Synthesis of UML-Models for Reconfigurable Hardware

Christina Dorotska, Bernd Steinbach, and Dominik Fröhlich

Technische Universität Freiberg
Institute of Computer Science
Germany

Abstract. The Unified Modeling Language (UML) is becoming an important competitor language for the development of hardware/software systems. The capability of transforming UML-models into hardware/software implementations is a key to this approach. While the generation of software from UML-models ist state-of-the-art, this article presents a novel approach to the direct synthesis for reconfigurable hardware.

1 Introduction

Reconfigurable architectures are an important class of hardware architectures, which combines the flexbility of traditional microprocessors with the performance gains of customized datapaths. *systems-on-chip* (SoC) is the main application of such architectures, besides high performance computing. This is mainly because, reconfigurable architectures enable the acceleration of the overall system whilst cutting development and manufacturing costs.

Another trend in the SoC domain is the development on increasingly higher levels of abstraction and the integration of modelling technology into the development flow. In this area object-oriented system-level modelling using UML is considered a very promising promising direction. The capability of synthesizing hardware implementations as customized datapaths directly from the UML-models is a key technology, because it can avoid manual and error-prone translations and thereby allows for a better handling of design iterations in the reality of system engineering.

In order to avoid the traditional breaks in paradigms, languages and tools, and to give the designer the maximum freedom when creating system designs, hardware synthesis from object-oriented models should preserve the object-oriented properties of the design in the final implementation. In this paper we present an approach to the synthesis of UML models to customized datapaths. In contrast to current behavioral synthesis this approach supports the full implementation of object-oriented models into customized datapaths [6][7][2].

The presented approach is based on the *Unified Modeling Language* (UML) and a dedicated action language. UML 2.0 and the action language are used for the object-oriented system specification and design at the *system level* of abstraction. The approach is backed by a dedicated tool called the MOCCA

compiler (Model Compiler for reconfigurable architectures). Given complete and precise models of the applications design and the design and implementation platforms, MOCCA can automatically perform partitioning, estimation, and the implementation of the system into hardware/software modules. The synthesized implementation is directly executable and exploits the capabilities of the hardware architecture.

A system is designed as executable and platform independent realization of the systems use-cases. This design model defines the structure and behavior of the system. System structure is defined in terms of UML classes and interfaces and their relationships, like dependency, generalization, and implementation. For concurrency specification we support active classes. System behavior is defined with operations, state-machines, and the MOCCA action language. Synchronization between concurrent control flows is supported with guarded operations. The model of computation is objects communicating through structured messages.

The direct implementation of such models in hardware is problematic due to the inherent dynamics of object-oriented models, polymorphism, and the communication between objects. In the following sections we outline the key concepts of objkect-oriented hardware synthesis. For the lack of space we do not detail modeling, design space exploration and software generation here nor we can give a thorough discussion of an application example. For a detailed discussion of these topics and the related work the reader is referred to [4], [1] and [5].

2 The Hardware/Software Interface

The hardware/software interface of object-oriented implementations with reconfigurable hardware defines the life cycle and access mechanisms of objects and components realized in reconfigurable hardware.

For efficiency reasons the life cycle of hardware objects is different from the life cycle of software objects. In order to avoid costly reconfigurations hardware objects are reused as much as possible. Because a true dynamic instantiation/destruction of objects is not efficiently possible in hardware, these objects are pre-instantiated at compilation time and synthesized into configuration bitstreams. The objects are dynamically allocated on demand. Object brokerage is performed by a dedicated software service. Additionally, in RTR systems the objects and hardware configurations are dynamically bound to logic resources.

The mechanisms for the access of objects are defined by the object interfaces. The interface of each object consists of a control interface, a data interface, and an exception interface.

Control Interface The control interface allows for identification, typing, and access to the object. This interface contains the object address, type and signals for the execution of the operations provided by the object.

Data Interface The data interface allows one to access the object state and to pass input/output parameters from and to objects. The interface contains the entire public state of the object and the parameters to/from services that are publicly accessible.

Exception Interface The exception interface reflects exceptional conditions that occur in the object. Depending on the exception handling of the object the information on the position and type of exceptions is represented.

The interface of each component representing a configuration bitstream comprises all interfaces of the objects accessible through the component interface. An implementation of the hardware object and component interfaces is presented in the next sections.

3 Hardware Implementation of Components

Figure 1 shows a template for the implementation of UML components in hardware. The component contains the pre-implemented objects O_i. The objects $O_0, ..., O_{n+4}$ are accessible via the communication interface. All other objects are not directly visible outside the component. The component also contains central circuits for the generation of clock and reset signals. There may be specialized clock generators for dedicated modules, such as external ZBT RAM (zero bus turnaround random access memory) or peripheral devices.

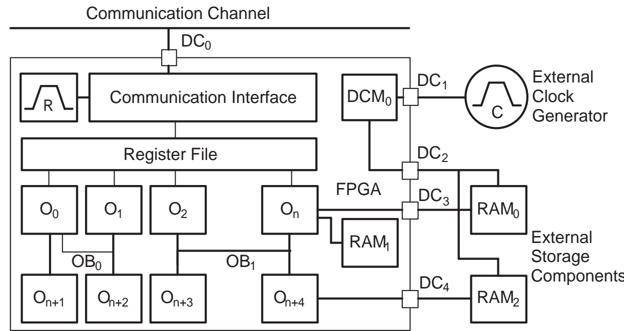

Fig. 1. Hardware Component Template

The presented approach can integrate pre-implemented hardware building blocks into the synthesized hardware. As illustrated in [5] IP is modeled in a *implementation platform model* using components, classes and interfaces. The model defines the interface of the building blocks, that can be used to establish connections between internal logic resources and external devices through the physical device interface.

Required components, that are modeled in the implementation platform model, are instantiated on demand. For instance, if RAM_0 was realized with `ZBT RAM Bank 0`, a `DCM Clock` (Digital Clock Manager) component is instantiated (DCM_0) and connected to RAM_0. The adaption of the external to the local

interface is realized by an appropriate VHDL wrapper component. The reset and clock signals of the design are automatically generated by the model compiler.

Multiple objects are clustered in a hardware configuration. The number and type of the hardware objects being clustered in a single configuration is determined either manually or automatically. For this the global message exchange of classes and their object creation/destruction characteristics is analyzed.

The public interface of publicly visible objects is realized by a register file. The register file allows one to access the control, data, and exception interfaces of the respective hardware objects. Collaborating objects communicate via direct connections or object buses (OB_i). In order to minimize bus contention there may be multiple object buses in one component.

Only the publicly visible objects can be instantiated by the software portion of the application. The other objects are hidden from the outside and are used as helper objects within the component. The access to an object of a given type must be independent of the object template and the dynamic object type. In order to accomplish this the public object interfaces of all objects of a given type and all of its subtypes must be identical. The realization of polymorphism is hidden behind the external object interface.

4 Hardware Implementation of Objects

Figure 2 shows a template for the implementation of objects in hardware. The template shows the implementation of one object. The object contains the pre-implemented operation O_i, the operation parameters P_i and the attributes A_i. The object can be of type T_0 or T_1. The sets P_i and A_i comprise the data interface of the object. Operation O_2 has a non-empty exception interface.

For each type the visible features must be provided in the interface. The set of visible operations is a superset of the operations that are defined by a type. Model compilers automatically eliminate unused features before design space exploration.

A control and type register implement the control interface. The type register holds the current dynamic type of the object. The control register contains the two signals *GO* and *DONE* for each operation. An operation is started by setting its *GO* signal. The end of execution is signaled by the operation when it sets its *DONE* signal.

Operation O_1 is polymorphic, that is, the executed behavior depends on the current object type. The selection of the behavior to execute is performed by a selector circuit whose implementation is depicted on the right hand side of Figure 2. Because the execution of both implementations of O_1 does not necessarily require the same time, the selector must also multiplex the *DONE* signal.

The object interface hides the execution of polymorphic behavior; that is, for the sender of a message which is handled by O_1 it is not relevant which implementation of this operation is executed. Consequently both implementations of O_1 share the same data interface. If both behaviors of the operations change data in their data interface, the implementations must be decoupled from the

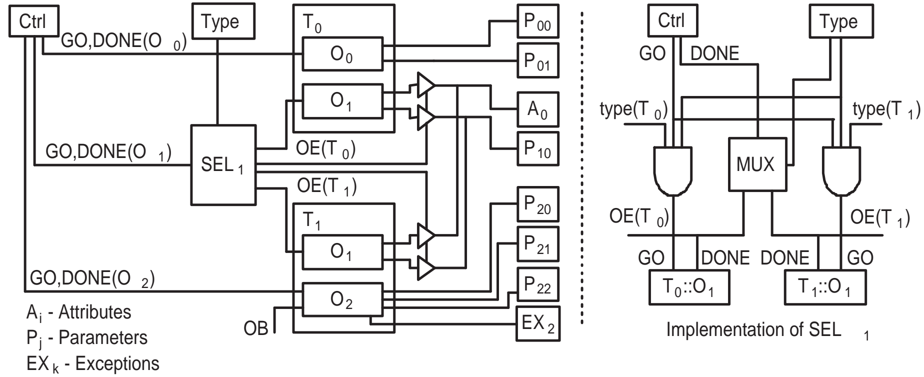

Fig. 2. Hardware Object Template

actual data by appropriate logic. If no other output enable was specified in the model the *DONE* signal is used.

The implementation of attributes and parameters is straightforward; they are mapped to a storage component of an appropriate width. If the implementation platform contains components with appropriate interfaces, the resources modeled are used to generate the storage components. The data interface of publicly visible objects is located in the register file.

Objects are connected to the data interface and other objects with direct connections or buses. This raises a synchronisation problem because multiple objects or operations may access features concurrently. Synchronization of publicly visible objects is performed in software. Concurrent accesses of proxy objects and their hardware counterparts are decoupled by the dual ported architecture of the register file. Potentially concurrent accesses to an element in the same hardware component are guarded with arbiters.

5 Hardware Implementation of Behavior

The behavior of classes and operations is implemented according to the FSMD model (finite state machine with datapath) [3]. This model is especially suitable for control oriented applications, and fits the message based computing paradigm of the object based model of computation.

Each behavior is constructed as controller with an attached datapath. The datapath performs the computations of the behavior, the evaluation of the conditions which control the datapath, and the components that store the inputs, outputs, and intermediate results. For the realization of the computation operations and buffers the resource services that have been allocated during design space exploration are used. The results of conditions are inputs of the controller. Scheduling of the datapath is performed and assigned to the behavior during

design space exploration. During implementation this schedule is then actually realized.

The controller is realized as Moore FSM. Datapath operations which require at most one clock cycle are associated with one FSM state. Multi-cycle operations are associated with a number of consecutive states. Independent operations and operations that execute at most one clock cycle may be chained to execute back to back in the same cycle.

Behavior implementations are not shared among different objects. Hence no synchronization of concurrent executions of the same behavior in different objects is required. To avoid write contention the datapath keeps local copies of attributes and inout-parameters. All modifications of the datapath are performed on the copies. If no explicit output enable is specified in the model, they are synchronized back at the end of execution and when the behavior executes a message transfer.

6 Conclusions

In this article we have presented a novel approach to the object-oriented hardware synthesis from UML-models. This article has shown that such specifications can also be realized with reconfigurable hardware in a straightforward and convenient manner. The paradigm of message based computation can be used to generate very efficient and scalable hardware implementations. The development of highly parallel applications is encouraged. Advanced model compilers for SoC can automatically implement heterogeneous multiprocessor solutions from UML models.

References

1. Thomas Beierlein, Dominik Fröhlich, and Bernd Steinbach. UML-Based Development of Applications for Run-Time Reconfigurable Architectures. In *Proceedings of the UML-SOC - International Workshop on UML for SoC-Design 2004 (UML-SoC'04)*, June 2004.
2. John. P. Elliott. *Understanding Behavioral Synthesis*. Kluwer Academic Publishers, 2000.
3. Daniel D. Gajski. *Principles of Digital Design*. Prentice Hall Inc., 1997.
4. Bernd Steinbach, Dominik Fröhlich, and Thomas Beierlein. *Languages for System Specification*, chapter UML-Based Codesign for Run-Time reconfigurable Architectures, pages 5–20. Kluwer Academic Publishers, 2004.
5. Bernd Steinbach, Dominik Fröhlich, and Thomas Beierlein. *UML for SoC Design*, chapter Hardware/Software Codesign of Reconfigurable Architectures Using UML, pages 89–116. kluwer, 2005.
6. University Irvine, Center for Embedded Computer Systems. SPARK: High-Level Synthesis using Parallelizing Compiler Techniques. `http://mesl.ucsd.edu/spark/`, April 2004.
7. Xilinx Inc. Forge - compiler for high-level language design. `http://www.xilinx.com/ise/advanced/forge.htm`, 2003.