Hardware Synthesis of UML-Models

Thomas Beierlein
Hochschule Mittweida
Labor Embedded Control
tb@htwm.de

Dominik Fröhlich
TU Bergakademie Freiberg
Inst. of Computer Science
dfroehli@htwm.de

Bernd Steinbach
TU Bergakademie Freiberg
Inst. of Computer Science
steinb@informatik.tu-freiberg.de

Abstract
The automated synthesis of application specific co-processors has been a complex and demanding problem. To handle the inherent informational and architectural complexity of this problem novel development approaches supported by powerful tools are taken. In this paper we present an approach that is based on UML and hardware/software co-design. We will show how Boolean problem solvers can be modeled with this language and discuss the synthesis of hardware from such models. Also we show important problems we recognized with the current approaches that should be investigated in the future.

1 Introduction
The development of application specific co-processors is a recent trend in many application domains with calculation-intensive problems. A system-level specification of a problem is partitioned into software and hardware modules such that the performance of the resulting implementation is optimum in terms of performance. The software modules execute the global control flow, and algorithms that are either uncritical to performance or too costly to be implemented in hardware. The hardware modules, comprising the application specific co-processor, implement the calculation-intensive algorithms. To allow for a fast adaption of the co-processor to the problem reconfigurable logic resources, like field-programmable gate-arrays (FPGA), are commonly used as implementation technology.

However, despite its huge potential this approach is quite seldom taken today, because the development of mixed hardware/software solutions is still a costly and error-prone task. In spite of the recent progress in high-level synthesis and verification the gap between the problem-level and the implementation-level is still too large to be bridged in an efficient and straightforward manner. Solutions to this problem must allow for system-level specifications with languages convenient to the application domain while still enabling the automated transformation of these specifications into ready-to-run implementations.

Our recent research focuses on the development of mixed hardware/software solutions for computation-intensive applications with the Unified Modeling Language (UML) [7]. Of course, the academic and industrial efforts to develop such systems with UML are not new. However, the most of these approaches have a strong focus on software. Common problems like, hardware/software partitioning, estimation, hardware/software synthesis, and verification are not sufficiently addressed [11]. Much of the related work can be found in the embedded systems domain. In [8] Rosenstiel et al. propose a framework for the object-oriented specification and partitioning. This work was recently extended to support the design entry with UML. From the classes in the UML model they generate skeletons of classes and methods, which are then completed manually and refined using traditional methods. This break in the tool-chains and paradigms is still very archetypal also to most tool-chains for UML. Other approaches use UML for the specification of systems with System-C [3], [12].

In this paper we present an approach that allows for the complete, precise and object-oriented specification of applications with UML 2.0 and its automated compilation into a directly executable mixed hardware/software solution. Our approach supports model validation, hardware/software partitioning, and synthesis. In contrast to the related works the whole development from specification to synthesis is performed directly on and driven by UML-models. A UML model compiler, MOCCA (MOdel Compiler for re-Configurable Architectures), implements our methodology in that it automatically performs vali-
dation, platform mapping, and application synthesis. The targeted class of application domains includes Boolean problem solvers (XBOOLE-system [2]) and Boolean neural-networks [6].

The rest of this paper is organized as follows. In section 2 we briefly outline our development approach and give a brief example of its application for a common Boolean problem. For a detailed discussion of the development methodology we refer the interested reader to [1] and [9]. In the core of this paper, in section 3, we focus on the synthesis of object-oriented concepts to reconfigurable hardware. Finally, in section 4 this paper is concluded.

2 Development Approach

In our approach the application as well as the employed platforms used for design, implementation, and deployment are described with with UML and the MOCCA action language (MAL) [1]. The overall development approach is very common to the most object-oriented development efforts.

The design of the application is captured as an object-oriented system-level UML-model. The system structure is defined with packages, classes, and interfaces. System behavior is defined with operations and state-machines. For the detailed behavior specification we specified the MOCCA action language (MAL). This language provides a large number of arithmetical, logical, and relational operators and constructs for control specification, which makes it convenient for the targeted application domains. The syntax of MAL orients toward the Java programming language.

Example 1 In Figure 1 an example design model is shown. It comprises of the three classes - Demo, TVLUtilities, and ExtTVLUtilities. Demo defines an operation main, which implements an algorithm that uses ternary vector lists (TVL). At some point in the algorithm (shown in the note) a given TVL must be orthogonalized. The orthogonalization-algorithm may be implemented by the operation orthogonalize in the classes TVLUtilities and ExtTVLUtilities. TVLUtilities implements the standard algorithm. This implementation of the orthogonalize-operation is overridden in ExtTVLUtilities in order to additionally perform block-building.

![Figure 1: Example: Orthogonalization of Ternary Vector Lists](image)

Given such a design model and a definition of our target hardware architecture, which is also described as UML-model, we proceed by mapping the application to the target hardware. This mapping is done such that the performance of the synthesized implementation realizes the specified application and exploits the performance gains offered by the underlying hardware.

Example 2 Figure 2 illustrates the implementation and deployment of the application from example This example completes the implementation and deployment specification of the design in example 1 on a given hardware architecture. Orthogonalization is used very often in TVL-arithmetic, also the according algorithm is quite calculation-intensive. Thus it may be advantageous to execute it in reconfigurable logic, in order to optimally exploit the parallelism inherent to the algorithm. Hence, the
classes TVLUtilities and ExtTVLUtilities are deployed on the FPGAs by means of component MathComponent. The main-class Demo is deployed on the system master node.

In the final synthesis step we realize the implementation and deployment model that was created during the previous platform mapping step. For the functionality deployed on micro-processors we generate software modules; the functionality that will be executed on reconfigurable logic respective hardware modules are generated. Also we have to generate functionality for the communication and synchronization between the hardware and the software parts. The hardware modules realize the application specific co-processor.

The MOCCA model compiler implements this approach in that it automatically performs validation, optimization, platform mapping, and synthesis. This tools allows for an automated transformation of the system-specification into directly executable hardware/software implementations.

3 Hardware Synthesis of UML-Models

3.1 Overview

In the previous section we briefly presented our development approach. We have shown the transformation of user design models into implementation and deployment models. These models define the realization of the user design on a given target platform. In this section we will present an object-oriented approach to synthesis of hardware from UML-models. In an object-oriented approach it is quite straightforward to implement the classes deployed on reconfigurable hardware and to instantiate them on-demand at run-time. However, this approach raises several issues relevant to logic synthesis. For the purpose of this paper we will concentrate on the dynamic creation/destruction of hardware objects and their implementation in the presence of polymorphic features.

3.2 Creation and Destruction of Hardware Objects

The dynamic creation and destruction of objects in hardware is a problem due to the static nature even of reconfigurable hardware. Even if the hardware is partially reconfigurable, which would allow for the
implementation of each class in a partial configuration, this raises significant problems during synthesis, verification, tool support, and also implementation. One problem is that the configurations are not independent from each other, because the objects have to share the same physical communication interface. The class instantiation per reconfigurable device is by far too expensive in terms of required and wasted area and device reconfiguration time.

Because of these problems we chose another approach. Instead of mapping each class to a configuration, we cluster multiple objects in a configuration (hardware objects). The number and type of the hardware objects being clustered in a single configuration is determined either manually or automatically by MOCCA [4]. Otherwise it tries to reconfigure the hardware with an appropriate configuration first. Figure 3 shows a template of a respective hardware configuration.

![Figure 3: Hardware Configuration Template](image)

The hardware objects are pre-instantiated at compilation time in the UML-components realized by the object classes. The pre-instantiated objects serve as templates that can be instantiated at run-time, which is exactly the notion of a class in the context of object-orientation. The actual creation/destruction of the hardware objects is delegated to a run-time service, called RTR-manager. The software part of the application may dynamically request hardware objects from the RTR-manager. This service checks if an object of the requested type is available in one of the configurations currently bound to logic resources. If so, it post-instantiates the object, marks it used, and serves the object to the application.

### 3.3 Implementation of Hardware Objects

If we assume that there may be multiple instances of the same class at any time, this implies that the application may get different hardware objects at each single point of creation. Thus, in order to implement the generalization of objects properly, we have to ensure that the interface of all objects whose classes are in a sub-type relationship in the inheritance hierarchy are compatible. The interface of an object comprises of all features (attributes and operations) in its native class definition and the features inherited from its super-classes.

The implementation of attributes is straightforward, they are mapped to a register of an appropriate width. When implementing operations we have to consider that they may be polymorphic. Polymorphic means that there are multiple implementations of an operation are available and visible in an object. Which implementation actually is executed when the operation is invoked depends on the dynamic type of the object. The known approaches to the implementation of object-oriented systems with hardware, do not support inheritance and polymorphic operations. They require that the classes of hardware objects do not specialize other (non-trivial or empty) classes. This way polymorphic operations are avoided.

Our recent research has shown two things: the prohibition of non-trivial inheritance and polymorphism limits the usefulness of the object-oriented approach dramatically, and allowing for inheritance typically leads to a better hardware utilization and may increase the performance of RTR-applications [5]. Hardware utilization is improved because modeling with inheritance typically means modeling only differences between the sub-class and its super-class. We reflect this observation in the actual implementation, and allow for switching the available and executed features at the creation time of hardware objects according to the dynamic type of the object. The run-time performance may be improved, because the expensive hardware reconfiguration can be avoided in certain situations.

The interface of hardware objects must be independent from the concrete object template and the
For this to accomplish the position of the registers comprising the public object interfaces relative to the start of the object interface must be equal within the class hierarchy. Additionally, polymorphic operations must share the same data and control registers.

**Example 3** Figure 4 shows the implementation of objects with polymorphic features for our example design in Figure 2. We assume that configuration MathConfiguration contains two objects, one of class TVLUtilities and one of class ExtTVLUtilities respectively. One possible register layout is shown in Sub-figure 4(a). Object0 starts at address 0x0 with respect to the start of the register-file, which may be mapped to the physical address 0xA000. Object1 starts at relative address 0x20 (0xA020). The relative register layout of both objects is the same. Object0 overrides the operation orthogonalize, but allows for switching its dynamic type (and hence the actually executed implementation) to TVLUtilities, by means of a type-ID (TID) written to the type register. The according logic is illustrated in Sub-figure 4(b). Object1 does not require a type-register, but still has to preserve the register layout.

![Object-Interface](image.png)

![Implementation of orthogonalize](image2.png)

Figure 4: Example Implementation of objects with polymorphic operation orthogonalize

Currently the logic synthesis of the behaviors of operations is delegated to HLS-tools, like SPARK [10]. Due to the recent advances in this field, the logic synthesized by modern HLS-tools is of a high quality in terms of performance and required area. However, this approach also reveals several problems.

Crossing the border from system-level to the (high) algorithmic-level, by means of an intermediate representation, typically leads to a loss of control and information. The system-level is the only level at which all information on the implementation and deployment of the developed application exists. In contrast, the algorithmic-level focuses on the implementation of behaviors local to some realization unit. However, on the transition between both levels valuable information that could be used to direct the actual implementation is lost. For instance, on the system level during the platform mapping step we explore the alternatives of the actual implementation of behaviors in terms of primitive operations provided by some implementation platform. Although the actual choice is reflected in the implementation and deployment models this information typically can not directly be forwarded to HLS-tools. Thus, we lose control on the actual synthesis. Moreover, the system-level estimation of the implementation characteristics is aggravated.

Another example of this kind of loss of context-information and control are estimates on the global message exchange and the execution probability and frequency of behaviors, which could be used in order to direct the optimization effort. In general, the optimization effort for a behavior should depend on its actual utilization. Behaviors that will never be executed concurrently could share registers and operators. We expect the potential of global optimizations to be quite large. However, with the current approaches to separate algorithm synthesis from system-level synthesis this potential is not used. In fact, the same arguments also hold for the software compilation from system-level specifications. Thus, we consider the research on system-level approaches to software and hardware synthesis, verification, and estimation as important problem for the future research.
4 Conclusions

In this paper we presented a novel approach to the development and compilation of UML-models for reconfigurable hardware architectures. The presented approach is based on the Unified Modeling Language and hardware/software co-design. On the problem of orthogonalization of TVLs we exemplified the modeling of Boolean problems with this approach. We also introduced a novel type of compiler that is able to synthesize hardware/software implementations from UML-models.

Currently we are validating and evaluating our development approach on calculation-intensive applications. The first results of the evaluation are very promising. Especially the good support for the system-level exploration of design alternatives at the UML-level and the immediate translatability into an executable implementation with hardware and software is very appealing to users. We are working on the QoS-benchmarking of our development environment to obtain information on the quality of the generated results. This will allow us present experimental results in the final paper.

Finally we showed some important problems for the future research on system-level hardware/software synthesis. We stated that, in contrast to current approaches, system-level synthesis should imply logic synthesis and software generation. This is necessary in order to reflect system-level decisions in the actually synthesized development artifacts. Also this approach enables for a number of yet uninvestigated optimizations.

References