Chapter 1

Hardware/Software Codesign of Reconfigurable Architectures Using UML

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

[1] *Technische Universität Bergakademie Freiberg*
Institute of Computer Science
Freiberg, Germany

[2] *Hochschule Mittweida (FH) - University of Applied Sciences*
Institute of Automation Technology
Mittweida, Germany

Abstract The development of systems comprising hardware and software components has been a demanding and complex problem. To manage the informational and architectural complexity inherent to these systems novel approaches are taken. In this chapter we present an approach that is based on the concepts of model driven architecture, platform based design, and hardware/software codesign.

1. Introduction

Reconfigurable architectures are a relatively novel means of constructing computer systems. These architectures comprise one or more microprocessors and reconfigurable logic resources. The microprocessors execute the global control flow and those parts of the application that are uncritical to the overall performance. The logic resources act as coprocessors and execute the performance-critical algorithms of the system or specialized input/output operations. In *runtime reconfigurable architectures* (RTR) the logic resources are reconfigurable while the system is in operation. RTR systems feature the dynamic adaption of the functionality executed by the coprocessors to the current requirements of the application.

The main applications of reconfigurable architectures are *system on chip* (SoC) and high performance computing. Reconfigurable systems enable the acceleration of the overall system whilst cutting development and manufacturing costs. In contrast to classical ASIC approaches the functionality implemented in hardware is not fixed, it may change even while the system is already deployed. This makes them an enabling technology for SoC prototyping and evolving systems.

The development of applications of reconfigurable architectures is a very complex and error-prone task. The most current development approaches, which are based on programming languages or mixed languages, are insufficient owing to their strong focus on implementation and the inherent technology dependence [1]. Thus novel directions must be taken. In this chapter we present a development approach that 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. The key concepts of object-orientation — inheritance, polymorphism, encapsulation, and information hiding — are preserved down to the implementation level. Thus the archetypal break in paradigms, languages, and tools of current object oriented hardware development efforts is avoided.

The full automation of the implementation process supports early verification and simulation. The time required for the implementation of a system level specification is cut down by orders of magnitude. As a result, the focus is shifted from implementation towards modeling. This offers tremendous gains in productivity and quality. The synthesized hardware and software modules fit together by definition because the compiler has automatically implemented them. The change of the system design, the algorithms, the partitioning, and the implementation platform is encouraged.

The rest of this chapter is structured as follows. In Section 1.2 we introduce our development approach. We describe the used models, artifacts, and the utilization of UML and the action language. In Section 1.3 we define how the UML system design models are mapped into implementations. The focus of this chapter is on the mapping of implementation models into hardware/software implementations, which is described in detail in Section 1.4. The approach is illustrated by AudioPaK an encoder/decoder for the lossless compression

of audio streams [6]. In Section 1.5 experimental results for the example are presented and then this chapter is concluded.

2. A Platform Model Driven Development Approach

2.1 Overview

In this section a brief overview of the general development approach will be given. Because of the lack of space we focus on the key concepts and artifacts. The methodology is not described here, a thorough discussion can be found in [1][13].

The development of applications is based on platforms, whereas different platforms are used for design, implementation, and deployment. As a result, a strict separation of development concerns is accomplished. Moreover, this approach eases validation, portability, adaptability, and reuse. Platforms have been used ever since in the areas of software and hardware development. However, platforms are mostly captured implicitly in language reference manuals, libraries, and tools, which hampers their automated interpretation by computers.

Platforms represent sets of assumptions, which are the foundation of any development effort. In our approach, these assumptions are made explicit by platform models, whereas each platform is specified by a dedicated platform model. Platform models abstract from the details of the platform described, but carry enough information to avoid iterations in the design flow. They are the basis for the definition of application models that describe a certain aspect of the system under development. The relationship between the platform models and application models is illustrated in Figure 1.1. The platform models define the space of applications which may be developed with the respective platforms. Each particular set of application specific models represent one point in the application space. Different platform models normally share application models.

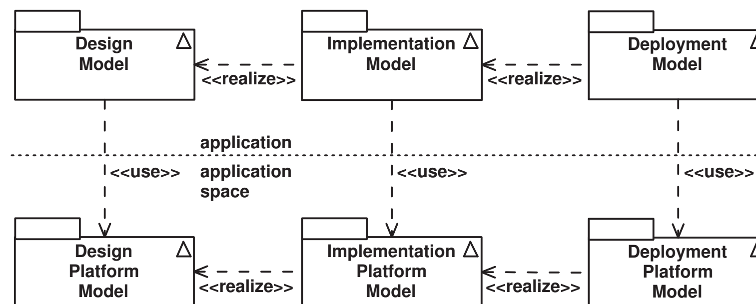

Figure 1.1. Relationships between Models

All models are described using UML 2.0 [10] and a dedicated action language called MAL (MOCCA Action Language). MAL was developed to enable the detailed specification of behavior in UML models in computation intensive and control intensive applications. This language is compliant to the UML action semantic specification. It has a medium level of abstraction because it requires the developer to make data organization and data access explicit. However, this allows us to employ standard analysis, optimization, and estimation techniques [1].

In the following sections we discuss the employed platforms and models and give some brief examples of their content and meaning.

2.2 Design Model and Design Platform Model

A *design model* defines an executable and implementation independent realization of the use cases of a system. This model defines the structure and behavior of the system. System structure is defined by UML packages, classes and interfaces and their various relationships. For concurrency specification active classes are supported. System behavior is defined by operations and state machines. Detailed behavior is defined by UML actions, whereas MAL is used as action language. Concurrent control flows are synchronized by guarded operations.

Each design model is based on a design platform which is specified by a *design platform model*. The content of a design platform model depends on the targeted application domain. It specifies the types, constraints, and relationships used for system design. For each type the relationship to other types in terms of generalizations and dependencies, the supported operations, and constraints are defined. The definition of the basic types is mandatory when developing systems using UML because the types defined by the UML specification are defined too loosely in order to be useful in real world designs.

EXAMPLE 1.1 *Figure 1.2 shows a small part of a design platform model. The example illustrates some design types whicht may be used in design models that are based on this platform model. For the* boolean *type the operations are shown. The operations represent the common logical operations and type casts one would expect. Constraints are exemplified by the* int *type. Design platform models contain typically additional types, e.g., for basic input/output, access to actors/sensor, and system control.*

It is important to note that in this definition the design platform model is not specific to a concrete action language. Model compilers use this model for the purpose of validating and optimizing the design model. Such compilers make only minimum assumptions about types. The validity of a design model is determined entirely by the design platform model and the UML well formedness rules. Designers may add new types and operations to the design platform model

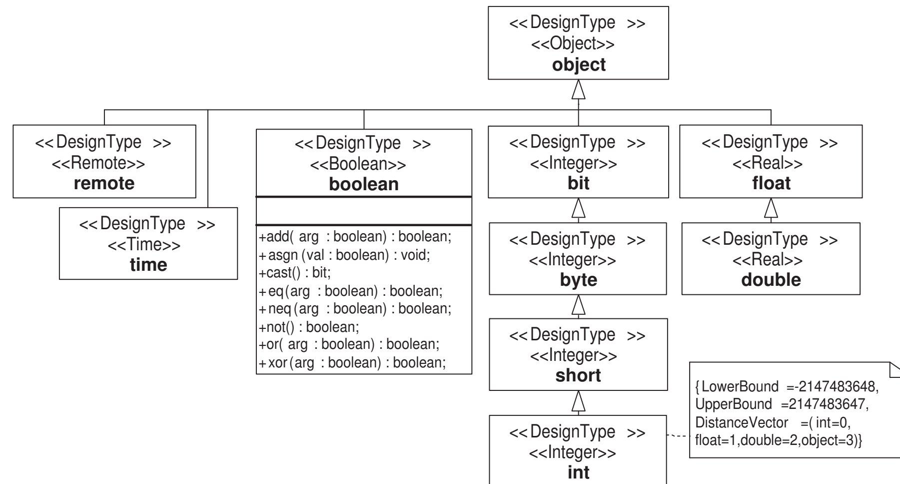

Figure 1.2. Design Platform Model Example

which are treated by model compilers as primitive types. For these elements the designer may then provide optimized implementations in the implementation platform.

EXAMPLE 1.2 *Figure 1.3 shows the design model of an AudioPaKCoder [6]. AudioPaK is a well known algorithm for lossless compression of audio information. In the example the class* `Main` *instantiates a number of AudioPaKCoder objects, which encode frames of audio samples, whereas each sample is represented by a 16 Bit integer. The frames are written to coder objects which concurrently encode them while* `main` *performs other tasks, such as input/output, filtering, etc.. The example shows the intra channel decorrelation algorithm performed by the operation* `encode`*.*

2.3 Implementation Model and Implementation Platform Model

An *implementation model* defines the realization of a design model in terms of implementation classes, components, artifacts and relationships. This model has the same functionality as the design model; however, it typically realizes this functionality differently. Implementation models define how structure and behavior are realized with the services provided by the implementation platform. There are many implementation models for a given design model. An implementation model is derived from a design model by applying a sequence of

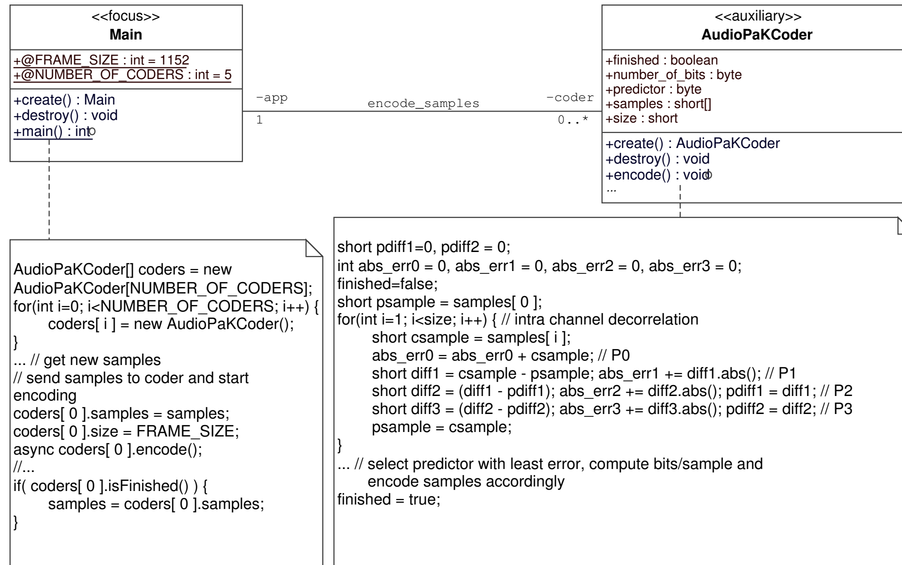

Figure 1.3. Design Model Example: AudioPaK Coder

transformations and mappings (see Section 1.3.4). The implementation model is created manually, or (semi-) automatically by model compilers.

Each implementation model is based on a specific implementation platform. Implementation platforms define the realization of design platforms, whereas each implementation platform realizes one design platform. Each implementation platform is specified by an *implementation platform model*. For each processing element in the hardware platform an implementation platform model is defined.

An implementation platform model is the set of types, constraints, transformations, and tools that may be used for the realization of design models. As design platforms, implementation platforms are defined on the basis of abstract instruction sets[1] and their execution characteristics. This model is used by model compilers to perform high level design space exploration (platform mapping), estimation, and synthesis.

EXAMPLE 1.3 *Figure 1.4 shows a part of an implementation platform model for a HDL implementation platform. The diagram depicts design types from Example 1.1 and the implementation types used for their realization. The im-*

[1] An instruction may be the native instruction of microprocessors, the operations directly implementable in hardware, but also high level operations of programming languages.

plementation types and operations are characterized by quality of service and generation information.

Figure 1.4. Implementation Platform Model: Types and Mappings

EXAMPLE 1.4 *Figure 1.5 continues the presentation of the VHDL implementation platform model. Implementation platforms may contain pre-implemented hardware/software modules. Model compilers can integrate such cores in the generated modules. The example shows a clock generator component, a PCI bridge, and a storage component that may be used in hardware designs. The components specify a number of interfaces for clocking, local/external access and interconnections, which are specified in detail by interfaces and classes. The components are implemented in VHDL, the UML model just specifies the necessary information to integrate them into hardware modules.*

The implementation components, types, and operations are characterized by their quality of service (QoS). As shown in Figure 1.4, the QoS characteristic of a type is an area value which defines the memory footprint or the number of gates of its instances. The QoS of an operation defines its area, latency, power dissipation or even abstract cost. Moreover, the elements of the implementation platform model may specify information to control the generator of a model compiler. Name and type mappings and implementation language patterns are examples for such information.

The syntax and semantics of the UML extensions which are used for design space exploration, estimation, generation, and synthesis are modeled by UML profiles. Profiles are UML models which define coherent sets of extensions of

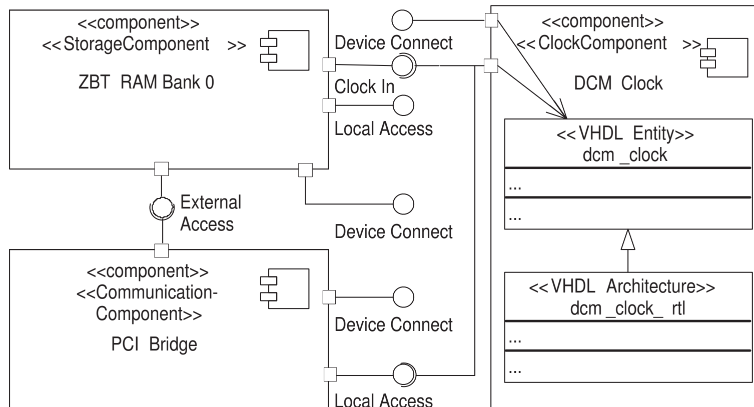

Figure 1.5. Implementation Platform Model: IP Integration

UML. Such extensions are commonly specific to application domains, implementation platforms, backend tools and configurations. There is a common set of extensions that is used in most implementation platforms, but each platform has its own set of additional extensions. Thus implementation platform profiles are defined hierarchically.

UML extensions can be interpreted by users and model compilers. In order to avoid design iterations, implementation models and implementation platform models must reflect the characteristics of the compiled/synthesized hardware/software artifacts as close as possible. Thus it is important to give the model compiler control over the implementation process. Owing to the huge variety of implementation platforms a model compiler for SoC should be able to adapt to the set of platforms being used. To make this adaption convenient and straightforward, the respective components of the model compiler are modeled in the implementation platform model.

EXAMPLE 1.5 *The approach to model the modeling compiler components in the implementation platform model is exemplified for the MOCCA compiler in Figure 1.6. In this part of the model the MOCCA components used for estimation, mapping, generation, and backend tools are specified. The component specification is used by MOCCA to adapt to the implementation platform. During the compilation these components are dynamically linked into the compiler. Users may implement new compiler components on their own, or specialize existing components to adapt the compiler to their concrete requirements.*

EXAMPLE 1.6 *From now on it is assumed that the design model class* `Audio-PaKCoder` *is realized by the VHDL implementation platform. The respective implementation model class and its relationship to the design model class is*

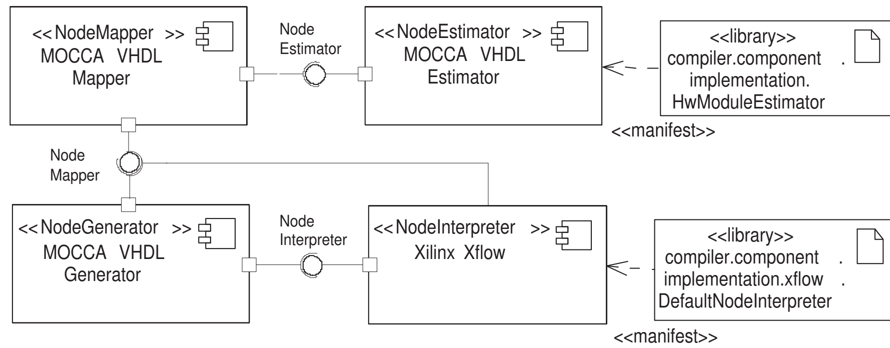

Figure 1.6. Implementation Platform Model: Compiler Components

illustrated in Figure 1.7. The `encode` *operation can not be implemented directly in hardware; the mechanism and protocols to access the sample array must be made explicit. A simple WISHBONE-like bus interface is used in the example [11]. The behavior of* `encode` *is transformed respectively to access the samples through this interface. This model will be refined in the course of this chapter.*

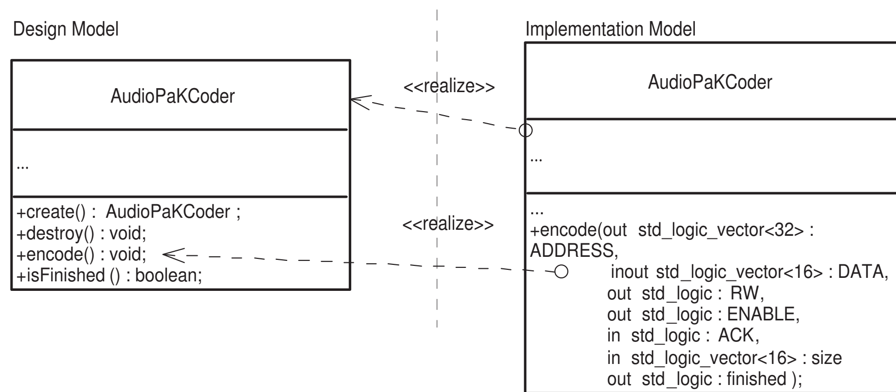

Figure 1.7. Implementation Model: AudioPaKCoder Mapping

2.4 Deployment Model and Hardware Platform Model

A *deployment model* defines the deployment of an implementation model of the application on a target hardware architecture. This model defines the

deployment relationship between the nodes of a hardware architecture and the artifacts which manifest the components of the implementation model. As a result, the deployment model fixes the execution of the implementation model. Examples of nodes are microprocessors, reconfigurable logic devices, or abstract execution platforms. In accordance with the UML specification a node may comprise a processing element (PE), dedicated memory, and peripherals. Common artifacts are executables, logic configurations, libraries, and tables. The deployment model is created manually, or (semi-) automatically by model compilers.

Each deployment model is based on a hardware platform. Hardware platforms define how implementation platforms may be realized. A hardware platform may realize multiple implementation platforms and an implementation platform may be realized by different hardware platforms. Hardware platforms are specified by *hardware platform models*.

A hardware platform model defines the nodes, communication paths, and constraints of a hardware architecture. Hardware platforms commonly do not specify the micro-architecture of hardware nodes; they define the services provided by the hardware resources. For instance, the number of logic and memory resources, clock rate ranges, scheduling policies, communication protocols are specified by constraints. The hardware platform model must contain just enough information to enable high quality design space exploration. The constraint representation is similar to the QoS in the implementation platform. This information of the hardware platform model is used to parameterize implementation platforms.

EXAMPLE 1.7 *In Figure 1.8 a portion of a hardware platform model and a deployment model based on this hardware platform is illustrated. The hardware platform consists of two nodes* `h0` *and* `h1`*, which are connected by a communication path. Artifacts being deployed on the nodes are implemented by a dedicated implementation platform. The artifact* `audiopak.exe` *is an executable program for* `h0`*. It manifests a component that realizes the* `Main` *class. The audio coders are implemented by a component that represents a configuration bitstream of node* `h1`*. This is made explicit by the according stereotypes.*

The deployment and implementation related models complement each other. The deployment platform model and implementation platform model are referred to as *target platform model*. The implementation model and deployment model are subsumed in the *platform specific model* [1]. A design model is implemented on a new target platform by using an according target platform model. The design model and design platform model are not required to change.

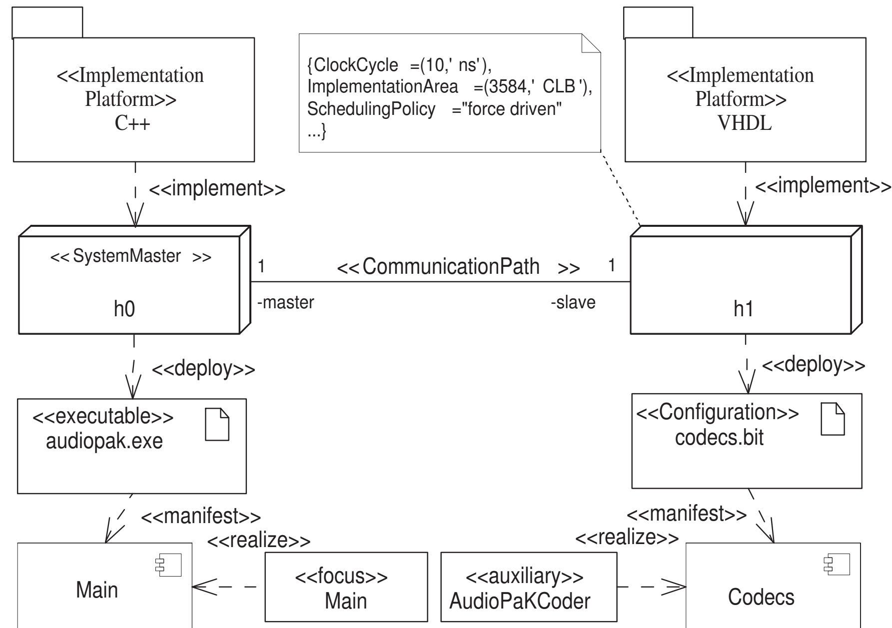

Figure 1.8. Deployment Model

3. Mapping Design Models onto Implementation Models

3.1 Hardware Architecture

The mapping of a design model to an implementation model depends on the physical and logical system architecture. The physical system architecture is determined by the architecture of the underlying hardware. The development approach targets SoCs with heterogeneous multiprocessor architectures that are complemented by reconfigurable logic resources. Figure 1.9 shows an architectural template for these architectures.

The hardware is a heterogeneous multiprocessor system that comprises a number of processing elements, realized as microprocessors or FPGAs (field programmable gate arrays). Each FPGA is associated with a set of configurations. Runtime reconfigurable FPGAs are associated with multiple configurations which are activated on demand. The nodes of the hardware architecture are connected through a common communication channel. The PEs may possess local communication channels to reduce contention on the global channel.

A dedicated PE acts as the system master. This node is commonly a microprocessor. It controls the overall control flow of the system, invokes functionality implemented by the slaves, and triggers the reconfiguration of the FPGAs.

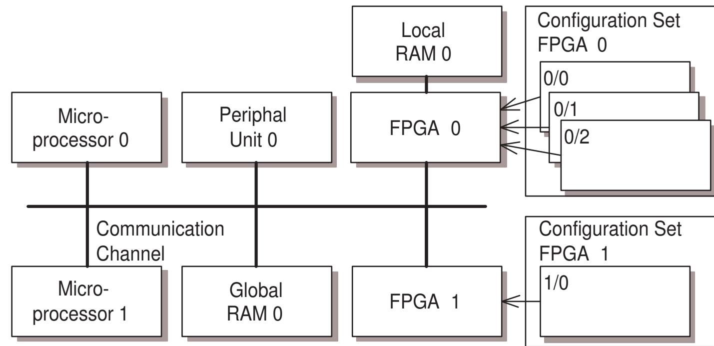

Figure 1.9. Hardware Architecture Template

The slaves commonly execute performance-critical behavior and special input/output operations.

3.2 Logical System Architecture

The logical system architecture is implemented with the resources of the hardware architecture. This architecture orients towards the object model of computation [3]. The system functionality is realized by objects communicating through structured messages. Each object has a state (the data encapsulated by the object) and offers services which may be accessed through the interface of the object. The services define the object behavior. A service is invoked by sending an appropriate message to the object. The respective service handler may change the state of the object and it may also send messages to other objects. Messages may be sent synchronously or asynchronously.

Objects are realized with PEs and memory resources. Objects of the same class may be executed on multiple PEs and may have different PE specific implementations. The messages are transmitted through the common communication channel or a local communication channel.

3.3 Design Space Exploration

During design space exploration a partition of the system function amongst the components of the target hardware architecture is computed. The implementation space, as defined by the target platform model, is explored for feasible alternatives implementing the system. The quality of each alternative being explored is estimated. The partition which optimizes the performance and satisfies the design constraints best is chosen for implementation. As a result, an implementation model and deployment model for the design model is computed.

Design space exploration can be performed fully automated by model compilers or based on models which have been partially partitioned by the designer. Of course, designers may also create the implementation and deployment model manually. The algorithms for design space exploration and hardware/software partitioning are chosen depending on the application domain and the model compiler. Model compilers for hardware/software systems are extendable to support user-specific partitioning algorithms. This degree of freedom in choosing the partitioning mode and algorithms is possible owing to the application of a common language. This is one of the strengths of UML based codesign of hardware/software systems.

Common object-oriented UML specifications are not directly implementable on the given target, due to the polymorphism and use of dynamic data structures. Thus during design space exploration those parts of the design model that are not implementable on a target platform must be transformed respectively. Design models are transformed to enable implementation and to optimize implementation and execution characteristics. The transformations are manifested in the implementation and deployment models.

3.4 Model Transformations

Model transformations are used to map and optimize design models. Mapping transformations are used during design space exploration to define the realization of design models. If a design model is not directly implementable model compilers search for a sufficient set of transformations. If no such transformation set exists the design model is not implementable with the implementation platform. There are three classes of transformations: allocations, behavior transformations, and structure transformations [13]. Allocation operators act on all elements of design and implementation models. These operators map model elements to the target platform by assigning sufficient sets of resource services.

EXAMPLE 1.8 *Figure 1.10 continues the implementation model of Example 1.6. For the action* `diff1=csample-psample` *in the operation* `encode` *the allocation of resource services is demonstrated. The respective activity group is shown in a compact notation, according to [9]. For the realization of the "-" operation, a sufficient resource service is allocated. The service is implemented by a* `sub` *operation of the VHDL type* `std_logic_vector-<16>`*. Other examples for this transformation type are allocations of storage and communication components as shown in Figure 1.5.*

Behavior transformations act on UML behavior specifications, e.g., state machines, activities and actions, and the model elements which implement them. Structure transformations act on model elements from which the structure of a UML model is built.

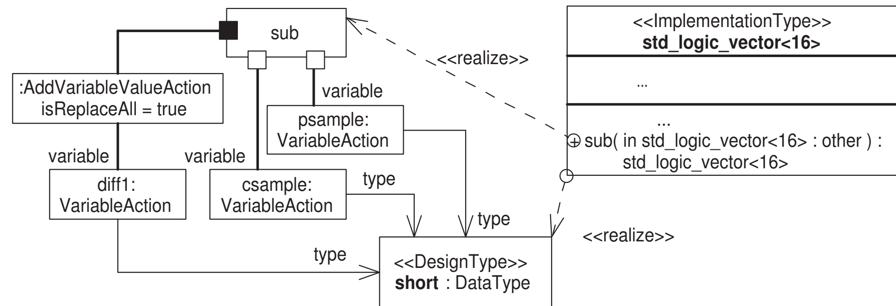

Figure 1.10. Allocation Transformation Example

EXAMPLE 1.9 *An example for behavioral transformations has already been given in Example 1.6. The interface and behavior of the* `encode` *operation are transformed in such a way that the samples are accessed via a bus interface. Similar transformations are necessary to accomplish message transfers in hardware modules. Another example of this kind of transformation are (de) compositions and optimizations, such as arithmetic/logic optimizations, dead code elimination, and constant value propagation. Behavior transformations are often accompanied by respective structure transformations.*

In the same way as programming language compilers and synthesis tools, model compilers may use transformations to optimize the implementation and execution characteristics of design and implementation models. In principle the most basic and advanced optimization techniques that are applicable to common object-oriented specifications are applicable to UML models as well, and there is good reason to do so. In contrast to the final implementation executable UML models describe the entire control and data flow of an application in a single and consistent representation. This information is used by advanced model compilers to perform more aggressive optimizations than would be possible on the final implementation. Examples are dead code elimination, constant and value range propagation optimizations that act globally on UML models [8]. Also optimizations known from behavioral synthesis are applicable.

4. Mapping Implementation Models onto Platform-Specific Implementations

4.1 Overview

The implementation of the system relates directly to the structure and behavior of the implementation model. The object-oriented properties of the design

model are preserved in the final implementation. Classes, interfaces, operations, and attributes of implemented objects map directly to their counterparts in the implementation model. Inheritance and polymorphism are preserved independently of the realization! In addition, the components and artifacts as specified in the implementation model are preserved in the implementation. Artifacts stereotyped as 'Configuration' map to configuration bitstreams of (reconfigurable) logic resources. The instantiation of a component manifested by such an artifact corresponds to the loading of the bitstream into the physical device. The same relationship applies for 'executable' artifacts and microprocessor devices.

For both hardware and software realizations, the question for the right level of abstraction of the final implementation and a suitable language arises. In an object-oriented approach it seems quite natural to choose a language supporting the object paradigm for software implementation. This approach makes the implementation convenient and straightforward. The final compilation is delegated to 3rd party compilers. However, as a result a fair amount of control over the final implementation is lost. In performance and resource critical applications, this uncertainty can cause iterations in the development flow. To avoid this problem model compilers for critical application domains may generate microprocessor specific assembly language implementations. For the purpose of this chapter C++ is used to implement software modules.

For the implementation of hardware modules the same considerations as for software apply. Owing to the tight timing and resource constraints imposed by the hardware it is even more important to reflect the implementation model directly in the implementation. In principle the implementation can be delegated to behavioral synthesis tools [4]. However, as for software implementations it is hardly possible to compute good estimates of the synthesized results. Moreover, the programming language based approaches, such as SPARK and Forge [14] [15], are restricted by the employed languages and the directly synthesizable language subsets of the targeted HDLs. Thus model compilers for SoC synthesize hardware modules directly from UML models on the register transfer level (RTL). For the purpose of this chapter, the implementation of hardware modules is described with synchronous VHDL-RTL designs.

The direct implementation of components, classes, and features is convenient and straightforward. Whereas in software this is a well understood problem, in hardware implementations this approach raises the following challenges:

Dynamic Object Instantiation/Destruction Owing to the static nature of even partially reconfigurable hardware, the efficient instantiation and destruction of hardware objects is not possible efficiently. The class instantiation per reconfigurable device is by far too expensive in terms of the number of required logic resources and reconfiguration time.

Polymorphic Features Polymorphism is an important property of object-oriented specifications. It should be supported directly by hardware implementations. Current approaches avoid polymorphism by prohibiting inheritance or overriding of behaviors.

Communication of Objects Objects should be able to communicate independent from their realization. In mixed software and hardware implementations no single, common mechanism for message exchange exists.

In the following sections the mapping of implementation models to hardware/software implementations is discussed. Owing to the focus of this book the main focus is on hardware implementations. Solutions of the stated problem areas are presented.

4.2 Software Implementation

The implementation model of a design determines the implementation of the design. Model compilers implement hardware/software modules that realize the same function and structure as the implementation model. Owing to different implementation patterns and styles multiple implementations are possible for an implementation model. These differences are reflected in the QoS in the implementation platform model so that it does not affect the quality of the design space exploration results.

The implementation patterns and rules are either manifested in the respective components of the model compiler or in code generation annotations in the UML meta model. The latter approach is taken by xtUML [7]. It has the advantage of being defined entirely with UML models and dedicated generation languages (archetypal language). However, it orients towards single language software implementations. Design space exploration, estimation, (automated) model transformations, and mixed language implementations are not directly supported. Thus in our approach the former approach is taken.

The classes of the implementation model being deployed on microprocessor nodes are directly implemented in C++. Local proxy objects manage the communication between local and remote objects. For each remote object that is accessed by a local object a proxy is instantiated locally. The proxy encapsulates the communication mechanism. Thus the objects of an application are not required to share a common address space. The proxy is explicitly modelled in the implementation platform model as *remote* type (see Figure 1.2). As a result, the model compiler can compute high quality estimates of the characteristics of distributed applications.

Objects which are implemented in reconfigurable hardware are managed by a dedicated service called *RTR Manager* [12]. This manager encapsulates the specifics of the reconfigurable hardware, e.g., reconfiguration modes, input and output functions, and communication. The most important task of

this service is to process application requests for the creation and destruction of hardware objects. Hardware objects are created and destroyed on demand. An application that instantiates an hardware object requests it from the RTR Manager by its type. The RTR Manager searches for a suitable object in the currently instantiated bitstreams and serves its proxy to the application. If no bitstream containing an object of the searched type is currently instantiated, the RTR Manager dynamically instantiates an appropriate bitstream.

EXAMPLE 1.10 *Figure 1.11 illustrates the basic architecture of the AudioPaK Example. The instance `main` of class `Main` and a number of proxies for hardware objects are implemented in software. The actual instances of the class `AudioPaKCoder` are realized by means of reconfigurable resources. Each hardware object is accessed from software through a dedicated proxy. The proxies are served to the application by an instance of the `RTR Manager` service.*

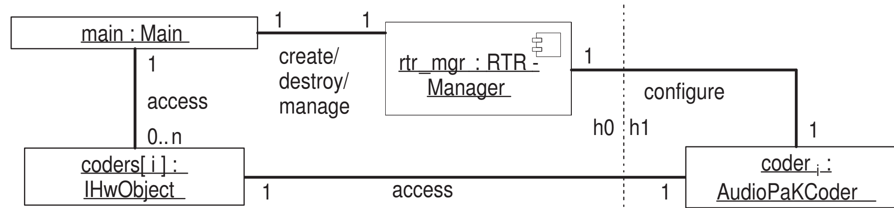

Figure 1.11. Software Architecture of AudioPaK Example

Proxies can be used directly in the software implementation to provide a simple yet fast mechanism for accessing the hardware objects. Alternatively, proxies may be wrapped by software implementations of the hardware object classes. If an instance of the software object is created the object tries to instantiate its hardware counterpart. In case of success the hardware object is used, otherwise the software object switches transparently to the software implementation. This approach also enables the transparent migration between hardware and software objects. Advanced model compilers generate such implementations automatically. Implementations are guaranteed to be correct because the compiler has generated them.

EXAMPLE 1.11 *Figure 1.12 shows a part of the operation `Main::main`. In the loop, a number of hardware objects will be created. Recall from Example 1.2 that the algorithm writes one frame with audio samples to the first coder object and starts it asynchronously. In the course of the algorithm it is checked whether the encoding is finished, and, if so, the encoded frame is read.*

```
...
coders = new smartptr<IHwObject>[ 5 ];
for(i = 0; (i < 5); i = i + 1) { // create coder objects
    coders[ i ] = RTRManager::getInstance()->createObject( 0 );
}
... // get new samples
coders[ 0 ]->write<short>(8, samples, 1152);
coders[ 0 ]->write<short>(2312, 1152);
coders[ 0 ]->start<char>(4, 2);   // start encode (async)
... // fill other coders
coders[ 0 ]->execute<char>(4, 1); // isFinished (sync)
if( coders[ 0 ]->read<bool>(2315) ) {
    samples =  ((short*)((int) coders[ 0 ].getPtr() + 8));
} ...
```

Figure 1.12. Software Implementation of `Main::main`

4.3 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. The hardware/software interface can be viewed from a logical and physical perspective. The logical hardware/software interface can be realized by different physical implementations. The concrete implementation depends upon the target platform and the model compiler.

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; the RTR Manager serves as object broker. Additionally, in RTR systems the objects and hardware configurations are dynamically bound to logic resources.

This mechanism is reflected in the life cycle of objects and bitstreams, which is illustrated in Figure 1.13. Because of the tight relationship between the hardware objects and their configuration context, as the container of the hardware objects, both life cycles influence each other. Each object and its bitstream will go through three states, `X_UNBOUND`, `X_BOUND` and `X_ALLOCATED` (where `X` is either `OBJ` for hardware objects or `BS` for bitstreams). As long as the bitstream is not loaded into the reconfigurable hardware, the bitstream and the contained objects are in state `X_UNBOUND`. When a bitstream is loaded it changes its state to `BS_BOUND`. The objects contained go to the state `OBJ_BOUND`. Objects allocated by the application change go from state `OBJ_BOUND` to `OBJ_ALLOCATED`.

All objects returned by the application will set their state back to `OBJ_BOUND`. The last object of a context returned causes the bitstream to be set back to `BS_BOUND`. Until the bitstream is unloaded from the hardware, the objects will still be available for allocation. The bitstream is not allowed to be unloaded from the hardware as long as it is in the state `BS_ALLOCATED`.

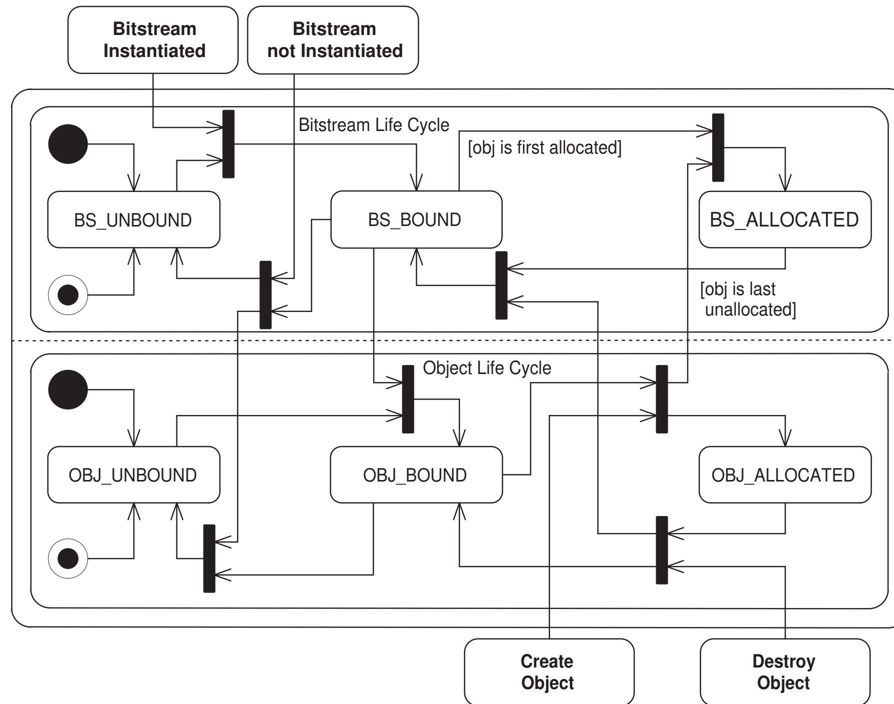

Figure 1.13. Object and Configuration Bitstream Life Cycles

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.

The *control interface* allows for identification, typing, and access to the object. Objects are uniquely identified in their object space. The ID represents the address of the object and is set during initialization. This field is only required if the object address must be made explicit in the object interface. The type field represents the dynamic type of the object. It is used to select the proper implementations of polymorphic features. This field is only required if the object may have different dynamic types. The message field uniquely identifies the type of service accessed by a message sent to the object. The service which

is executed in response to the message may depend on the dynamic object type. The message parameters are passed through the 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.

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. This enables other objects to react appropriately.

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.

4.4 Hardware Implementation of Components

Figure 1.14 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.

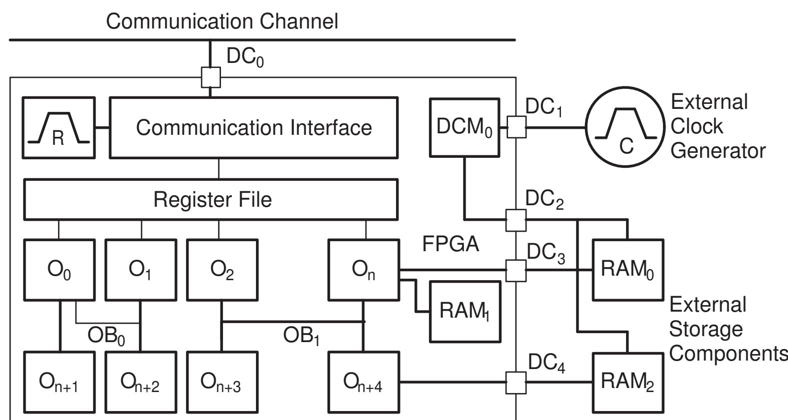

Figure 1.14. Hardware Component Template

The ports DC_i are manifestations of the `Device Connect` interfaces as modeled in the implementation platform model (see Figure 1.5). This interface type

is used to establish connections between internal logic resources and external devices through the physical device interface. From the UML specification of these interfaces the model compiler generates such implementations. The generation process is controlled by the generation information in the model. In the VHDL implementation the interface of the top level design module comprises of all signals of the DC ports.

Additionally required components modeled in the implementation platform model are instantiated on demand. For instance, if RAM_0 was realized with `ZBT RAM Bank 0` (Figure 1.5), a `DCM Clock` (Digital Clock Manager) component is instantiated (DCM_0) and connected to RAM_0. The interfaces of both components are modeled by UML interfaces and classes. The logic using the storage is connected to its `Local Access` interface. 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. If the number of sink flip-flops on the clock tree exceeds the maximum fan-out of the clock generator, a clock generator component is instantiated to buffer the signal.

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 number of concurrently required object instances is estimated from real or estimated execution profiles of the application [2].

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. During design space exploration the model compiler tries to identify reasonable groups of collaborating objects. For each object group an object bus is generated which connects all member objects of the group.

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.

Model compilers for SoC perform the interface layout during generation. In the real layout alignment constraints on the items in the register file, which are imposed by the communication channels, are considered. Model compilers assign to each element in the register file an address that satisfiess the alignment

constraints in the system. In addition the corresponding address decoders are automatically generated. Software modules accessing a hardware object use only the relative addresses of the member elements of the object interfaces. The software proxies are parameterized with the absolute object address by the RTR Manager. The proxies compute the absolute address of an element when it is accessed. This ensures that software and hardware always fit together and that the object access is independent of real object addresses and implementations.

EXAMPLE 1.12 *Figure 1.15 illustrates the implementation of a hardware component for our design example. In this implementation a PCI bus (Periphal Component Interconnect) is used as communication channel. The PCI bridge from our implementation platform (see Figure 1.5) is used to adapt the external bus to the internal bus* `Local Access`*. The external interface of the component was modeled as* `Device Connect` *UML interface. A register file is connected to the internal bus. The address decoders and address range decoders AD/ARD_i select the appropriate register from the file at each PCI access. The* `AudioPakCoder` *objects are also connected to the register file. Their control interface is implemented in the address decoders and the control register. The other registers realize the data interface; the BRAMs (Block RAMs) have been modeled in the implementation platform, the other registers are generated by the model compiler. Owing to the flat class hierarchy of the example no type register is required. The exception interface is empty because no exceptions are thrown in the example.*

4.5 Hardware Implementation of Objects

Figure 1.16 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 implements 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 behavior to be executed when the operation is started depends on the current object type. The selection of the behavior to execute is performed by a selector circuit whose implementation is

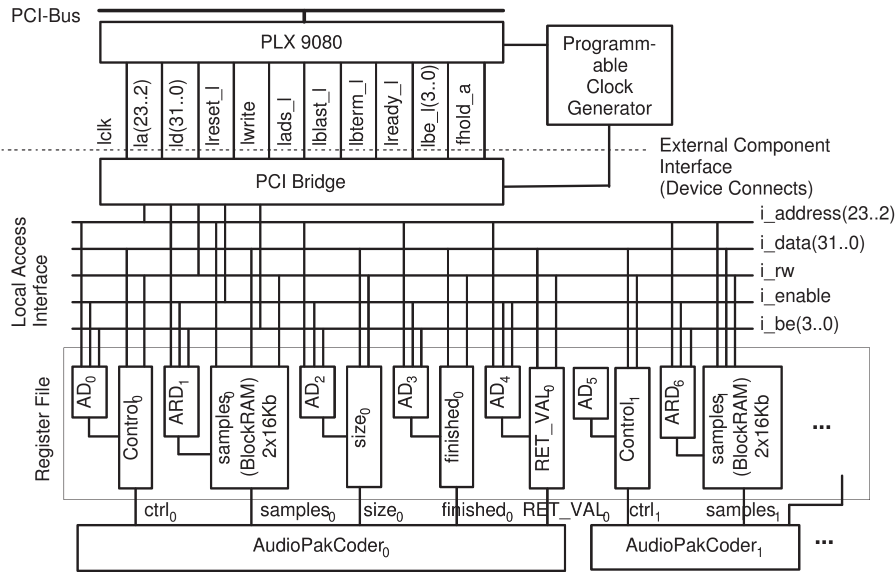

Figure 1.15. Component Implementation AudioPaK Example

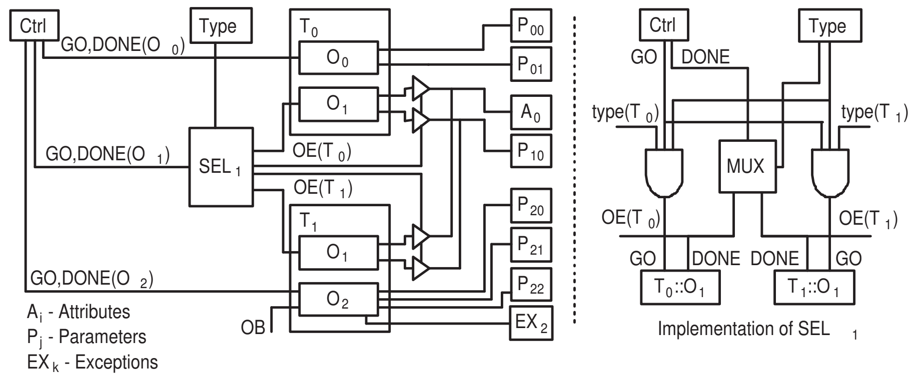

Figure 1.16. Hardware Object Template

depicted on the right hand side of Figure 1.16. 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 must be irrelevant which implementation of this operation is executed. Thus 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 actual data by appropriate logic. If no other output enable was specified in the model the *DONE* signal is used.

With a growing number of polymorphic operations the implementation requires a reasonable amount of logic resources. However, the support of polymorphism also provides significant advantages. If the implementation supports the object-oriented features the designer has more freedom for the system specification. Moreover, because object-orientation means implementing the differences between classes, the direct implementation of class hierarchies can help to reduce the amount of logic resources required. In each class of the hierarchy only the new and overridden features of the class in comparison to its superclasses must be realized. Also the probability that an object of a required type is contained in the current bitstream is raised because there are virtually more objects of different types. Hence the overall number of reconfigurations may drop. In experiments we have shown that this approach can improve the overall performance by orders of magnitude [12]. The full implementation of class hierarchies is only advantageous however, if the classes in the hierarchy are actually instantiated by the application.

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 (`Local Access`, `External Access`), the resources modeled are used to generate the storage components. For performance and synchronization reasons 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 significant synchronisation problem because multiple objects or operations may access features concurrently. For the publicly visible objects the software proxy objects synchronize concurrent accesses by sequentializing them. 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.

If the implementation uses an implementation type or operation which implements the `Device Connect` interface, the signals of this interface are routed to the top level VHDL module. This mechanism is used to include peripherals into the generated designs.

EXAMPLE 1.13 *Figure 1.17 continues the implementation of the AudioPaK-Coder example. As the figure suggests, this implementation is very simple*

because the `AudioPaKCoder` *does neither implement polymorphic behavior nor does it access other hardware objects; hence intra-device synchronization problems do not arise. The data interface of the object is realized in the register file as presented in Example 1.12. Notably, the interface to access the sample array has been transformed into a WISHBONE-like bus interface [11]. The transformed interface was already introduced in the implementation model (see Example 1.6).*

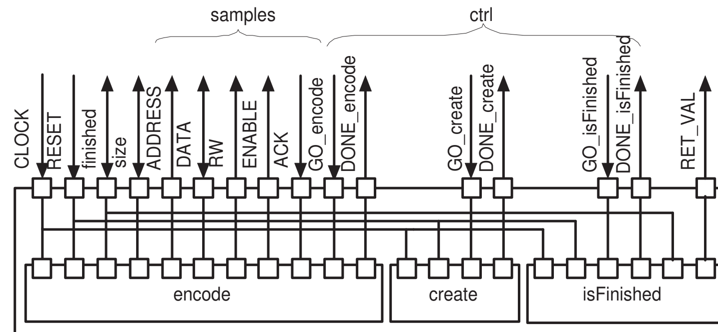

Figure 1.17. Object Implementation AudioPaK Example

4.6 Hardware Implementation of Behavior

The behavior of classes and operations is implemented according to the FSMD model (finite state machine with datapath) as Moore FSM [5]. 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. For each PE the global scheduling policy is specified in the deployment platform model. For a better control of the implementation the designer may also specify a local, operation-specific policy in the implementation model.

The controller is realized as FSM. Each of the operations of the datapath is associated with a number of states of the FSM. 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. State transitions are performed synchronously.

EXAMPLE 1.14 *Figure 1.18 shows the realization of the first loop of the behavior of the operation `AudioPaKCoder::encode`. Owing to the lack of space, the computations in the loop are not shown. The FSM* [2] *on the left side is decomposed into a controller, a datapath, and a synchronization process. The loop is executed as long as the loop counter `i` is less than `size`. The synchronization process is not shown. It synchronously sets the current FSM state and the `DONE` signal when the FSM is in the final state.*

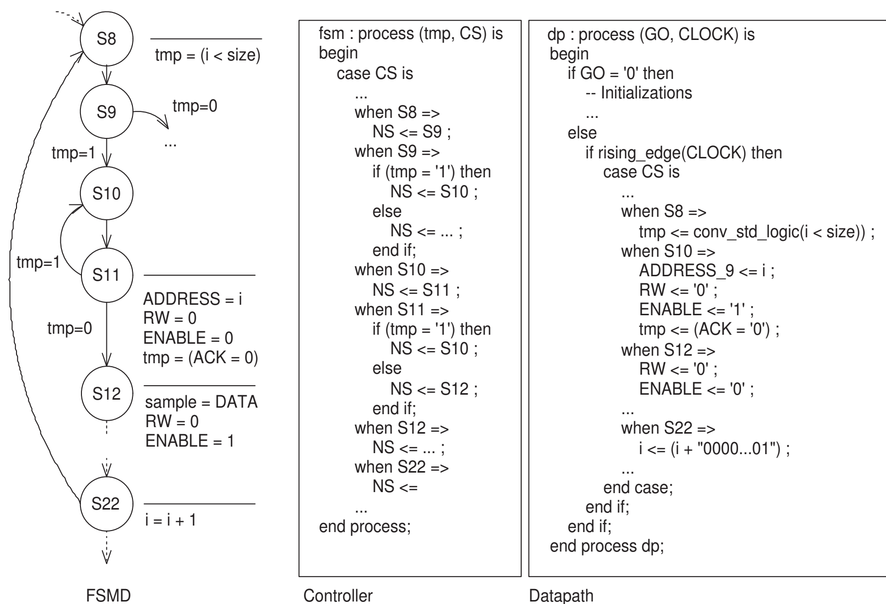

Figure 1.18. VHDL Loop Implementation Example

In contrast to software implementations each behavior is realized for each hardware object, that is, the implementation of a behavior is not shared by objects which provide the same behavior. Hence no synchronization of concurrent executions of the same behavior in different objects is required.

[2]We use the FSM notation here instead of UML State Machines in order to emphasize the relation to the VHDL implementation. Also the currently implemented FSMs have less powerful semantics as UML State Machines.

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 computation and when the behavior executes a message transfer.

As shown in the previous example, array accesses in the model are transformed to explicit bus transfers. Similar transformations are performed to accomplish message exchange between objects. Message transfers between operations of the same object are commonly inlined by the model compiler. This requires more implementation resources but again minimizes data contention.

5. Experimental Results

The design model from our permanent AudioPaK example was automatically implemented with the MOCCA compiler using C++ and VHDL-RTL implementations platforms. The key concepts of the implementation have already been presented in the previous sections. The overall compilation/synthesis time of the design model into the final hardware/software modules takes approximately 5 to 10 minutes, depending on the degree of optimization.

The coder was tested on a hardware platform comprising a Pentium IV processor running at 2.4 GHz (master) and a Xilinx Virtex-II FPGA with approximately 3 Mio gate equivalents running at 100 MHz (slave). Master and slave are connected with a 33 MHz PCI bus. The slave implements the AudioPaK coder objects, and the master is responsible for the filtering and distribution of the audio information to the coder objects and the audio clients in a network.

The current scarcely optimized FPGA implementation of one coder object requires about 1800 slices, which corresponds to 12% of the available area. Additionally two 16Kbit BRAMs are used to store the sample array. The majority of the resources is consumed by the `encode` operation. The resources required for the PCI bridge are neglectable.

The FSM generated for the `encode` operation consists of 214 states, which results in a complex controller. Despite the relatively low arithmetic complexity of the encoding algorithm, its implementation is rather expensive. The algorithm comprises six loops realizing the intra-channel decorrelation, computation of the encoding parameters and the actual coding of the audio samples. The algorithm potential for parallelization is quite small.

In Figure 1.19 the performance characteristics of the coder implementation is shown. The coder implementation is quite efficient. Depending on the frame size, between 15 and 20 audio channels can be encoded concurrently with one coder object at 96KHz sampling rate. To avoid contention on the objects, we found it useful to implement up to five coder objects that are executed in 'round robin' order. The time to transfer the samples from/to the hardware may

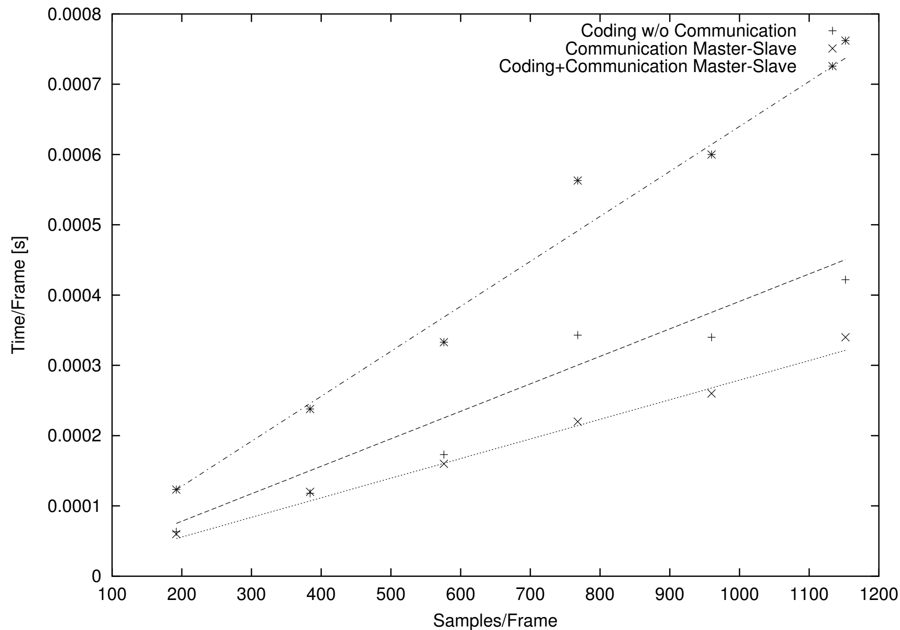

Figure 1.19. AudioPaKCoder Coding and Communication Effort

be dropped in the future by connecting the coder directly to the audio input devices.

6. Conclusions

In this chapter we have presented a novel approach to the UML-based development of applications for reconfigurable architectures. The approach incorporates the key concepts of model driven architecture, platform based design, and hardware/software codesign. We have shown that UML can be used beneficially to develop a wide range of relevant SoC applications. This has been exemplified with a simple application for the encoding of audio data streams. With this application the transformation of platform-independent UML design models into final implementations of hardware/software modules has been demonstrated.

The paradigm of platform based design can be used by specifying platforms with UML models. Platform models make the important abstractions, assumptions, and constraints of platforms explicit. The formal description of platform models with UML makes them automatically interpretable. In contrast to conventional approaches, platform models enable the flexible, yet automated, transformation of UML models into final implementations. These transformations can be performed (semi-) manually or completely automated by model

compilers. As a result, the capabilities of current approaches to behavioral synthesis and compilation are pushed to the system level. This improves system quality and enables one to cut down development time by orders of magnitude.

Whilst the software implementation of object-oriented specifications is state of art, this chapter 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.

The capabilities of our approach have been demonstrated with a real world design of a coder for high quality audio streams. With this example it has been shown that the approach provides significant gains in system quality and development efficiency. Especially does the short implementation time motivate the exploration of different design alternatives in order to improve the overall quality and to reduce costs.

References

[1] Th. Beierlein, D. Fröhlich, and B. Steinbach. UML Based Codesign of Reconfigurable Architectures. In *Proceedings of the Forum on Specification and Design Languages (FDL'03)*, Frankfurt a.M., Germany, September 2003.

[2] I. Drost. *Estimation of Execution Probabilities and Frequencies of OO Models*. Diploma Thesis, 2003. University of Applied Sciences Mittweida, Germany.

[3] A. Eliens. *Principles of Object-Oriented Software Development*. Addison Wesley Longman Publishers, 2000.

[4] J. P. Elliott. *Understanding Behavioral Synthesis*. Kluwer Academic Publishers, 2000.

[5] D. D. Gajski. *Principles of Digital Design*. Prentice Hall Inc., 1997.

[6] M. Hans and R. W. Schafer. *Lossless Compression of Digital Audio*. Technical report, Client and Media Systems Laboratory, HP Laboratories Palo Alto, 1999.

[7] S. J. Mellor and M. J. Balcer. *Executable UML - A Foundation for Model Driven Architecture*. Addison Wesley Longman Publishers, May 2002.

[8] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[9] Object Management Group. OMG Unified Modelling Language Specification (Action Semantics). Version 1.4., January 2002. `http://www.omg.org`

[10] Object Management Group. OMG Unified Modelling Language Specification (Super Structure). Version 2.0, July 2003. `http://www.omg.org`

[11] Silicore and OPENCORES.ORG. Specification for the: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. Revision B.3, September 2002. `http://www.opencores.org`

[12] H. Riedel. *Design and Implementation of a Run-Time Environment for RTR-Systems*. Diploma Thesis, January 2004. University of Applied Sciences Mittweida, Germany.

[13] B. Steinbach, D. Fröhlich, and Th. Beierlein. *Languages for System Specification*, chapter UML-Based Codesign for Run-Time Reconfigurable Architectures, pages 5–20. Kluwer Academic Publishers, 2004.

[14] University Irvine, Center for Embedded Computer Systems. *SPARK: High-Level Synthesis using Parallelizing Compiler Techniques*. April 2004. http://mesl.ucsd.edu/spark/

[15] Xilinx Inc. *Forge Compiler for High-Level Language Design.* 2003. http://www.xilinx.com/ise/advanced/forge.htm