool development and training efforts. The results achieved with our metamodeling approach are promising.

References

Meta-Modelling the SystemC Standard for Component-based Embedded System Design

Martin Küster¹, Alexander Viehl¹, Andreas Burger¹, Oliver Bringmann¹,², Wolfgang Rosenstiel¹,²

¹ FZI Forschungszentrum Informatik, 76131 Karlsruhe, Germany
{kuester|viehl}@fzi.de
² University of Tübingen, 72076 Tübingen, Germany
{bringman|rosenstiel}@informatik.uni-tuebingen.de

- Extended Abstract -

1 Introduction

The language SystemC [1] has proven itself to be an inevitable standard when virtual prototyping is used during the development of embedded systems. Besides methodologies as platform-based design SystemC was clearly designed in order to enable component-based design.

Existing approaches on HW/SW co-design such as [2] do not take into account full support for a component-based methodology involving separation between the type level of a component and its instance. Furthermore, support is missing for typed interfaces which is a key element of the SystemC language.

Instead of focusing on a UML-based modeling notation we chose the Eclipse Modeling Framework (EMF) as a basis for the SystemC metamodel. This is mainly due to two reasons. First, the SystemC meta model needs to be concise, and we need full understanding of the semantics in order to support code generation. Using UML with its extensive set of constructs it would be virtually impossible to support code generation of all artifacts, leading to inconsistencies between the generated code and the modeled instances. Second, EMF-based domain-specific languages (DSLs) have recently gained much attention and led to the creation of several open-source tools. This involves essential capabilities such as support for textual syntaxes, model repositories or new transformation languages. However, we outline how the proposed framework can be coupled with MDA-based system modeling in order to automatically generate SystemC virtual prototypes in [3].

The use cases involving the SystemC models are twofold: code generation, i.e. generation of fully-functional C++ code conforming to the SystemC standard is one. The other is connection to high-level UML artifacts such as components, or hardware blocks. Instead of applying stereotypes (sc_module etc.) to existing design elements it is easier to design software and hardware first, then establish the allocation of software components to hardware nodes and generate the necessary SystemC elements from the UML model in a final step. Following this
approach, we do not "spoil" the design models with SystemC constructs that make sense only after the deployment has been determined.

Further, the framework allows the interconnection with bottom-up and round-trip engineering of C, C++ and SystemC designs by using an Eclipse CDT-based frontend which is already implemented but out of scope of this extended abstract.

2 SystemC Meta-Model

![SystemC Metamodel Core](image)

**Fig. 1.** SystemC Metamodel Core. *Note: types of interfaces etc. are not displayed*

The core idea of introducing a dedicated meta model for SystemC is that in model-based or model-driven engineering, we need an abstraction for both linkage with other modeling languages such as UML and, which is the focus of this paper, that we want to generate code from the models created.

In the case of SystemC, we need to understand that all SystemC constructs are basically C++ elements (makros, calls to static libraries etc.). On the one hand, the semantics is clearly determined by the execution of the simulator code. On the other hand, staying on the code level means that designers have to cope with the wrong level of abstraction. We need to raise the level in order to link it with other design models. On the contrary, we need to include all necessary details in the model that we can generate code from the SystemC
elements included in the model. Everything that can be expressed in the SystemC language shall be able to be modeled with the newly created meta model.

Approaches such as SATURN [4] or Riccobene et. al [5] have focused on UML profiles applied to SysML constructs in order to generate verification code. The set of available constructs, is however reduced compared to the IEEE standard [6]. Little support is given to a strict component-based design methodology, where components (resp. modules) can be defined on a type level and be used as instances, and where different types of connections (port-port-, port-export-, delegation bindings, ...) can be modeled.

The scope of the meta model is restricted to the core elements of the IEEE standard, ([6], Ch. 5). This includes modules, interfaces, channels, ports, events, sensitivities, threads, methods, and data types. The predefined channels and interfaces (signals, mutexes, or fifos for instance) are included, too, but in a separate package. They are not included in the diagrams (Fig. 1) for better readability.

**Type vs. Instance Level** For the discussion of the meta model, we will not go into detail of every meta class, but highlight the key design decisions. A distinct feature of the meta model is its clear separation into the type level (Fig. 1) and the instance level (Fig. 2). This is a key element, which is very hard to reproduce unambiguously in a UML model, especially if there is a deeply nested hierarchy of modules.

In SystemC, bindings between ports can be specified as part of the constructor of the `SC_MODULE`, or in the `sc_main` method. Both versions can be modeled (the constructor version will result in a `InnerAssembly`, the other in a simple `Assembly`. ) Such a differentiation is impossible in a UML-stereotype metamodel. However, it is good practice to instantiate modules and bindings in the `sc_main`. This fact should not be neglected.

**Communication Styles** Another important aspect of the meta model is its separation of binding types. SystemC allows a number of different ways to connect (ex-)ports, (sub-)modules, processes, etc. The meta model reflects this by introducing three different binding types: `DelegationBinding` for binding of ports to ports of sub-modules, `PortBinding` for binding of two ports via a local channel, and `ExportBinding` for binding of exports to ports. This binding type is dedicated to connecting ports to a local channel hidden by the export.

**TLM extensions, Pre-defined Channels** The transaction-level modelling (TLM) is a way to abstract from implementation details of the communication between modules by focusing on the functionality of the data transfer. For that, SystemC contains a number of new standard `SC_INTERFACEs` that can be used to model such data transfer. The interfaces that are part of the TLM library are included in the meta model. Basically, all interfaces are derived from `SC_INTERFACE`. The two mostly used implementations are **simple_initiator_socket**
and simple_target_socket are derived from SC_PORT and SC_EXPORT respectively. They are excluded from the diagrams since they are straightforward and do not bring any new insight.

All pre-defined channels, such as sc_signal are included in the metamodel, but excluded from this discussion, too. To us, it is not clear why they are part of the standard instead of a library.

3 An Example SystemC Model

To illustrate the use of the meta model we want to give a very simple example (see Fig. 3). Again, only the elements that are not self-explanatory or not visible in the diagram are detailed. The examples includes two software components (ImageProc and Display) that are allocated to a hardware platform consisting of a DisplayUnit and a CPU plus a memory and bus connecting the modules. The instances are AssemblyContexts in our terms.

The example illustrates the different bindings types: the System top module delegates to one of its sub-modules via the IImageProcessorPort. The communication between the image processor and the display component is realized via PortBindings that use a simple pre-defined sc_signal. Finally, all communication in the hardware platform is modeled on the transaction level (TLM). For that, connections between the initiator and target sockets are modeled as ExportBindings.

Threads and methods are not displayed in the diagram. In the Display component, we defined two threads (write_partial and write_complete) that are responsible for displaying an image (or part of it). They are sensitive to two SC_EVENTS that are signalled if the method is called. This enables asynchronous calls and is required if the components run on two different hardware elements and need to explicitly route the call from one to the other.
The idea of the separation of the model into two separate sub-models is that software designers should, first, not care about the allocation of their components. Components, interfaces etc. are defined separate from the hardware. After both models are established, the allocation can be modeled. In a final step, the two UML models are combined in an integrated SystemC model that represents the (hardware and software) components as modules and contains a hierarchy of models that interact via the given hardware connections. This last step can be automated as a model-to-model transformation taking a UML model (software components) and a SysML model (hardware) and results in a SystemC model.

4 Code Generation

A main goal of the designed SystemC metamodel was to be able to reconstruct or create SystemC code in a model-to-text manner. This involves easy mappings such as the one from SC_Module to a SystemC module class. The more interesting cases result from binding on the instance level and processes and sensitivities.

Table 1 summarizes our findings. Due to space limitations, we highlight only those elements that are different from existing SystemC metamodelling approaches.

5 Conclusion and Future Work

This extended abstract presents a framework for component-based design of embedded hardware/software systems based on a meta-model of the SystemC 2.2 standard including TLM2.0. We outlined the approach of distinguishing between...
(Inner)Assembly

∀ contexts: i = 0...contexts.size() − 1
contexts[i] * = new contexts[i] → module ( name )
in SCCTOR for inner assembly

Export

sc_export<type> name
name(...) declaration;
instantiation in SCCTOR

ExportBinding
export( channel ) in SCCTOR

PortBinding
port[0]( channel ) via local channel

PortBinding
port[1]( channel )

DelegationBinding
delegatedPort(delegatingPort) port to sub-module

Table 1. Mapping of SystemC metamodel elements to code

static structural and binding/assembly aspects of SystemC designs. A brief modelling example was given in order to understand the separation of functional and structural hardware concerns. The overall framework which embeds the presented solution further includes SystemC code generation based on Model2Text transformations [3] as well as bottom-up integration of functional legacy components in C/C++/SystemC CP. Future work will address the capabilities of SystemC2.3 in order to directly model software-related aspects as for example thread scheduling instead of realizing it through a user-level library [7].

References

Code Generation for QEMU-SystemC Cosimulation from SysML

Da He, Fabian Mischkalla, Wolfgang Mueller

University of Paderborn/C-Lab, Fuerstenallee 11, 33102 Paderborn, Germany
{dahe, fabianm, wolfgang}@c-lab.de

Abstract. With the continuously increasing complexity of modern electronic systems, the engineers are facing more and more challenges (e.g., time-to-market) in the hardware and software co-development. In this article we present a SysML-based modeling approach for HW/SW combined systems and a code generation scheme to automatically provide the QEMU-SystemC based virtual simulation environment.

1 Introduction

Due to the increasing complexity of electronic systems Model Driven Engineering (MDE) based design techniques like Systems Modeling Language (SysML [6]) are becoming more and more interesting for combined system documentation and early testing. However, there is still a big gap from abstract UML specifications to executable models, i.e. virtual prototypes, for dynamic verification and synthesis.

![Design flow](image_url)
In [1], [2], [3], [4] and [5], we developed an efficient HW/SW codesign approach based on SysML by covering the design flow (shown in the Figure 1) from modeling to implementation via retargetable code generation. For modeling, we defined additional UML based extensions on top of SysML, which support: (i) target SW integration, (ii) convenient SystemC modeling, and (iii) SystemC based high-level synthesis.

This paper focuses on code generation for the automatic configuration of the HW/SW cosimulation infrastructure composed of QEMU and SystemC. We implemented the retargetable code generation scheme in ARTiSAN Studio® (a UML/SysML development suite for model engineering) to generate C/C++ code for target SW binaries, synthesizable SystemC for platform components like interconnects, memories, and HW accelerators as well as other utilities such as self-managed makefiles and scripts for complete design flow automation. We address QEMU in its full system mode as a CPU emulator for different processor architectures running the target SW binaries along with an operating system.

2 HW/SW Codesign and Code Generation

Our HW/SW codesign methodology focuses on system architecture modeling in terms of component-oriented view. For this, we use SysML for structural modeling. As such we are able to define individual system components, their interfaces, and interconnections. The entire system model can be specified by SysML front-end editors like ARTiSAN Studio®. Based on our custom annotations to SysML, we map SystemC execution semantic to the specification model. This 1-to-1 relationship provides efficient and automated code generation as well as round-trip capabilities by common UML tools. In our codesign methodology, a specification model consists of three different types of components: (i) software executables, (ii) processors, (iii) platform specific interconnect and hardware components.

Figure 2 gives an overview of the code generation for the automatic configuration of a QEMU-SystemC cosimulation environment. Each HW component is translated into a SystemC module. Such components can be either predefined components from external libraries or explicitly modeled by the user. In the first case, legacy SystemC code is referenced from libraries, such as for TLM Bus and TLM Transactors as shown in Figure 3. For the latter, a new SystemC module with a header and an implementation file is generated. For each SW component referencing external source the code generator generates a register address map extracted from the model, user space driver functions and makefiles to cross-compile it to a target SW binary. The register address map is generated as C macros in a header file containing the definition of base address, high address of the device as well as offsets of each internal registers. The user space driver functions provides software developers easy access to the hardware devices via generic byte/half-word/word read as well as write operations. A target processor platform based on a dedicated ISA (Instruction Set Architecture) and an OS (Operating System) image is specified by processor models that are allocated from these binaries. Finally, automatically generated shell scripts compile this bundle to an executable image including OS and application software binaries.
Figure 2: SysML based code generation for an automatic configured QEMU-SystemC cosimulation environment

A processor component communicates over so-called TLM interconnects with other system components. For this communication, a QEMU Plugin (cf. Figure 3) containing all accessible device declarations via the QEMU standard API call (cpu_register_io_memory) is automatically generated. The QEMU Plugin is implemented in form of a runtime library and is loaded into the QEMU core dynamically. As the QEMU plugin is based on the information about the modeled system architecture in terms of devices and their register address map, it may differ from design to design. In order to automate this configuration, we developed an ARTiSAN Studio® code generator to extract the information from the specification model and to generate the plugin. The automatic code generation significantly reduces the configuration effort for setting up the entire QEMU-SystemC based cosimulation environment.

3 QEMU-SystemC based Cosimulation

With code generation, the complete cosimulation environment is automatically configured and can be launched by a simple button click. Due to the support of multiple target architectures by QEMU, we can easily select them in the specification model. Currently, Linux and Android are supported for PowerPC405 as well as ARM926EJ-S in form of OS library elements.

The diagram in Figure 3 outlines the detailed infrastructure of our cosimulation framework. The QEMU emulator and SystemC simulator are running as two separate processes which are synchronized by means of inter-process communication by shared memory. On the SystemC side the TLM Transactor (on top of the TLM Bus in
Figure 3 transforms data from the shared memory to TLM transactions, so that they can be forwarded by the TLM bus and processed further by other individual SystemC modules. On the QEMU side, we use the aforementioned QEMU plugin to connect the QEMU core to the SystemC process.

![Cosimulation framework diagram](image)

Our cosimulation framework is based on an asynchronous communication paradigm, which means that the SystemC simulator and QEMU emulator run concurrently until SW applications from QEMU invokes an I/O operation (read or write operation), which exchange data between both simulators by blocking calls. After the I/O access is finished, QEMU and SystemC resume concurrent execution until the next I/O operation. During cosimulation the QEMU emulator acts as a master that can initiate I/O operations while the SystemC part only reacts as a slave. The synchronization between QEMU and SystemC must be currently ensured by the application design itself as the applied QEMU is untimed.

Figure 4 shows more details of the sequential flow of read and write I/O operations. Each I/O operation in the SW application invokes a corresponding callback function that is implemented in the QEMU plugin. The callback functions then send the data to the TLM Transactor of the SystemC process and wait for the acknowledgement/data via blocking receive calls. While the QEMU is waiting, the SystemC continues processing the transaction and returns the acknowledgement/data thereafter. In case of read operation, the data are simply retrieved via plugin and transactor from SystemC.
4 Conclusion

In this paper, we introduced a flexible MDE based HW/SW codesign approach to close the gap between comodeling, cosimulation, and implementation via automatic code generation. The code generation includes C/C++ for target SW, synthesizable SystemC for HW components and self-managed scripts for design flow automation. The code generation framework automatically configures the QEMU-SystemC cosimulation framework. The implemented tool environment was successfully validated in the SATURN project by several partners who applied it for several industrial case studies [3].

References

5. EU Project: SysML bAsed modeling, architecTUre exploRation, simulation and syN-thesis for complex embedded systems (SATURN), www.saturn-fp7.eu


Code generation for component-based applications

Ansgar Radermacher and Chokri Mraidha

CEA, LIST, Laboratory of Model driven engineering for embedded systems
Point Courrier 94, Gif-sur-Yvette, F-91191 France
{ansgar.radermacher,chokri.mraidha}@cea.fr

Abstract. This paper describes an extensible framework of code generation from executable component-based application models in which the structure, the behavior and the interactions of components is specified. The presented approach aims to provide a set of concepts that enable to create application components that are independent of the underlying OS and middleware via a flexible tool chain that is parameterizable and extensible. Another objective is to reduce the complexity of the code generation by dividing it into several steps that produce intermediate artifacts. The first steps are model-to-model transformations steps, the last is a model-to-code generation. Separating the steps towards code generation has two advantages: it enhances the flexibility as different parts of the chain might be exchanged and it enables the reuse of standard code generators that do not have specific support for component model features such as ports and connectors.

1 Introduction

Code generation from component-based models implies to take into account on the one hand their structure with their internal behavior and on the other hand their interaction with other components. Component execution could happen in heterogeneous contexts. Middlewares are employed in this case to abstract lower level execution resources and communication mechanisms provided by the underlying operating system and the hardware platform on which components are deployed. In order to avoid that code generators are specific to a communication mechanism, the code generation framework should be highly configurable and flexible. As shown in figure 1, the approach presented in this paper tackles this problem by taking as input three models describing the software components, the hardware platform, and component instances, including their configuration and allocation to hardware components. The objective of the approach is to support a set of concepts that enable the creation of application components that are independent of the underlying

![Fig. 1. Overview of the approach](image-url)
OS and middleware via a flexible tool chain that is parameterizable and extensible (the flexibility is primarily achieved by feeding model libraries as input in the transformations, not by changing the transformation chain itself).

The paper is organized as follows. Section 2 gives a description of the process input models. Section 3 shows the transformations of these input models towards an implementation model. Section 4 describes how code is generated from the system. Section 5 concludes the paper.

2 Input models

The set of input models contains three sub-models that are shortly described in the sequel: (1) the software component model, (2) the hardware platform and (3) the deployment model.

Component model We use the UML MARTE [3] component model that describes components (with their internal behavior), and the interaction points (ports) characterized by the transferred data or by the provided/required services. Interactions are realized by the design patterns described in section 3. The concurrent behavior is specified via the HLAM (High Level Application Modeling) sub-profile of MARTE. This concurrency model identifies the components that possesses execution resources (RtUnit) and the shared ones (PpUnit) which do not have execution resources, but resources that manage the concurrent access.

Platform model A hardware architecture is described in a similar way as the logical architecture by composition of elements. A class called “HWArchitecture” represents the complete platform. The attributes of this class represent nodes. Each node is typed with a class that captures the properties of this node. Each node can have a further internal structure, a hierarchical structure can thus be modeled.

Deployment model The deployment of an application consists to define the component instance, their configuration and their allocation to an execution node. The UML composite structure defines the roles that are played by each component of the system. The deployed component is always an instance of a component that has a specified allocation to an execution resource (node or thread) and a specific configuration. In UML, an instance is defined by an “InstanceSpecification”, the values of properties are given via “slots”. In case that the property represents a sub-component, the associated value is a further instance specification. The resulting hierarchical structure is quite difficult to maintain and is automatically generated and maintained by tools. Figure 2 shows the deployment of a very simple application to a platform consisting of two nodes.

![Diagram of Component deployment on a platform](image-url)

Fig. 2. Component deployment on a platform
3 Generation of an implementation model

This section describes two main design patterns that are applied to the application components during the transformation to an implementation model. The implementation model details how interactions and technical aspects, such as thread-safety are (PpUnit concept in MARTE) are realized. The proposed transformations exploit information declared in a model library which contains interaction components and container elements, as explained below. The objective behind this two transformations is a separation of concerns: the functionality of a component must be specified independently of the technical constraints of the the environment including the available communication mechanisms.

3.1 Components, connectors and containers

A component is an entity represented by a UML class. A component owns interaction points called port. A component needs to declare explicitly which services are required and which services are provided. In order to maximize reuse, a component may not know which components offer a certain service. A component has an internal structure in form of parts (typed by other components) that are wired among either other or with ports of the enclosing component. A component may be a “type”, i.e. an (abstract) class within a set port ports and no internal structure or an “implementation” which realizes the services that are offered by the ports. One or may implementations might realize (implement) the same component type.

A connector is a UML element connecting two parts or ports within a composite. In UML, a connector denotes an interaction between two elements without detailing how this interaction is realized. In our case, this information is declared by additional information added to the connector (synchronous/asynchronous communication). After the application of the container design pattern that is described in the following section, an interaction component is generated in the implementation component.

A container is an entity that encapsulates a component and realizing non-functional aspects. This enables a separation between functional and non-functional: the implementation of a component can be focussed on the business logic.

3.2 Connector and container pattern

The connector pattern is based on the idea of Garlan and Shaw [6] that a realization of an interaction must be a first class citizen of the application model. It must be possible that an interaction – like an application component – has multiple realizations and can be configured. The idea of connectors has been formalized in the context of UML in [4] and [5]: a UML connector – a “line” within a composite class – is replaced by an interaction component as shown in the left part of Fig. 3.

The interaction component is typically defined in a model library in form of a template. This is required, since the interaction component must be able to adapt to the context in which it is used. For instance, in case of an operation call of a component, the interaction component must provide the same operation as the component. In most cases, the formal parameter of the template is either an interface or a data-type. The implementation of the interaction component must be adapted as well, in order to correspond to the interface. The implementation body is thus provided in a form a a template of a typical model-to-text (M2T) language, in our case Acceleo.

The container pattern may be used to modify the way a component interacts with its environment. This pattern has been identified by existing middleware solutions, such as CCM[2] and Fractal[1]. The container encapsulates an object (component) and can provide additional service and observer or manipulate the interactions
with the component. In order to enhance flexibility (and readability) of the application model, it is preferable to add information about the container (in form of rules) to the model, but not the container itself. Thus, a transformation step adds the container to the model.

![Diagram](image)

**Fig. 3.** Connector reification and container expansion

The right side of Figure 3 shows this principle: a component “C” is enriched with rules to apply. This information is evaluated by a transformation that creates the container and adds the elements that are associated with the rule. The component becomes an *executor*, i.e. the business code behind a component. It is possible to distinguish two different types of elements within the container: the *interceptor* and the *extensions*. The interceptor is placed on a delegation connection between a port of the container and the executor. The extension is an additional element which can be connected with external ports of the container.

![Diagram](image)

**Fig. 4.** Container rule for a state machine

The current container libraries offer for instance the production of execution traces or the realization of mutual access. An interesting feature is the support for state-machines. Its realization requires three elements within the container: the state-machine itself, an event pool and an interceptor which feeds the pool (e.g. each time an operation is called, an associated call event is produced). These elements are captured by the rule shown in Figure 4.

In a similar way as the FIFO interaction component, the state machine and the interceptor are defined in a package template. The model of the state machine is
defined for a component (class). The implementation of the state machines depends on the formal parameter which is instantiated with a class. The message interceptor is typed with an interface, since it primarily captures call-events.

The following code fragment describes the template of an operation which manages the execution of a state-machine. The code, here using the C++ language, is modeled by an UML opaque behavior. The access to the model element is embedded between \([\) and \(\])\), i.e. \([\text{name/}]\) allows to access the name of a UML element. This excerpt shows the power to adapt the behavior of embedded components via the container to add new functionality that is well separated of the business code.

\[
\begin{array}{l}
\text{for (sm : StateMachine | ownedBehavior->select(oclIsKindOf(StateMachine)))}\\
\text{switch(m_currentState)}
\end{array}
\]

\[
\begin{array}{l}
\text{for (state : State | sm.region.subvertex->select(oclIsKindOf(State)))}\\
\text{case [clazz.name/][state.name/]:}
\end{array}
\]

The instantiation of the template with a fictive class “A” is shown in the next code fragment. This class contains a state machine with the state “state1”, which appears in the code’s “switch” statement. In each state, the events defined as triggers for transitions are compared with the event read from the pool (in which a timer or interceptor write).

\[
\begin{array}{l}
\text{switch(m_currentState)}
\end{array}
\]

\[
\begin{array}{l}
\text{case A_state1:}
\end{array}
\]

### 4 Code generation

The result of the previous phase is a component model of the application, enriched with reified connectors and expanded containers. The code generation starting from this model requires two actions: (1) the realization of the component deployment consisting of a splitting the global model into sub models for each execution node and (2) the transformation of ports and connectors that do not have a direct equivalent concepts in object-oriented programming languages.

The former is not as trivial as it may seem since dependencies have to be taken into account and composites may have to be deployed on multiple nodes due to allocations of theirs parts. This imposes constraints such as only read-only attributes in these composites to ensure consistency. We do not discuss this issue further in the context of this paper and focus on the second aspect, the transformation of ports and connectors.

Ports and connectors do not possess a direct equivalent in an object-oriented programming language. Thus, it is necessary to relate component-oriented concepts to object-oriented concepts, i.e. classes, interfaces, attributs and operations. We distinguish ports with provided interfaces and ports with required interfaces (a port might also have both). A port providing an interface is an access point to a service and a caller needs to obtain a reference to this service, in our implementation pattern via a specific operation. For instance, if a component owns a port “p” providing interface “I”, the realization of a component needs an operation “get_p” returning a reference to the service. The implementation of this operation is determined automatically: in case of a delegation connector between the port and an internal part of a component, this reference is returned, otherwise a reference to the component reference itself is returned.
A port with a required interface is an interaction point which requires a reference of another component that provides the interface. Thus, the component needs to store this reference and provide an operation to initialize the reference in the moment of instantiation. For instance, a port “q” with an required interface is transformed into an attribute which stores a reference to a port providing the interface and an operation “connect,q” which initializes this reference.

Connectors within a composite are transformed into a realization of a specific operation that creates the connections between parts, i.e. contains suitable combinations of some-part.connect some-port other-part.get other-port.

Once the transformation of component-based models to object-oriented models is done, a “classical” code generator taking an object-oriented UML model as input is sufficient for the code generation (in our case C++). For each class or interface, a C++ class is generated. The UML packages are transformed into C++ name spaces. The organization of the files follows the same as in Java. A name space corresponds to a directory and thus reflects the package structure in UML.

The dependencies to the external packages are translated into include directives to libraries. The generated code can now be compiled, for instance in the context of the Eclipse CDT environment.

5 Conclusions

We presented a code generation chain for component-based application. The use of intermediate steps towards code generation has the following advantages: (1) it enhances the flexibility as different parts of the chain might be exchanged – e.g. if another programming language is targeted, (2) it facilitates the maintenance of the generation step, since intermediate steps can be validated, and (3) it facilitates the use of existing components such as the standard model to code generators.

In the presented approach, the container and connector transformations are flexible: whereas the transformation itself is fixed, the added elements are instantiated from extensible libraries of interaction components and container elements. Currently, the component to object-oriented model transformation uses a fixed pattern how ports and connections are mapped, we will make this pattern adaptable to achieve compatibility with different middlewares, such as for instance CCM or the robotic middleware Orocos.

References

UML/MARTE methodology for high-level system estimation and optimal synthesis

Héctor Posadas, Pablo Peñil, Alejandro Nicolás, Eugenio Villar

Microelectronics Engineering Group, Universidad de Cantabria, 39005-Santander, SPAIN
{posadash, pablop, nicolasa, villar}@teisa.unican.es

Abstract. Design of embedded systems is facing the challenge of their growing complexity and strict performance requirements. Model-driven design solutions are very common in this context, where the UML/MARTE profile is a well-known solution for real-time, embedded system modeling. During the design process, several specification alternatives can be considered; specifically, the HW/SW platform, concurrent application structure, application allocation into HW/SW platform resources, etc. The exploration of these design alternatives enables a set of performance estimations to be obtained in order to choose the optimal specification, facilitating system implementation and minimizing designer effort. The paper proposes an UML/MARTE methodology the enables automatic estimation of the system to be implemented. Once the optimal system specification has been defined, the proposed UML/MARTE methodology enables the final system to be implemented through an automatic synthesis process.

1 Introduction

The growing SW complexity of current embedded systems has led designers to increase the abstraction level of the first stages of the design process. Designing at higher levels of abstraction provides an effective way to deal with the complexity of large systems. The effects of creating different execution flows and deciding on different HW resource allocations have to be evaluated early in the design process. Thus, system-modeling methodologies and implementation flows have to be flexible enough to enable optimizing performance by evaluating multiple design decisions with minimal designer effort.

In this context, model-driven design methodologies based on UML are commonly adopted to handle the early design of embedded systems [1,2]. The models enable easy and fast description of the entire system, which is then used as input for the steps of code generation and integration [3]. However, pure UML usually lacks the semantics required to adequately model all the characteristics of embedded systems. As a consequence, these models are commonly developed following different profiles

1 This work has been founded by the PHARAON FP7-288307 and the Spanish TEC2011-28666-C04-02 MCI projects.
which add additional semantics to the original basic UML components. Among these profiles, MARTE is gaining increasing interest for the development of real-time, embedded systems.

Taking MARTE-based models as input, several synthesis approaches have been proposed. Gaspard2 [4] is a design environment for data-intensive applications which enables MARTE description of the application and the hardware platform, generating an executable TLM SystemC platform at the timed programmers view (PVT) level. In [5] a design flow based on high-level languages (SysML, MARTE, SystemC, etc) enables the generation of the deterministic multi-threaded code for parallel implementations. In [6], a component-based modeling methodology based on UML/MARTE and explicitly designed for supporting DSE is presented. In [7] a semi-automatic solution for generation of HW/SW infrastructure from UML models is presented.

However, all these solutions are oriented to generating completely fixed models, especially in their concurrent structure, which limits their applicability when the system must fulfill different constraints. In that context, design space exploration of concurrency and allocation must be considered in order to find the solutions accomplishing all the requirements. In this way, the Zeligsoft infrastructure [3] supports the generation of codes for different resource allocations, but without providing simulation services that enable system constraints to be considered early in the design process. Other solutions [8,9] enable different system-level simulations, but with limited exploration capabilities. However they do not provide enough capability to explore the optimal system’s concurrent architecture.

To solve this problem, this paper presents a methodology to help designers to explore and automatically implement different design system concurrency architecture alternatives in the POSIX domain. The approach enables the optimization of the concurrent structure in the UML/MARTE model by easily modifying the communication semantics and interfaces used as communication mechanisms. Using this infrastructure, the models are automatically implemented, generating all the files required to simulate the architectures in a fast native co-simulation tool. Then, the designer can easily modify the model based on the performance evaluations resulting from the modifications proposed.

![Fig. 1. Original sequential architecture of the MPEG4 application used as an example and later concurrent architectures evaluated modifying the communication semantics.](image)

MeCoES 2012
These model modifications basically focus on concurrent SW architecture and re-source allocation. It is possible to modify the communication mechanisms among application components to force them to be sequential, concurrent, to create multiples copies of a component, etc. (Figure 1). Then, the communications are automatically implemented and the result evaluated using the synthesis tool developed. Thus, at the end of the exploration process, optimized synthesized codes are automatically generated for direct integration in the physical platform.

2 UML/MARTE design flow

The exploration process that can be performed by the proposed infrastructure is based on a five-step flow (Figure 2). First, the system is modeled following the proposed UML/MARTE modeling methodology, providing the C/C++ files containing the components functionality. Then, the model is automatically transformed into an executable code and, the fast, native simulation tool SCoPE [10] is called, in order to obtain performed metrics characterizing the UML model. Next, the performance metrics are analyzed by the user, obtaining conclusions about which changes in the model channels can optimize the system.

When the system obtained fulfills the requirements, a synthesis tool is called. The synthesis tool analyzes the information of the UML model and generates all the elements required to create the executable SW to be mapped to the physical platform, including communication wrappers, main files, etc.

To support this flow, the UML/MARTE methodology proposed enables these changes to be performed by focusing on the deep modeling of communication channels as a way of enabling the modification and exploration of the application’s concurrent architecture. This exploration will enable an optimal use of the HW platform resources, taking advantage of potential parallelism.

2.1 UML/MARTE Meta-modeling

The UML/MARTE system modeling methodology defined to enable the exploration and synthesis flow is a component-oriented one, following the Model Driven Architecture (MDA) principles in the development of HW/SW embedded systems. The application is divided into functional components that are connected through communication media, and mapped to the processing elements of the HW platform.
The SW components provide and require functions that are grouped in interfaces, using the MARTE modeling facilities. Additionally, new communication semantics have been added as a way to connect the provided and required interfaces of the functional components, considering different behaviors. These semantics are captured in channel models, providing a powerful, flexible and easy-to-use way to define and explore the system’s concurrent architecture. Specific information about the new channel semantics can be found in [11].

At the same time, the modeling methodology is based on the idea of the separation of concerns. This separation is achieved by providing distinct system views to the designer; each one for a relevant aspect: data model, concurrency structure, communication mechanism, HW platform, SW functionality, etc.

3 Generation process

Once the model is properly created, the generation processes required to perform the exploration flow and the final synthesis can start. To do so, all the different codes required to perform the simulation and the final synthesis must be generated from the information in the UML model, as described in figure 3. Since the native co-simulation tool used in the flow (SCoPE) and the target domain supported (Linux) are both based on the POSIX API, the generated C files used to integrate all the components are valid for both steps of the flow, simplifying the process.

![Fig. 3. Flow implemented to generate the files required for simulation and final implementation in the physical platform.](image)

First, some generators developed in Acceleo and integrated in Eclipse analyze the UML/MARTE model, translating all the information into XML files and generating the Makefiles required during the compilation process. These files are:

- `DataModelFile.xml`, containing the specification of data types used in the model.
- `ApplicationFunctionalityAndApplicationStructure.xml`, describing external ports and internal characteristics of the components, such as their associated C/C++ files or the number of internal threads.
- `Communication.xml`, describing the communication mechanisms that interconnect the system components and their associated semantic characteristics.
- `InterfaceFile.xml`, describing the interfaces used by the application components.
- `MemoryAllocation.xml`, specifying the application component mapping to memory partitions.
- `HWPlatform.xml` and `SWPlatform`, describing the HW components (processor, memories, buses, etc.) and the SW platform components (OS, drivers, etc.).
- `Mapping.xml` describing the allocation of the memory partitions to HW resources.
Starting from these intermediate files, a code generator tool is called. This tool automatically generates a set of several files that are required to create the executable files and the configuration files for the SCoPE generator. The creation of the executable files includes the following steps.

The first step consists in the generation of the C files implementing the semantics of the channels. Generation of threads, service calls, data splitting and synchronization mechanisms are implemented in order to generate the concurrent architecture defined in the UML model. A file is generated for each application component defined in the model, implementing the management required to provide the communication behavior defined for each function of the component interfaces, depending on its role (provided or required).

The goals of the second step are the generation of wrappers for the interfaces related to each component. Transfer mechanisms are implemented in a generic communication library providing different implementations for inter-process communication, intra-process communication and communications between different operating systems (TCP/IP). Thus, adaptation wrappers are required to connect the generic functions of the library to each function of the component interfaces, considering the function name and the type, size and direction of the arguments and returned value. All this information is encapsulated in generic buffers that are transferred by the communication library and recovered in the target component.

When memory space definitions indicate the type of transfer required for each communication, main files are generated, one for each memory space. This enables the support of multi-OS systems, and simplifies the integration of third party codes.

Finally, all the generated C files are compiled together with the application files provided by the user using the Makefiles generated from the UML/MARTE model. Then, the compiled code can be executed in the SCoPE simulator together with the XML files required to configure the virtual platform. This compiled code can also be used for integration in the physical platform.

5 Application Example

The UML/MARTE approach proposed has been applied to a MPEG-4 encoder application, trying to decide on an optimal implementation on an OMAP4 HW platform. The MPEG-4 encoder is an industry-standard, consisting of a motion-estimation and compensation phase followed by transformation and entropy coding phases. The UML model created contains a set of functional blocks: MEMC (MotionEstimation-MotionCompensation), TCTU (TextureCoding-TextureUpdate), EntropyCoding (EC) and BitstreamPacketizing (BP).

The MPEG-4 encoder implementation used in this paper enables different system configurations to be established by modifying the channel semantics of the model, following figure 1. From an initial sequential implementation, channel semantics enable the definition of different parallel regions. Additionally, several copies of a parallel region can be called in parallel, operating with split data. Automatically generated code controls the concurrency, data management and synchronization required to interconnect all the components.
During the exploration, the 6 configurations shown in figure 1 have been evaluated with the simulation tool, as shown in table 1. As a result, one of the best configurations found (“concurrent 3”) has been automatically integrated into the final platform without additional effort.

Table 1. Performance estimations obtained by the simulator during the exploration phase.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Sequential</th>
<th>Concurrent 1</th>
<th>Concurrent 2</th>
<th>Concurrent 3</th>
<th>Concurrent 4</th>
<th>Concurrent 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated time</td>
<td>12.54 s</td>
<td>8.25 s</td>
<td>7.26 s</td>
<td>6.82 s</td>
<td>6.86 s</td>
<td>6.82 s</td>
</tr>
</tbody>
</table>

6 Conclusions

The paper presents an approach for easy exploration of concurrency architectures in embedded designs. The approach takes advantage of an extension to UML/MARTE to model the different architectures, and automatically synthesize the SW communications from the UML/MARTE models. The automatic synthesis process and the intermediate generation of XML files enable easy exploration of different allocations of SW components using the SCoPE tool. From the UML model, a generator synthesizes the communication wrappers and the main C files in a completely ad-hoc way for the application, reducing the overhead obtained with more generic design alternatives.

The approach enables easy and early exploration of the system concurrency architecture, simplifying the consideration of constraints in the design process, and obtaining the final executable file for the physical platform from a single, integrated flow.

References

10. SCoPE www.teisa.unican.es/scope
Abstract. In this work we present a novel methodology to verify formally specified data models. The elaborated approach includes the design of a formal check language – subsequently called MSCL (Model Specification Constraint Language) – used for verification. The MSCL allows expressing constraints on the specification. Besides the development of the MSCL a Python check engine was drafted to process MSCL constraints on the formal data model and to guarantee a neatly integration in the industrial Infineon meta modeling framework. Concepts and notations from established standards were picked up, reviewed or incorporated in a gainful way. First experimental results show that all required and relevant conditions can be formulated clearly and efficiently with the MSCL. Furthermore all constraints formulated in the MSCL can be conducted on any formal data model with the check engine.

1 Introduction

The importance of embedded systems has increased constantly over the last decade. These systems realize complex controlling and data handling tasks which should be often performed in safety-critical applications. In order to guarantee these crucial application constraints, it is necessary to fully integrate verification methods in the embedded system design flow. Today there exist several methods to verify formally specified models for example Computation Tree Logic (CTL)[1] or Linear Temporal Logic (LTL)[1]. The verification of the implementation can be ensured by using methods of software and system verification [2, 3]. Thereby formal equivalence checks, assertion-based verification (ABV) or formal model checks can be employed. A verification of the specification in typical embedded
system development flow results in huge efforts because often no formally defined specification is present.
The employment of meta modeling during the embedded system development provides the ability to integrate verification and validation methods already in the early specification phase. Using such a development technique leads to a formalized specification data model at an early development stage. Integrating such methods in the development process has the advantage of detecting errors early and of strongly reducing the costs for error removal and late design iterations.
In this work we propose a methodology to verify formal specifications of data models. For this purpose we integrate our method in the Infineon (IFX) meta modeling development flow. Due to industrial applicability, requirements concerning the language development and the implementation have to be satisfied by the check engine. These industrial demands are defined in detail in section 1.1 as well as a brief explanation of the IFX meta modeling development flow \((\text{metagen})\) and the structure of the used meta models. In section 1.2 we give a short overview on existing meta model verification techniques and state-of-the-art techniques to verify the specification model. Section 2 presents the model specification constraint language (MSCL) and section 3 deals with the implementation of the language and the check engine. In section 3.1 we briefly show some examples of MSCL constraints.

1.1 Preliminaries

Infineon meta modeling development flow (\text{metagen})

The IFX meta modeling development flow (\text{metagen}) is layered in three levels of abstraction. The first abstraction level consists of the meta model. In this level the components and the attributes of the data model are defined. The specification of the meta model is defined by a UML\^[4] class diagram. The exemplary representation and structure of such a meta model is explained in detail in section 1.1. The UML class diagram of the meta model is represented as a XML\^[5] data structure. This XML data structure is the input for a Python\^[6] generator which generates an API-based meta model. This consists of several Python classes, depending on the structure of the meta model.
The next abstraction level is the meta model instance which is represented by a filled instance of the API-based meta model. An instance can be filled with data using the generated API or by different other inputs as for example XML, IP-XACT\^[7] or Excel. Compared to the embedded system design flow these instances are located at the specification stage. They have to be checked by the MSCL to achieve the actual goal of embedded system specification validation.
The next development step is to generate different views from the model. These generators are written by the designer for every meta model as the generators are specific for every meta model and every resulting view. Examples for such views are C-code, SystemC\^[8] models or VHDL/Verilog models. This layer correlates to the executable model in the traditional development flow.
Structure and design of the meta model

In section 1.1, the meta models are defined by UML class diagrams. An example which consists of attributed classes and two different connection elements is given in Fig. 1. The first one is a has-a relation and denotes how many sub classes from one class type a class is allowed to have. The number of sub classes is specified by denoted multiplicity. In the example it is allowed that the class Component can have no RegMemSets or a variable number of RegMemSets. A RegMemSet (RMS) is a set of Memory and Register components. The other connection element realizes a reference relation. Thereby the corresponding class has no sub class of this class type but a reference to this class. This meta model structure can also be seen in Fig. 1 where an Interface has a reference to a variable number of RMS but none of these RMS are sub classes of the Interface.

Design and implementation requirements

The integration of the MSCL has to fulfill several requirements for making it industrially applicable within the scope of the IFX framework. In the following only the key requirements are mentioned. The formal constraint language must:

- be adaptive for arbitrary meta models
- be specified elegantly and legibly
- be qualifiable
- be resultant in a boolean result
- be connectable with boolean operators
- identify all objects in the data model
- possible to compare attributes and objects

Moreover there exist requirements for the implementation of the check engine. These are that the implementation should be done in Python to ease the integration. Another requirement is that the implementation is easily changeable and extendable. This is necessary because changes in the language structure should be easily adopted. The check engine should also use the structural information of the meta models in order to optimize the traversal algorithms as well as it
should use the context sensitive API generated by metagen to access the data objects within the data model.

1.2 Related Work

Schematron
Schematron is a rule-based validation language for XML documents and ISO/EC-standard since May 2006. It is a XML-based schema language with a small number of elements. It uses XPath for the addressing of elements in the XML document. Therefore Schematron is capable to express constraints on XML documents and to check them. Hence we cannot use Schematron in our context because it can exclusively be used for the validation of XML documents.

Object Constraint Language (OCL)
The Object Constraint Language (OCL) [11] is standardized by the Object Management Group (OMG)[12] and is a declarative standard to specify constraints for the compliance validation of UML models. OCL constraints are specified with the regard to the basis of the meta model and checked on the model level. Therefore, OCL satisfies the basic requirements which are defined by the meta modeling development focus but there does not exist a Python implementation of OCL which is required in section 1.1. Also we want to have a huge flexibility for changing the formal language due to requirements which could be raised by the industrial application. The consequence of these facts are, the necessity of implementing our formal check language and the corresponding check engine.

2 Model Specification Constraint Language (MSCL)
The MSCL is structured in six different layers. Each of these layers is separated explicitly in assignment and usage. Their logical structures are shown in Fig. 2. The following enumeration gives a brief overview of the layers.

![Fig. 2: MSCL Layer](image-url)
– The **Abstraction Layer** is the extension layer of the MSCL in which own operators, specific variables and data sets can be defined. Further it is possible to integrate different traversal algorithms. The operators and traversal algorithms are implemented in Python, due to the industrial requirements which are formulated in section 1.1. This layer is orthogonal to the other layers because it is not required for specifying MSCL constraints.

– The **Assertion Layer** is the top layer of the MSCL. It contains the severity level of the MSCL constraint, the report message and the MSCL property. The MSCL defines six different severity levels: *Debug, Info, Warning, Error, Critical* and *Failed* which qualify the constraint.

– The **Property Layer** defines different properties (simple properties) which can be connected to a MSCL property via boolean operators. Each of the simple properties is evaluated to a boolean value. The simple properties itself are partitioned in direct properties and indirect properties. The difference between these both types is explained in detail below.

– The **Set Layer** specifies the data set for which the constraint is evaluated. These data sets can be defined with notations like paths, class name definitions, iterators or iterator sets. All of these set definitions can be combined in one set definition and filter expressions can also be added to restrict the data set. Also this layer contains different set operators like *UNION, INTERSECT, DIFFERENCE, SUM, MIN* or *MAX*.

– The **Proposition Layer** is the condition layer of the MSCL. The proposition is evaluated for every data object defined in the set layer. For the specification of the proposition this layer provides an extended Python expression language. The expression language allows full Python expressions and some additional functionality like special evaluation functions, direct or indirect attribute references, etc.

– The **Link Layer** provides the access to data objects and their attributes. This layer is also used by the filter expressions of the set layer to limit the set of data objects.

A MSCL constraint specified by the six layers given above is always evaluated to three different values: *True*, *False* and *Failed*. The first two values are boolean values. The value *Failed* is no logical value but it displays that the MSCL constraint cannot be evaluated. The evaluation of the MSCL property is aborted and the value *Failed* is propagated through the layers to the Assertion Layer. In the example which is represented in Fig. 2 all MSCL language parts in the belonging MSCL layer are printed in bold. The MSCL constraint checks if all *Registers* are filled with non-overlapping *BitFields*. A *BitField* overlaps if its *Offset* minus its *DataWidth* is unequal the *Offset* of the preceding *BitField*. In the example the property layer is represented by a direct property with the *FORALL* operator to check the condition for all *BitFields*. Other direct property operators are *EXISTS, UQ* or *FORONE*. Direct property operators operate on a *SET_OF_VALUES* in contrary to the indirect property operators which work on a *SET_EXPRESSION* and a *EXPRESSION*. These operators, for example *EQ, NQ, GT, GE, LT, LE* or *EQSTRC*, need two parameters because they are
relations. The following EBNF rule shows the difference between a direct property and an indirect property.

**Rule 1**

```
DIRECT_PROPERTY ::= DIRECT_PROPERTY_OPERATOR
                  ( { DIRECT_PROPERTY } '"SET_OF_VALUES"' )

INDIRECT_PROPERTY ::= INDIRECT_PROPERTY_OPERATOR
                     ( '"SET_EXPRESSION"',' EXPRESSION"' )

SET_EXPRESSION ::= SET_COLLECTION_OPERATOR'("SET_OF_VALUES")'

SET_OF_VALUES ::= [ SORT_EXPRESSION ] PATH_EXPRESSION EXPRESSION
```

The data set for which the proposition of the MSCL constraint should be checked, is defined by a `PATH`. A `PATH` defines a dedicated address within the data model by a set of data objects. In the example it defines the path to all Bit-Fields in the data model. Other possibilities to specify sets are the usage of `CLASS_NAME_EXPRESSIONS` or `PATH_OPERATORS`. By using `CLASS_NAME_EXPRESSIONS` it is feasible to specify sets by the meta model class name of the data objects and an optional filter expression for some restriction of the set. The `PATH_OPERATORS`: `UNION`, `INTERSECT` or `DIFFERENCE` are different quantity operators which need two set definitions as parameters. These set definitions implemented the set concept of the MSCL and enables a huge flexibility in addressing data sets and to check these sets on consistency.

The definition of the consistency condition is adopted by the proposition layer. In the example in Fig. 2 the proposition checks if no BitField overlap another BitField in the current Register. This is realized by using some MSCL functions which extend the Python expression language. The first of these functions is the iterator function `PREDECESSOR`. The MSCL defines three types of iterators `PREDECESSOR`, `SUCCESSOR` and `PARENT`. These iterators also can be used in the set layer to check sequences of the data objects. The if-expression and the function `ISFIRST` in the proposition layer of the example in Fig. 2 is used because the first BitField of a Register cannot overlap another BitField. In the example the link layer is illustrated by the attribute access in the proposition layer. The attribute access is realized by the keyword `self` followed by the name of the attribute. In the MSCL the attribute access is separated in direct and relative attribute access. The access to the attributes `Offset` and `DataWidth` in the mathematical minus term is direct. In contrary, the access of the `Offset` of the preceding BitField is relative to the position of the current BitField. The relative access is expressed by the iterator definition in front of the attribute access definition. Another relative attribute access can be realized for example by using a relative path to an attribute in the proposition layer. The only restriction for such an access is that the path should lead to exactly one attribute.
3 MSCL checker

The MSCL checker engine combines the MSCL constraints and the data model traversal algorithm. Fig. 3 shows the MSCL check process. Currently two tree traversal algorithms are supported, the breadth-first search and the depth-first search. Other traversal algorithms are fully supported by the checker engine through the full support of the abstraction layer in our MSCL check engine implementation.

The combination between the constraints and the traversal is achieved by using the visitor pattern [14]. Hence each MSCL constraint is represented as a visitor object. These visitors traverse independently through the data model. This functionality is accomplished by reflection. Because of that, each data object within the data model is wrapped in a base element and gets a generic accept method. This method uses datatype introspection to identify the methods of the current data object which can be called to send the visitor to the sub data objects of the current data object. If available, this method uses meta model information to send the visitor only to sub-structures of the data model element where the data objects to be checked are located. Each visitor contains a generic visit method which implements the check functionality. At first it checks if the current data object is one of the searched objects and then it evaluates the consistency check. The MSCL constraint itself is represented as a Python abstract syntax tree (AST) to be able to modify the constraint dynamically. This is necessary if there are links within the MSCL constraint to other objects or attributes which are relative to the current data object. The AST representation is possible because the proposition layer is syntactically and semantically specified like the Python expression grammar. Hence it is feasible to support nearly every data model which is designed by the IFX metagen.

3.1 Examples

To evaluate our method, we specified constraints which has to be checked on defining embedded systems specification with metagen. Our first requirement
defines that all Registers in the formal data model should be totally filled with BitFields and none of these BitFields are allowed to overlap a other BitField. The listing 1.1 shows this requirement as a MSCL constraint. In the following examples the MSCL report message and the severity level are not displayed because of clarity.

1  \textsc{assert forall} (Component/RegMemSet/Register
2 \quad \{ self.DataWidth == \textsc{sum}(BitField/self.Width) \})

Listing 1.1: First MSCL constraint

For realizing this, a \textit{PATH} definition is used which defines a set of all \textit{Register} of the data model. In the proposition the MSCL function \textsc{sum} is used which takes an indirect reference to a set of attributes and sums it up. The MSCL check engine checks this constraint in 0.02 seconds on a data model which consists of 108 \textit{Registers} and overall 10380 BitFields. The next requirement to be checked is that all BitFields are writeable from hardware or software but not from both sides. The belonging MSCL constraint is displayed in listing 1.2.

1  \textsc{assert forall}(\textit{classes} BitField
2 \quad \{ "WRITE" not in self.AccessExternal \textbf{if} "WRITE" in self.AccessInternal
3 \quad \textbf{else} "WRITE" in self.AccessExternal \})

Listing 1.2: Second MSCL constraint

The specification of the data set in this MSCL constraint is done by a class definition. Hence no path to the BitFields is defined and only the keyword \textit{classes} and the instance name of the BitFields is denoted. The proposition is defined by an if-then-else condition to consider the side-effects of this constraint. The MSCL engine checks this constraint in about 0.16 seconds. The MSCL constraints are performed on an Intel Core 2 Quad 2.50 GHz with 4 GB of RAM using Ubuntu 11.10 64bit.

4 Conclusion

We presented a method to verify specifications of embedded systems which are defined as formal data models based on meta modeling development. Therefore we designed a new formal check language called MSCL in order to easy formulate clear and elegant constraints. The MSCL gives the ability to perform consistency checks on arbitrary specification data models which are generated by the IFX meta modeling infrastructure metagen. Within the MSCL we implemented a set concept which enables the addressing of sets of data objects via iterators to check sequences of data objects, path definitions to choose data sets by there structure in the data model and class name definitions to select sets of data objects by their instance name in the meta model. Because of this concept it is possible to efficiently select arbitrary parts of the data model. The conditions for the checks are defined in the proposition layer which is implemented as an extended Python expression layer. Due to this the proposition layer has an extensive functionality to formulate constraints which is partly shown in our examples in section 3.1. Nevertheless the MSCL itself can be easily extended by an extension layer, the
so called abstraction layer. This layer gives the ability to specify user-defined operators, variables and traversal algorithms which makes the MSCL much more powerful in the specification of constraints. Furthermore we briefly introduced our MSCL check engine which is integrated seamlessly in metagen and allows to check the specifications of embedded systems generated by metagen. In the future, we want to make deep performance evaluations and memory optimizations in order to show that our approach is scalable and applicable in an industrial settings.

Acknowledgment

This work has been developed in the project SANITAS. SANITAS (project label: 01 M 3088) is partly funded by the German ministry of education and research (BMBF) within the Research Program ICT 2020.

References

4. OMG: Omg unified modeling language (2011)
Verification of Embedded Systems
Using Modeling and Implementation Languages

Mathias Soeken¹,³, Heinz Riener², Robert Wille¹, Görschwin Fey², and Rolf Drechsler¹,³

¹ Institute of Computer Science, University of Bremen
Group of Computer Architecture, D-28359 Bremen, Germany
² Institute of Computer Science, University of Bremen
Group of Reliable Embedded Systems, D-28359 Bremen, Germany
³ Cyber-Physical Systems
DFKI GmbH, D-28359 Bremen, Germany
{msoeken, briener, rwille, fey, drechsle}@informatik.uni-bremen.de

Abstract. We propose a verification flow which aims at functional verification of embedded systems. We assume that the system is formally specified in a modeling language and implemented in a high-level programming language. The verification flow proceeds in two stages: first, we check for behavioral properties which indicate bad states at the level of the specification. Second, we prove that the individual components of the system conform to their specification. The first stage allows for the detection of design flaws in the system without considering implementation details, whereas the second stage refines the verification task allowing for the detection of functional bugs in the implementation. In both stages, the description of the verification task is automatically generated from the modeling language and the implementation language, respectively.

1 Introduction

Guaranteeing functional correctness of embedded systems is time-consuming and challenging. In modern system design flows, checking for functional correctness, i.e., functional verification, accounts for more than half of the total development time [1]. Due to time-to-market constraints, however, push-button approaches are desirable.

Formal methods aim at automatically checking whether a system implementation conforms to a formal specification. One such method is model checking [2,3] which exhaustively explores the state space of a system implementation and generates a counterexample if and only if the formal specification does not hold. Model checking has successfully been used to prove functional correctness of hardware designs at the gate and register-transfer levels. Today, research focuses on lifting functional verification to higher abstraction levels, e.g., the Electronic System Level (ESL) [4–6]. At the ESL, a designer is not only faced with guaranteeing functional correctness of an isolated component but has to consider multiple components interacting with each other and their environment. The main challenge at the ESL is to tackle the possibly large state space to overcome the state-explosion problem.

Previous work includes assume-guarantee reasoning [7] and synthesis from specification [8,9]. Assume-guarantee reasoning attempts to prove or refute a property by utilizing the specifications of individual components to allow for a step-wise refinement of the property. Synthesis generates a system which is correct-by-construction from a formal specification. Both techniques, however, are computationally expensive. Moreover, providing a formal specification which entirely describes the functionality of a system is challenging for larger system designs.

In this paper, we propose a functional verification flow focusing at the ESL by lifting parts of the verification technique to a meta-modeling level. Given a formal specification
in a modeling language and an implementation in a high-level programming language, our verification flow proceeds in two stages.

In the first stage, we check the formal specification for behavioral properties which indicate bad states. For instance, we verify that a system is free from deadlocks. For this purpose, the formal specification and its properties are translated to instances of the Satisfiability (SAT) problem. We abstract from the precise implementation of the components using the formal specification as an abstract, behavioral model. The behavioral model is used to detect flaws in the design in the absence of a precise implementation.

Afterwards code representing the structure can automatically be generated from the formal specification, i.e., only the precise implementation is left to the designer. Hence, in the second stage just the correctness of these implementations has to be checked, i.e., we refine the verification task leveraging the implementation and check whether the individual components adhere to the formal specification. Our verification flow allows for an early detection of design flaws and considers the implementation to search for functional bugs only when necessary. To this end, the description of the verification task is automatically generated from the modeling language and implementation language, respectively.

In the following, we use the Unified Modeling Language (UML) [10] and the programming language C++ to describe the structure and the behavior of components. Our verification flow, however, applies to other modeling and programming languages, too. Moreover, we restrict our perspective to finite-state systems, i.e., the number of components in the system is fixed.

The remainder of the paper is structured as follows. In Section 2, we describe our terminology. In Section 3, we introduce our verification flow and present an implementation using SAT-based Bounded Model Checking (BMC) [11]. Section 4 concludes the paper.

2 Preliminaries

We introduce the terminology used in the paper by means of the example shown in Fig. 1: Fig. 1(a) shows a class diagram which describes the structure of the classes Host and Client. Additionally, the class diagram is annotated with formal constraints leveraging UML’s Object Constraint Language (OCL) [12]. These constraints serve as a formal specification. Fig. 1(b) shows an object diagram which describes the state of a
system consisting of one client and one host component. Lastly, Fig. 1(c) sketches the actual implementation of the client component in C++.

In Fig. 1(a) the structure of all host and all client components is described by classes. We call these classes Host and Client, respectively, and we say that the individual host and client components are instances (or objects) of the respective classes. A class provides attributes and operations. The attributes serve as variables which store values. The operations define behavior which can be invoked and modify the values of the attributes. For instance, the class Client in Fig. 1(a) has two attributes active and data and one operation connect. The connect operation can be invoked to establish a connection between a client and a host component which modifies the value of the variable active.

The valuation of the attributes of all instances of a system comprise the state of the system. For instance, Fig. 1(b) shows a state $\sigma_t$, $t \in \mathbb{N}$, of a system, with the assignment $\text{hdata} = 42$, $\text{active} = \text{false}$, and $\text{data} = 0$. When the connect operation is invoked in state $\sigma_t$, the attribute active is set to true resulting in a new state $\sigma_{t+1}$. We say that there is a transition from a state $\sigma_t$ to another state $\sigma_{t+1}$ if and only if an operation $o$ exists which results in state $\sigma_{t+1}$ when invoked in state $\sigma_t$. We denote this transition by $\sigma_t \rightarrow_o \sigma_{t+1}$. The object diagram in Fig. 1(b) shows a possible sequence of transitions starting from state $\sigma_t$ where the connect operation is invoked multiple times.

We use OCL constraints to define formal properties for classes. The OCL constraints are annotated into the class diagram using UML note elements, e.g., in Fig. 1(a). We distinguish three types of OCL constraints: an invariant $I$, a precondition $\preceq$, and a postcondition $\succeq$. We use $I$ and $O$ to denote the sets of all invariants and all operations, respectively. An OCL constraint $\varphi \in \{I, \preceq, \succeq\}$ holds in a state $\sigma$ denoted by $\varphi(\sigma)$ if and only if the respective state $\sigma$ satisfies the condition $\varphi$.

The invariants define global constraints which apply to all instances of a class. Each invariant of a class has to hold for all states. For example, in Fig. 1(a), the value of attribute hdata has to satisfy $\text{hdata} < 256$ for all instances of the class Host in all states.

Preconditions and postconditions define constraints which apply to a particular operation of a class. We use $\preceq_o$ and $\succeq_o$ with subscript $o$ to denote the pre- and post-condition of operation $o \in O$. The invocation of operation $o$ in a state $\sigma$ is valid if and only if $\preceq_o(\sigma)$ holds.

3 Verification Flow

In this section, we first present the general idea underlying our verification flow in Section 3.1. We then describe the verification tasks of the first and the second stage in Section 3.2 and Section 3.3, respectively. Finally, we discuss implementation details in Section 3.4.

3.1 General Idea

The proposed verification flow is shown in Fig. 2. The inputs of the verification flow are a formal specification described by a UML class diagram annotated with OCL constraints and an implementation of a system written in a high-level programming language. In the figure, we have separated the two stages of the verification flow in two boxes with dashed borders. The top box denotes the first stage which focuses on the class diagram. The bottom box denotes the second stage which uses the implementation to refine the verification task.

The interaction of the components are conducted in the first stage in the absence of an implementation. If the verification on the specification level fails, design bugs can be determined and fixed without considering implementation details. If no further
When design bugs are found, the individual components are verified in the second stage. For this purpose, also information from the specification level is required. This particularly includes the respective pre- and postcondition from the operations as well as the invariants specified in the class diagram. In the second stage, functional bugs are detected which may be caused by an erroneous implementation of operations.

### 3.2 Verification of System Behavior

In the first stage, we check whether a verification condition $\tau$ can be refuted for a fixed system configuration in the absence of an implementation. We use the annotated constraints in a UML class diagram as a formal specification. The negated verification condition $\neg \tau$ denotes a formal property which characterizes bad behavior in the system, e.g., a possible deadlock situation.

The system configuration provides a fully or partially defined initial state of the system $\sigma_0$, i.e., the number of components is fixed and their attributes are assigned to precise values. We automatically generate a logic formula using a BMC approach. The logic formula encodes a reachability problem on a state-transition graph, where the states correspond to the states of the system and the transitions correspond to the formal specification of the operations. The verification condition defines the states which must not be reachable. A satisfying assignment of the logic formula serves as a counterexample for the verification condition, e.g., a deadlock is a sequence of operation invocations from which the invocation of no other operation is valid.

The generated logic formula for the verification task of the first stage is shown in (1), where $I$ is the set of invariants.

$$\bigwedge_{i=0}^{k-1} (\bigwedge_{l \in \mathcal{I}} I(\sigma_l) \land (\bigwedge_{i=0}^{k-1} (\prec_{\alpha_i}(\sigma_l) \land \succ_{\alpha_i}(\sigma_{l+1})) ) \land \neg \tau(\sigma_k)$$  

(1)
The satisfiability of the formula is decided utilizing a state-of-the-art solvers for the SAT problem. From a satisfying assignment, a sequence of operation invocations $o_0, \ldots, o_{k-1}$ is derived such that all invariants, pre- and postconditions, and the negated verification condition are satisfied. The length of the sequence of invoked operations is bounded by $k$, i.e., if no counterexample is found within the bound, $k$ has to be increased, up to a reasonable value.

The verification condition $\tau$ can be adjusted for a particular pattern of bad behavior to be checked. For instance, to check for a deadlock we use
\[
\tau(\sigma) = \bigvee_{o \in O} c_o(\sigma),
\]
i.e., at least one precondition of any operation has to hold in state $\sigma$.

### 3.3 Verification of Individual Components

In the second stage, we check whether the individual components are functionally correct using the implementation written in a high-level programming language to refine the verification task. Also here, we generate a logic formula for each operation which is satisfiable if and only if the implementation of an operation $o$ does not conform to the formal specification. The logic formula is built by encoding the implementation of an operation as a logic formula $\text{impl}_o$ conjoined with the precondition, the postcondition, and the invariants.

We consider two states $\sigma_t$ and $\sigma_{t+1}$ denoting the system before and after the operation $o$ has been invoked. Furthermore we assume that the invariants are correct, i.e. our approach considers an underapproximation of the state space if the invariants are too strict. The problem whether the implementation of the operation is functionally correct, i.e., the implementation conforms to the specification, is formalized as (3):
\[
\bigwedge_{I \in I} (I(\sigma_t) \land I(\sigma_{t+1}) \land c_o(\sigma_t) \land \text{impl}_o(\sigma_t, \sigma_{t+1}) \land \neg q_o(\sigma_{t+1}))
\]

The logic formula checks for the existence of two states $\sigma_t$ and $\sigma_{t+1}$ such that $\sigma_t$ and $\sigma_{t+1}$ satisfy the system invariants $I$, $\sigma_t$ satisfies the precondition $c_o$, $\sigma_{t+1}$ violates the postcondition $q_o$, and $\text{impl}_o$ describes the behavior of the operation’s implementation, i.e., the transition $\sigma_t \to o \sigma_{t+1}$ with $o$. A satisfying assignment corresponds to a counterexample which proves an inconsistency of the formal specification and the implementation. The counterexample can be used to analyze and fix the inconsistency. Otherwise, if the logic formula is unsatisfiable, no such counterexample exists, i.e., the operation’s implementation conforms to the formal specification.

### 3.4 Prototypical Tool

We have built a prototypical tool which implements the described verification flow. This tool generates a logic formula representing the considered verification tasks and utilizes a Satisfiability Modulo Theories (SMT) solver to determine a solution for it. The generation of the corresponding SMT solver input from the OCL constraints is described in [14, 13]. To generate the logic formula from the C++ source code, we leverage the Low Level Virtual Machine (LLVM) [15] compiler infrastructure as described in [16]. First, we translate C++ to an LLVM Intermediate Representation (LLVM-IR). Then, we use a BMC approach [11] to translate the LLVM-IR into a logic formula. We unroll loops in the LLVM-IR for a fixed number of iterations, introduce one logic variable each time a program variable is written, and encode the individual instructions to semantically equivalent logic constraints. Finally, we conjoin the logic formula encoding the OCL constraints and the logic formula encoding the implementation and assert correlating variables in the resulting logic formula to be equal.
4 Conclusions

We proposed a two-staged verification flow which focuses on the ESL. The verification flow is applicable to a system which is formally specified in a modeling language and implemented in a high-level programming language. In the first stage, we use light-weight model checking to detect design bugs early without considering the implementation details. Thus, the first stage can be used even if the implementation is not available. In the second stage, we refine the verification task leveraging the implementation to detect functional bugs. In both stages, the description of the verification task for the solving engine is automatically generated.

References