Model-Driven Compilation of UML-Models for Reconfigurable Architectures

Thomas Beierlein[1], Dominik Fröhlich[1 2], and Bernd Steinbach[2]

[1] Hochschule Mittweida
Dept. Information Technology and Electrotechnique
Germany
{tb | dfroehli}@htwm.de
[2] Technische Universität Bergakademie Freiberg
Institut of Computer Science
Germany
steinb@informatik.tu-freiberg.de

Abstract. The development of systems comprising of hardware and software components has been a complex and demanding task. To handle the informational and architectural complexity inherent to these systems novel development approaches supported by powerful tools are taken. In this paper we present an approach that is based on Uml and the concepts of platform-based design, model-driven architecture, and co-design. We will introduce how these concepts can be used to direct the development and compilation of Uml-models into readily executable applications. Further we will present a tool that implements the concept of model-driven compilation in order to automatically compile hardware/software implementations from Uml-models.

1 Introduction

It is a recent trend in the development of embedded computer systems to integrate classical microprocessors (uP) with reconfigurable logic resources, like field programmable gate arrays (Fpga). The uP executes the global control flow and those parts of the application that would require too much effort if implemented in hardware. Calculation-intensive parts of the application or critical I/O operations are performed by the Fpgas. In run-time reconfigurable architectures (Rtr) the reconfigurable logic allows for the dynamic adaption of the implemented functionality to the executed algorithm, by reconfiguring its logic resources. Often these components are integrated on the same chip, together with sensors, analog circuitry or even micromanaging components to systems-on-chip (SoC), in order to improve performance, power dissipation, size and cost of the system.

To better cope with the steadily growing informational and architectural complexity of such systems new approaches to their development are taken. A common trend in this area is toward the application of object-oriented (OO) methodologies, languages, and tools for the development of embedded computer systems. However, considering OO in the context of embedded systems is a two edged sword. From the design point of view, the application of OO design principles may enlarge the class of problems being tractable in the reality of systems engineering. But that does not come without cost. Typical OO specifications are hard to analyze and to implement in hardware, due to their dynamic features, such as dynamic message dispatch and the utilization of dynamic data-structures. Recent advances in design theory, architectural and system-level synthesis, and reconfigurable logic offer novel options to address these problems efficiently.

Our recent research focuses on the development of Rtr-applications with the Unified Modeling Language (Uml) [11]. Of course, the academical and industrial efforts to develop embedded 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 [17]. In [13] 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 completed manually and are then refined using traditional methods. This break in the tool-chains and paradigms is still very archetypal also to most software tool-chains for Uml. Other approaches use Uml for the specification of systems with System-C [3], [18]. As with the forthcoming Uml 2.0 academical and industrial groups have started to use this language also for the complete design and specification of hardware/software systems [15].

In this paper we present an approach that allows for the complete, precise and object-oriented specification of hardware/software systems with UML 2.0 and its automated compilation into a directly executable application comprising of hardware and software modules. 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. The approach supports the hardware/software co-design of applications for RTR-architectures. The design of the application is captured as system-level UML-model. To increase reuse and re-targeting, also the employed design-, implementation-, and hardware-platforms are specified with UML. A UML model compiler, MOCCA (MOdel Compiler for re-Configurable Architectures), implements our methodology in that it automatically performs validation, platform mapping, and application synthesis. For the purpose of this paper it is important to emphasize that the compilation process is entirely driven by the models.

The rest of this paper is organized in three sections. In section 2 we give a brief overview of our methodology and introduce the main development activities and artifacts. For a detailed discussion of the development methodology we refer the interested reader to [2] and [14]. Because platforms and models are key concepts of our approach, we discuss their meaning and relationships in in section 3 thoroughly. In the core of this paper, in section 4, we focus on the model-driven compilation of object-oriented concepts to reconfigurable hardware. Finally, in section 5 this paper is concluded.

2 Development Methodology

Figure 1 shows the basic activities and artifacts of our development approach. The illustrated approach incorporates the specify-explore-refine paradigm of hardware-software co-design into the concept of Model Driven Architecture [7], [12], [10]. It is a particular goal of MDA to separate the application from the platform[3], comprising of hardware, operating systems, compilers etc., used for its implementation. This not only supports the re-targeting of the application but can also simplify the mapping of a given application to a particular platform.

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) [2]. The overall development approach is very common to the most object-oriented development efforts. What makes it different to the known approaches is its strict separation of the concerns and information into separate models. This not only eases the validation, portability, adaptability and reuse of the developed systems, but also better supports the re-targeting of the employed tools.

In the following section we discuss the employed platforms and models more deeply and give some brief examples on their content and meaning.

3 Platforms and Models

3.1 Overview

Figure 2 shows the important models and the platforms they describe and their relationships. The models below the horizontal line define the space of possible application architectures and characteristics. Above the line the models defining a concrete application, and hence representing a point in the application space are depicted.

3.2 Design Model and Design Platform Model

The design-model defines an executable and platform independent realization of the use-cases. It 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[4]. System behavior is defined with operations, state-machines, and

[3] The term *platform* is quite overloaded with meanings and definitions today. In the course of this paper we refer to the definition given in [9]: "A platform [...] is a layer of abstraction that hides the unnecessary details of the underlying implementation and yet carries enough information about the layers below to prevent design iterations."

[4] In UML active classes are classes that own a thread of control.

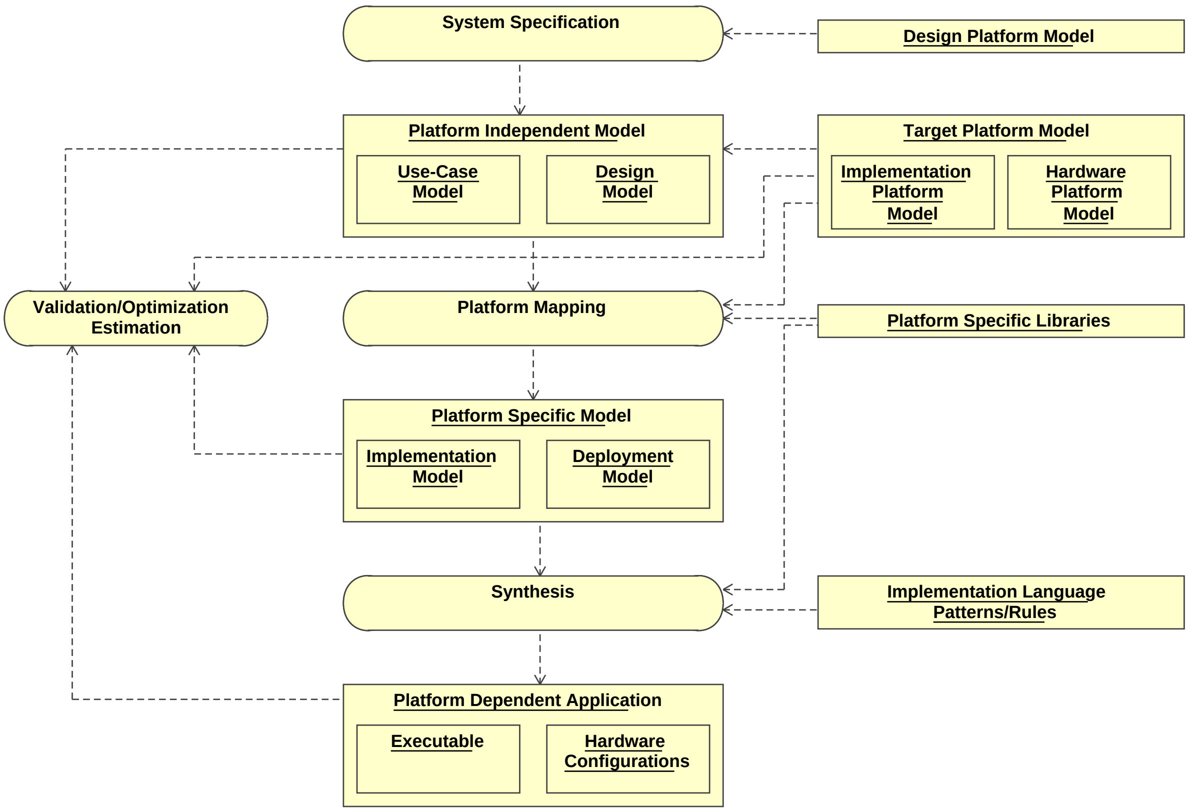


Fig. 1. Development Methodology - Activities and Artifacts

the Mocca action language. Synchronization between concurrent control flows is supported with guarded operations. The model of computation is objects communicating through structured messages.

Each design is based on a design platform. A design platform is the set of types and constraints that may be used to define user designs. A design platform reflects the basic assumptions we make when constructing systems and makes them explicit in a separate model. Such design platforms can also be found when defining systems with programming languages, for instance. However these platform are usually implicit and captured in the definition of the language and utilized libraries. Making this platform explicit is a requirement for a platform independent, retargetable approach to compilation. Design platforms can and should be shared by different designs. The concrete content and representation of a design platform depends on the targeted application domain.

Design platforms are described in various ways, for instance in language reference manuals and library descriptions. In our approach we describe the design platform as Uml-model that defines the available abstract data types (Adt). For each type we define its relationship to other types in terms of generalizations and dependencies, the supported operations, and constraints, like domains and distance vectors to

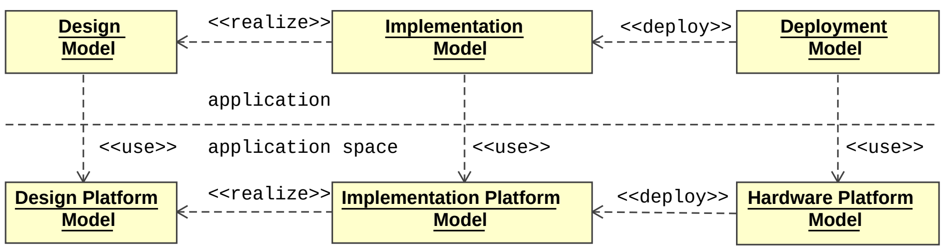


Fig. 2. Relationships between Models

other types. Please note that the definition of the basic types is in fact a must when developing systems with UML, because the few types used in the UML-spec are mainly intended for the specification of UML itself and are defined too loosely in order to be useful in real-world designs.

Example 1. Figure 3 shows a small portion of an example design platform model. It shows a number of types and their relationships. For the `boolean` type the operations defined on its instances are shown. These are the common logical operations and the allowed type casts. For the `int` the figure illustrates the constraints on this type, e.g. its domain and the distance to other types. Remarkable is also the `classifier`-type that is the base type of all user classifiers in the design models based on this platform. A design platform may contain more types, for instance for basic input/output, access to actors/sensor, and system control. Also these types can be used like native types in user designs.

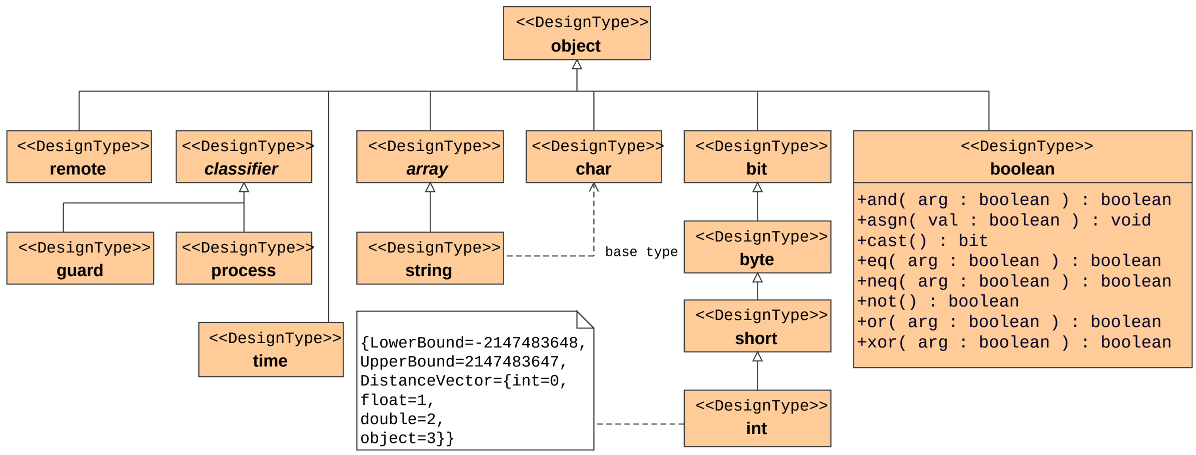


Fig. 3. Example part of a Design Platform Model

The semantic of the design types and their operations is insulated from the semantic of the action language. Therefore the action language is not fixed to this approach but may be changed specifically to the application domain. The specification of the operations is inspired by Smalltalk. The MOCCA compiler uses the design platform model for various task during the compilation process. For instance during validation of user designs, for instance for type compatibility checks and automated insertion of type casts. This compiler does only make minimum assumptions about design types, what design models are considered valid it is entirely controlled by the design platform model and the UML well-formedness rules, of course.

Example 2. In figure 4 a, rather contrived, example design based on the above design platform is illustrated. It comprises of three classes `Demo`, `MathUtilities`, and `ExtMathUtilities`. The latter two classes implement an operation `encode`, that takes an `int`-array and a key, and returns the encoded elements to the caller. `ExtMathUtilities` overrides the operation of `MathUtilities`, e.g. to implement a different algorithm. `MathUtilities` additionally implements an operation `reverse` that bit-reverses the value of its input parameter. We will use this as a running example, to show how to compile the design for a RTR-architecture.

3.3 Implementation Model and Implementation Platform Model

The implementation model defines how the design model is realized in terms of implementation classes, components, and artifacts. This model is created from the design model during the platform mapping phase, either manually, by the user, or (semi-) automatically by MOCCA. As the design model, the

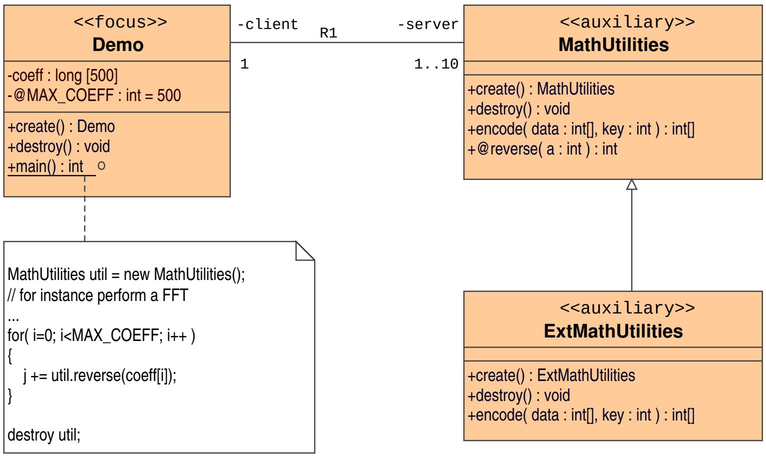


Fig. 4. Example Design Model

implementation model is based on an implementation platform, which is defined in a separate model. We define an implementation platform to be the set of types, type mappings, constraints and tools that may be used for the realization of user designs. Implementation platforms define all necessary information for high-level platform mapping, estimation, and synthesis. It is subject of this model to provide efficient implementations for the design types of the design platform and model their integration in the compilation environment, in oder to allow for automatic generation of directly synthesizable and executable hardware and software modules.

In general we define a implementation platform on the basis of abstract instruction sets and execution characteristics. The term instruction may be the native instruction of microprocessors, the operations directly implementable in hardware, but also high-level operations of programming languages. For each processing element in the hardware platform an implementation platform is modeled, however an implementation platform model can be shared by processing elements. To illustrate the content of implementation platform models, we introduce it with two examples for a C/C++ platform.

Example 3. Figure 5 shows a part of an implementation platform model. The example depicts some of the design types from figure 3 and the implementation types being used for the realization of the design types. Because we can allow for stacked and nested implementation platforms, we can also find such type mappings between different implementation types. During automated platform mapping MOCCA creates one or more implementation types for the types of the user design not belonging to the design platform and establishes realization-dependencies between them.

The implementation types and the operations they implement are precharacterized by their quality-of-service (QoS). The QoS-characteristic of the type itself is an area value that defines the memory footprint of the instances of this type, or the number of gates in case the implementation platform is deployed on reconfigurable hardware. The QoS of operations implemented by the types, may define the latency, power dissipation, or even abstract cost. Also the types and their operations specify information to control the utilized generator, like name-mappings and implementation language patterns for the utilization of primitive operators etc. It is important to emphasize that the straightforward definition of the type-mappings was made possible by the explicit formulation of the design platform.

The QoS-characteristics as well as the control information for the hardware/software generation is represented with the common UML extension mechanisms. The format of this information depends on the concrete requirements of the generator, mappers, and estimators of the implementation platform. In order to allow for a flexible and straightforward re-targeting the according MOCCA components and their relationships are also defined in the implementation platform model.

Example 4. Figure 6 continues the specification of the implementation platform model in figure 5. In this part of the model the MOCCA-compiler components for the C/C++ implementation platform are specified.

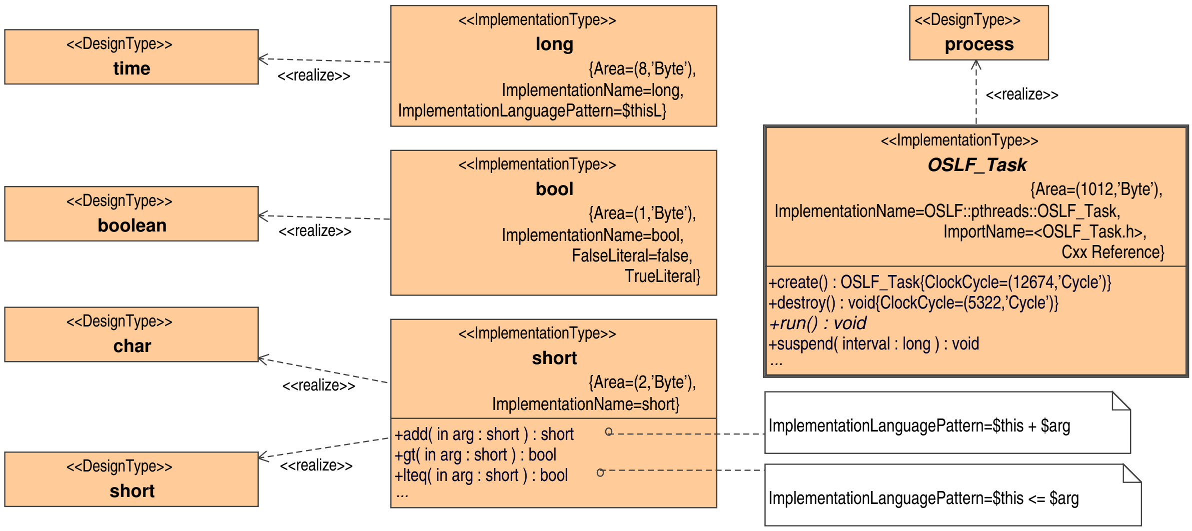


Fig. 5. Example Implementation Platform Model - Type Mappings

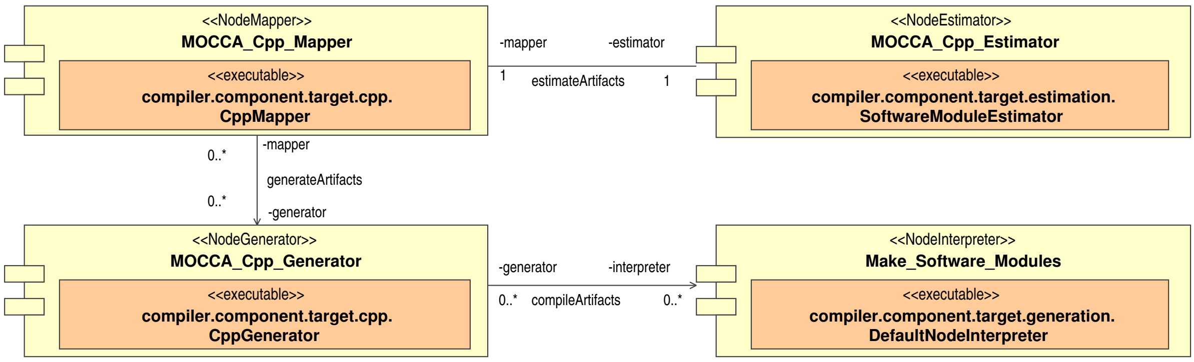


Fig. 6. Example Implementation Platform Model - Compiler Components

It is important to note, that the component specification is used by Mocca in order to adapt itself to the concrete target platform. During the compilation Mocca dynamically links modeled implementation platform specific components into its extensible architecture. Users may implement new compiler (back-end) components on their own, or specialize existing components, in order to adapt the compiler to their concrete requirements. To link the compiler into existing tool-chains an implementation platform model may also specify interpreter-components. These can be used for instance to invoke programming language compilers, logic synthesis tools etc. transparently during the compilation process.

3.4 Deployment Model and Hardware Platform Model

The implementation model of a system defines the realization of the design model, and the same applies for the implementation platform model. The actual deployment of the implementation model components on the nodes of a target hardware architecture is defined by the deployment model. In this model deployment relationships between (processing-) nodes and the artifacts manifesting the system (e.g. executables, hardware configurations, libraries, tables etc.) are defined. Again, this model may be defined (semi-) manually or entirely automated by Mocca.

The deployment model is defined in terms of an underlying hardware platform, that is defined in a hardware platform model. We define the hardware platform model as the set of abstractions of hardware nodes, their communication channels and constraints that may be used for deployment and execution of design (platform) model implementations.

Example 5. This example completes the implementation and deployment specification of the design in example 2. Figure 7 shows the implementation and deployment of the design on a given hardware architecture. The classes `MathUtilities` and `ExtMathUtilities` realize a common component `MathComponent`. This component is manifested by a reconfiguration bitstream for the FPGA. The main-class `Demo` is deployed by means of an executable on the system master node. The figure also illustrates the relationship between implementation platforms and processing elements. One implementation platform is the C/C++ platform described above, the other is a platform based on the SPARK-HLS compiler [16] and a specialized compilation pass generating synthesizable VHDL for the communication interface, register-files and synchronization.

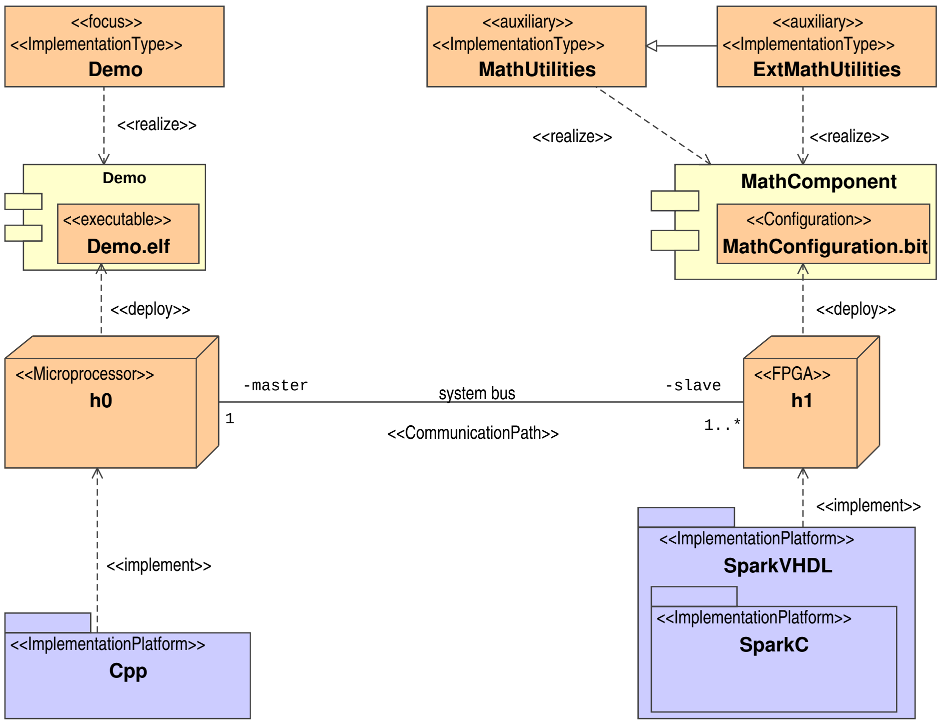


Fig. 7. Example Implementation Model and Deployment Model for Design Example

As can be seen from the example we do not specify the micro-architecture of the hardware itself. Instead we characterize the hardware components and the resource services they provide sufficiently well in order to enable a fast design space exploration based on sufficient estimates of the implementation characteristic of the explored implementations.

4 Implementing UML-Models with reconfigurable Hardware

4.1 Overview

In section 2 we briefly presented our development methodology and in the previous section we presented the platforms and models on which our approach is based. We have shown the transformation of the user design model 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 the generation of platform dependent implementations. Due to the lack of space we will concentrate on the generation of configurations for reconfigurable hardware.

When considering the direct implementation of UML-models with hardware one must first determine the model-elements required to implement. 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 three problem areas:

– Dynamic creation/destruction of hardware objects.
– Implementation of objects with polymorphic features.
– Communication between software- and hardware objects.

In what follows we will discuss these issues and propose solutions to them.

4.2 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. For this we analyze the global message exchange of classes and their object creation/destruction characteristics. For a description of a previous variant of the clustering algorithm, based on use-cases, we refer the reader to [6]. The algorithm tries to identify groups of closely cooperating classes and behaviors that are critical to overall performance and can be implemented together in hardware. Moreover we are trying to estimate the number of concurrently required instances from the profiles. To accomplish this, we developed a profiling technique to estimate the execution probabilities and frequencies directly in the Uml-models [4].

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[5]. The actual creation/destruction of the hardware objects is delegated to a run-time service, called Rtr-manager. The Rtr-manager is responsible for the administration and brokerage of the reconfigurable hardware and hardware objects present in the system [8]. 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. Otherwise it tries to reconfigure the hardware with an appropriate configuration first. Our research has shown that this approach offers a significant performance advantage.

Figure 8 shows an architecture template for the implementation of hardware configurations with multiple objects. Each hardware configuration contains a communication interface, a register file, and a number of hardware objects. The communication interfaces provides access to the external communication channel. This channel may be everything from a on-chip bus to a serial line. The register-file comprises the interface of all objects in a configuration. To improve performance the data communicated through the interface is buffered. The register-file can be accessed over the communication channel and from the hardware objects, but each object should only have access to its own slice of the register file.

4.3 Implementation of Hardware Objects with polymorphic Operations

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[6]. The public interface of an object comprises of all features modeled as public. Public and non-public features can be implemented equally, except that the only the public features are accessible via the communication interface.

[5] Note, that the notion of classes as object templates is one of several interpretations of class.
[6] We do not allow for multiple inheritance yet, that is each class may only specialize one super-class.

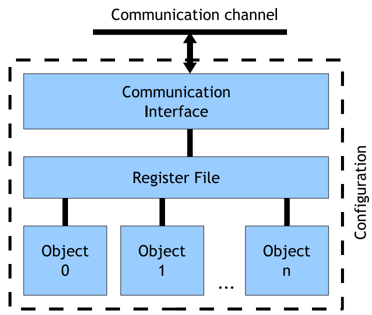


Fig. 8. Hardware Configuration Architecture Template

The implementation of attributes is straightforward, they are mapped to a register of an appropriate width in the register-file. The logic synthesis of the methods of operations[7] is delegated to the Spark-Hls, which is transparently integrated in the compilation flow by means of the implementation platform model as presented in section 3.3. Synchronization between the software and hardware-objects is accomplished by a simple handshake protocol. Operations may have a GO/DONE pair of control bits. By triggering GO the operation is started, the end of execution is signaled with the DONE-bit. The GO/DONE-bits are accessible via control-registers in the register file. These control line only have to be generated for operations, that

– modify attributes, or
– require multiple clock-cycles for execution, or
– override an operation that requires a GO/DONE in case of polymorphically implemented objects.

An operation is executed by writing its input parameters to the appropriate registers, eventually triggering its GO-signal, waiting for the DONE-signal, and reading the output and return parameters.

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 [5]. 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 [8]. 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 reduced in certain situations.

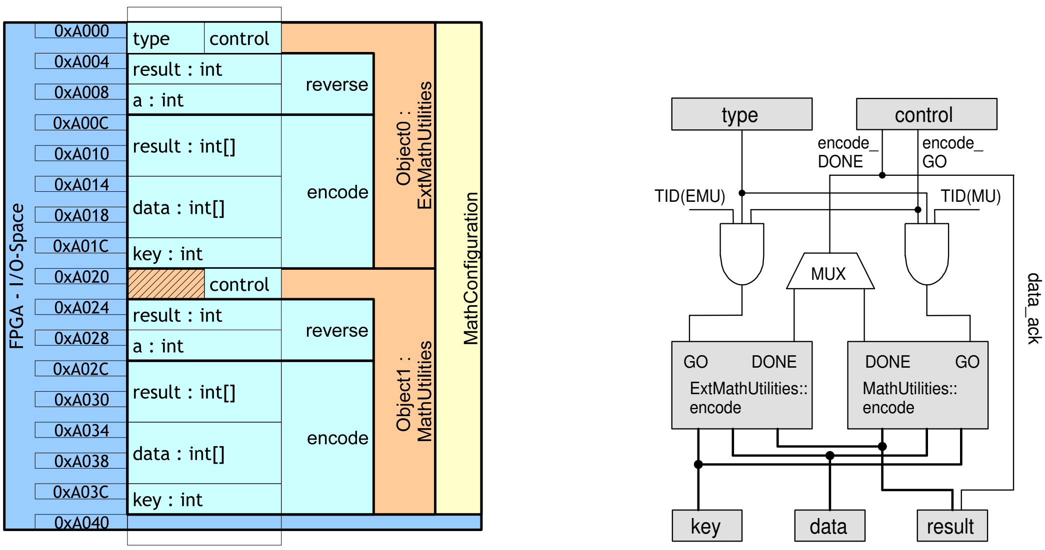
Example 6. Assume that in figure 7 we pre-instantiate one object for each of the classes `MathUtilities` and `ExtMathUtilities` in configuration `MathConfiguration`. Lets further assume that at run-time the object of class `MathUtilities` is allocated (created) and that the software requests the Rtr-manager to create another object of this class. In a flat implementation we would have either to reconfigure an unused Fpga or to reject this request. If we allowed for inheritance, we could, however, allocate the object of class `ExtMathUtilities`, switch off the features specific to `ExtMathUtilities` and enable the features of its super-class instead. On our test hardware platform (Windows PC 2.5GHz with Xilinx XC2V3000 Fpga Pci-board) we have reached speedups of up to 10000 in this situation [8]. On embedded devices with slower microprocessors this value decreases because the hardware object management takes longer, but the speedup is still significant.

Of course, the full implementation of inheritance is only advantageous in a given hierarchy, if the classes in the hierarchy are actually instantiated at run-time. The set of classes being created is computed at compilation time with an optimized version of the well-known Rta-algorithm (Rapid-Type-Analysis) [1].

[7] In Uml a method is defined as an implementation of the behavior of an operation.

The interface of hardware objects must be independent from the concrete object template and the dynamic object type. 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 registers and GO/DONE-bits.

Example 7. Figure 9 shows the implementation of objects with polymorphic features for our example design in figure 7. Again, we assume that configuration `MathConfiguration` contains two objects, one of class `MathUtilities` and one of class `ExtMathUtilities` respectively. One possible register layout that fulfills the requirement of compatible interfaces is shown in sub-figure 9(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 `encode`, but allows for switching its dynamic type (and hence the actually executed implementation) to `MathUtilities`, by means of a type-ID (`TID`) written to the type register. This type-ID is automatically assigned by Mocca. The according logic is illustrated in sub-figure 9(b). `Object1` does not require a type-register, but still has to preserve the register layout. For the purpose of this example we assume that the application does not require random access to the arrays in the interface of the `encode` operations, such that they may be implemented as Fifo. For the lack of space we stripped the interfaces of the object constructors and destructors.



(a) Object-Interface

(b) Implementation of `encode`

Fig. 9. Example Implementation of objects with polymorphic operation `encode`

4.4 Communication between Hardware and Software-Objects

The synthesis of communicatin interfaces and message exchange between different hardware nodes is an important problem in the most co-design approaches. For this a detailed specification of the hardware architecture, available communication channels and protocols is required. As shown in the section 3.4 we specify the hardware architecture and communication channels in the hardware platform model. Due to the previously restricted capabilities of Uml for detailed hardware modeling we do, however, not specify the communication protocols in Uml[8].

Instead we model the elements that implement these protocols by means of implementation types in the implementation platform model. This allows to use only abstractions during the generation of the hardware- and software modules. Additionally, in the hardware architecture model the relationship

[8] The capabilities for detailed modeling of hardware and protocols have been strengthened in the forthcoming Uml 2.0. This revision improves the modeling of state and behavior, and now supports timing diagrams.

between the communication protocol implementations and the communication channels on which they are used ist modeled.

The abstractions are replaced during the back-end logic synthesis and compilation by their actual implementations. For this, the modeled abstractions must be implemented once with the used programming languages or hardware description languages. The actual hardware and software generation is controlled by the code generation information appended to the abstractions. The same technique is used to support platform mapping and estimation.

This approach allows us not only to model communication interfaces, but all kinds of supplementary elements required to implement user designs on a given implementation platform. For instance on hardware platforms we may require special clock-generators, storage elements, or even arithmetical/logic units. Instead of coding this into the compiler we model these elements explicitly. Each compiler components can then use the information carried by the according model elements to perform their specific task. This enables the implementation of widely generic and parametrizable hardware and software generators, mappers, and estimators.

Example 8. Figure 10 shows the abstraction of a synchronous, dual-ported 8-bit register as used, for instance in the register files. This register may be implemented in VHDL, which is indicated to the model-compiler with stereotypes. Both types model the interface of the register with operations or attributes (not shown). Modeling the interface with operations supports comprehension, because it supports a natural grouping of port variables according to their semantics. Because UML does not make any assumptions about mechanisms for message exchange and signaling the abstraction with operations is UML conformand. The example shows how the QoS of the element is modelled, for the `setClock`-operation the generator control information is illustrated. The modeled usage-dependencies are transformed into according `library` and `use` declarations by the VHDL-generator.

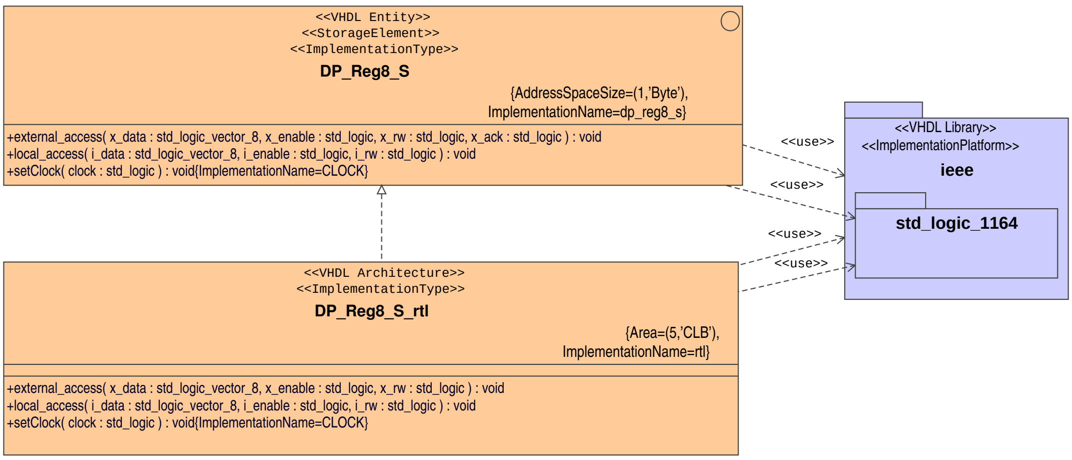


Fig. 10. Example Specification of an 8-Bit synchronous, dual-ported register

For the realization of the actual message exchange a similar approach is used. The mechanism for the messaging between objects located on the same hardware node is realized by the generators and back-end synthesis/compilation tools. From the mechanisms used for the message exchange between remote objects (objects not located on the same node) we abstract by means of a remote-type (see figure 3).

An instance of remote on a given node serves as a proxy for an object located on a different node. This type defines a common interface that provides basic services for the creation/access/destruction of remote objects. Each implementation platform may define a specialized impementation of remote. Thus, the supported communication types and direction are not captured in the compiler but the models. The remote type may be specialized, for instance to provide sophisticated object (un-)marshalling functionality.

Example 9. In this example we illustrate the automatically code generated for the message exchange between instances of our design classes `Demo` and `MathUtilities`. As assumed in figure 7, the instances of Demo may be executed on a microprocessor and are implemented on a C++- platform. Instances of MathUtilities may be executed on a FPGA and are implemented on a VHDL-platform. Figure 11 shows the generated C++-code for the instantiation of the remote-object, the access of its reverse operation, and the final destruction of the hardware object. The design model type remote is implemented on the C++-platform by the type IHwObject. The creation and destruction is delegated to the RTR-manager. The classes are identified by a unique type-ID, which is automatically generated by MOCCA. The read/write-operations take as first parameter a relative address of the register in the object interface.

```
int Demo::main( void ) {
   smartptr<IHwObject> mutil0;
   int i;
   ...
   mutil0 = RTRManager::getInstance()->createObject( 0 );
   if( mutil0.getPtr() != NULL ) /* auto check of instance Data.Demo.main.304.mutil0 */
     mutil0->execute<char>( 4, 1 ); /* execute Constructor MathUtilities::create()  */
   if( mutil0.getPtr() != NULL ) {
      i = 0;
      for(i = 0; i < Demo::MAX_COEFF; i = i + 1) {
         mutil0->write<int>(36, Demo::coeff[ i ] );
         Demo::coeff[ i ] = mutil0->read<int>( 32 );
      }
      if( mutil0.getPtr() != NULL ) /* auto check of instance Data.Demo.main.304.mutil0 */
              mutil0->execute<char>( 4, 4 ); /* execute Destructor MathUtilities::destroy() */
      RTRManager::getInstance()->destroyObject( &mutil0 );
   } ...
}
```

Fig. 11. Example of generated C++-code for remote object creation, access, and destruction

The abstraction with the remote-type is quite useful and generic. For instance it allows also for implementations that support the allocation of objects through networks. We are currently working on according implementations of IHwObject and the RTR-manager.

5 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 platforms and their UML-models. The platforms are used throughout the each phase of development, from design to the final implementation and deployment. We have shown that platform models serve two main purposes. First, carefully developed platform models make the basic assumptions and restrictions of a dedicated platform explicit to the developer. This may improve the quality, comprehensibilty, and portability of system developed with this approach. Second, platform models are a very useful tool to direct the automated transformation of system specifications into final products. Dedicated model compilers use platform models to perform automated validation, platform mapping, synthesis, and optimizations on user models in order to create directly executable applications from system-level specifications.

As an example for such a model-compiler we have presented MOCCA. This compiler implements the presented approach. Currently we are validating and evaluating our development approach on several real-world designs. This includes the development of performance critical applications of embedded systems, like encoders/decoders for multimedia-streams. Another application, less related to resource constrained embedded systems, is the development of co-processors for Boolean problem solvers and neural network applications. Other applications may be the rapid-prototyping of software 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.

If accepted for presentation we could contribute a short practical presentation of our approach and the MOCCA-compiler. Currently we are working on the QoS-benchmarking of our development environment to improve the results of automated platform mapping and estimation. This information will also allow us to obtain information on the quality of the generated results.

References

1. David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, December 1997.
2. Thomas Beierlein, Dominik Fröhlich, and Bernd Steinbach. UML-Based Co-Design of Reconfigurable Architectures. In *Proceedings of the Forum on Specification and Design Languages (FDL'03)*, Frankfurt a.M., Germany, September 2003.
3. Pierre Boulet, Jean-Luc Dekeyser, Cedric Dumoulin, and Phillipe Marque. MDA for SoC Design, An Intensive Signal Processing Experiment. In *Proceedings of the Forum on Specification and Design Languages (FDL'03)*, Frankfurt a.M., Germany, September 2003.
4. Isabel Drost. Estimation of Execution Probabilities and Frequencies of OO-Models. Diploma Thesis, 2003. University of Applied Sciences Mittweida, Germany.
5. Anton Eliens. *Principles of Object-Oriented Software Development*. Addison Wesley Longman Publishers, 2000.
6. Dominik Fröhlich. Use-Case Driven Partitioning of Dynamically Reconfigurable Systems. In *International Workshop on Boolean Problems (IWSBP'02)*. University of Technology and Mining Freiberg, Germany, 2002.
7. Daniel D. Gajski and Frank Vahid. Specification and Design of Embedded Hardware-Software Systems. *IEEE Design and Test of Computers*, pages 53–66, Spring 1995.
8. Henning Riedel. Design and Implementation of a Run-Time Environment for RTR-Systems. Diploma Thesis, January 2004. University of Applied Sciences Mittweida, Germany.
9. Benjamin Horowitz, Judith Liebman, Cedric Ma, T. John Koo, Alberto Sangiovanni-Vincentelli, and S. Shankar Sastry. Platform-Based Embedded Software Design and System Integration for Autonomous Vehicles. In *Proceedings of the IEEE*, volume 91. IEEE, January 2003.
10. Stephen J. Mellor and Marc J. Balcer. *Executable UML - A Foundation for Model Driven Architecture*. Addison Wesley Longman Publishers, May 2002.
11. Object Management Group. OMG Unified Modelling Language Specification (Super Structure). `http://www.omg.org`, July 2003. Version 2.0.
12. Object Management Group - Architecture Board ORMSC. Model driven architecture - a technical perspective (mda). `http://www.omg.org`, June 2001. Draft.
13. Carsten Schulz-Key, Tommy Kuhn, and Wolfgang Rosenstiel. A Framework For System-Level Partitioning of Object-Oriented Specifications. In *Proceedings of the Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI'01)*, 2001.
14. Mocca Project. The Mocca-compiler for run-time reconfigurable architectures. Web-Pages, January 2004. `http://www.htwm.de/lec/mocca` (under construction).
15. UML-SOC - International Workshop on UML for SoC-Design. `http://www.c-lab.de/uml-soc`, June 2004. San Diego, sponsored by DAC.
16. University Irvine, Center for Embedded Computer Systems. SPARK: High-Level Synthesis using Parallelizing Compiler Techniques. `http://mesl.ucsd.edu/spark/`, April 2004.
17. Wilkie, King, Clark, and Weaver. *The Action Language Specification Reference Guide*. Kennedy Carter Ltd., December 1996. `http://www.kc.co.uk`.
18. Winson Zhu Yongxin. Synthesizable SystemC Code from UML-Model. In *International Workshop on UML for SoC-Design*, June 2004. `http://www.comp.nus.edu.sg/zhuyx/usoc04.pdf`.