Generation of deterministic MCU/FPGA hybrid systems from UML activities

Ruediger Willenberg
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
Email: willenbe@eecg.toronto.edu

Zamira Daw, Christian Englert, Marcus Vetter
Dept. of Information Technology Engineering
University of Applied Sciences Mannheim
D-68163 Mannheim, Germany
Email: {z.daw, c.englert, m.vetter}@hs-mannheim.de

Abstract—In this paper, we extend a model-driven design process for embedded systems to FPGA target platforms. We introduce a VHDL code framework that implements buffered data flow structures and encapsulates hand-written computation kernels. We demonstrate how a toolchain uses the framework to generate system code from behavioral models called activity diagrams. Further, we introduce an interface solution that allows the designer to model and generate hybrid systems of microcontrollers and FPGAs together. We examine consistency issues between programming models and present a case study for cross-platform system implementation.

I. INTRODUCTION

Increasing performance requirements on real-time embedded systems have made the design process more complex and error-prone. Growing hardware capabilities like multicore and multi-threaded parallel processing only exacerbate the complexity issue. This is especially true of safety-critical systems in the medical, automotive and aerospace domains.

The emergence of hybrid systems, which combine classical CPUs and microcontrollers with specialized hardware like Digital Signal Processors and reconfigurable devices, adds new complications to the design process. Different platforms use different semantics and programming models. Moreover, efficient interfaces for each distinct combination of platforms must be defined and implemented.

We propose to use Model Driven Engineering to design and implement hybrid systems of C++ and VHDL-programmed platforms. To bridge programming-model and interfacing issues, we use activity diagrams (see Figure 1 for example), which model data and control flow as part of a behavioral, algorithmic description.

We present the priorly introduced DMOSES (Deterministic MOdels for Signal-processing Embedded Systems) development process [1], which provides a C++ framework for automatic model-based code generation. We extend the framework with an equivalent VHDL code base to generate data and control flow structures suited to FPGAs. Specific computation kernels are still implemented by the programmer in an efficient, platform-specific solution. The framework provides encapsulation of a kernel that makes it compatible with other components and allows design re-use in other solutions.

We also provide an infrastructure for inter-platform communication within the bounds of the behavioral model.

A platform-specific solution for connecting an on-chip PowerPC® system with C++ components to VHDL components on the same FPGA is detailed. From the software perspective, a filesystem-based Linux driver is offered. The driver interfaces as well as the modeled VHDL components remain unchanged even if the actual hardware connection between MCU and FPGA changes.

The paper is structured as following: After exploring related work in Section II, we explain the methodology of model-driven development with UML in Section III. Section IV introduces the original DMOSES process for modeling and generation of deterministic C++ code. We detail our VHDL additions and related considerations in Section V. An infrastructure for consistent cross-platform modeling and a case study are presented in Section VI. In Section VII, we discuss possible consistency issues and planned future work. Section VIII concludes.

II. RELATED WORK

Graphical approaches to system modeling have been standardized in the last decade by the Object Management Group (OMG) in the Unified Modeling Language (UML)[2]. UML is a formal, object-oriented definition of 14 diagram types that model structure and behavior of software. The OMG has promoted software development approaches based on modeling under the term Model Driven Architecture (MDA)[3], also known as Model Driven Engineering. This includes processes that directly generate target platform code from detailed models.

The generation of code for digital systems from UML has been examined before. However, most of these systems concentrate either on the implementation of structural models like class diagrams[4], the implementation of automata with state diagrams[5][6] or a combination of both[7]. Schattkowsky et al.[8] have used sequence and state diagrams to generate Handel-C. Fernandes et al.[9] have examined generating VHDL from Petri nets[10], which are related to activity diagrams.

We have decided to concentrate on the behavioral activity diagram because it best combines control and data flow semantics that overlap between processor and FPGA programming models. The activity diagram also allows a behavioral
description while facilitating the structural partitioning, as we will demonstrate in Section VI.

III. METHODOLOGY

A. Model-Driven Engineering

Model Driven Engineering is characterized by an application-centered approach to system design. The designer starts his work independently from the later implementation platform by modeling the functionality needed to solve a certain application problem. Only after identifying the resource requirements a platform needs to be specified. If structural and behavioral models are refined to a detailed level, implementation can happen through automatic code generation.

A typical Model-Driven Engineering flow is shown in Figure 2. Since most development processes are fitted towards a certain application domain, (deterministic embedded systems for DMOSES), a domain-specific UML profile is applied before the modeling starts. This way, the UML diagrams can be extended with specific properties necessary to accurately model the intended solution (Section IV-B details DMOSES profile aspects).

When the modeling phase is finished, a Model-to-Model-transformation is executed. In this step, the data structure that represents the graphical model and its properties is parsed and transformed into an instance of a domain-specific metamodel. This metamodel defines a class and instance structure that closely mirrors the intended code structure, but is still platform independent. In the final step, called Model-to-Code-transformation, the metamodel is then again parsed with the help of code generation templates to generate the platform-specific code.

B. UML Activity Diagrams

A UML activity diagram (Figure 1) is a directed graph that is used to describe the sequence of execution steps of an algorithm or any other planned activity. It is related to other graph-based algorithmic descriptions like common flowcharts, Petri nets and Kahn process networks[11]. Its main components are:

- Control nodes: Essential elements of execution control for starting, ending, conditional branching, forking and joining of flows are modeled by specialized control nodes. Figure 1, for example, includes an Initial node as well as a Fork, a Merge, and a Final node.

- Action and Activity nodes: The rounded rectangles of actions and activities symbolize the actual operations happening in an algorithm. Action nodes are considered “black boxes” which do not specify any information about their inner workings except their interfaces to the outside world. In contrast, activity nodes denote an instance of another activity diagram, thereby enabling modular design and re-use of functionality. Activity nodes are distinguished from action nodes through a small fork-symbol in the corner (e.g. nodes CameraTrigger and Recognition in Figure 1).

- Edges: Edges model directed flow between the nodes and therefore denote the sequence of execution. The most important extension over flowcharts is the introduction of data flow, called object flow in UML terminology, in addition to control flow. By attaching squares called object pins to actions and activities, input and output data (and therefore interfaces) are modeled. Edges connecting object pins denote the handover of data between nodes.

To illustrate data and control flow semantics, activity diagrams assume the existence of tokens. Initial nodes as well as input-side object pins of an activity each offer a token to their connected edges when the activity execution starts. Those tokens are transported by the edges to the following nodes. An action or activity node only starts execution when all input object pins received a token and all incoming control flows also brought a token. (Note: while the number of incoming
data objects is clearly defined through the number of input object pins an action type has, any number of control flows can be connected to the node. The same distinction holds for outgoing flows.) After the action has finished execution, all the outgoing flows are provided tokens at the same time.

IV. THE DMOSES DEVELOPMENT PROCESS

The DMOSES process has been designed with the goal to avoid embedded systems models and specifications that lead to instable and unpredictable behavior. Its main components are:

- An extension profile for UML activity diagrams that introduces additional model properties to assure deterministic, platform-independent behavior
- A metamodel structured so that instances of it can easily be converted into code
- A C++ source framework that serves as a basis for code generation by template
- A toolchain, based on the open source Integrated Development Environment Eclipse[12], which brings together modeling, model transformation and code generation in one place

A. Semantic ambiguities in behavioral UML

Figure 3a shows a simple example of an underdefined model. An initial control flow is being forked into two control flows. If this activity is executed on a single-threaded system, one of the two execution flows is arbitrarily preferred. Even on a multi-threaded system of any kind, it is completely unclear in what time relation the actions A and B are executed. If the actions share resources, this implicitly leads to race conditions. Moreover, while the model is behavioral and supposed to be platform-independent, the execution order can arbitrarily change with a change of platform. Given that A and B could also be stand-ins for more complex modeled sub-activities, further conflicts could lure under the surface.

B. Profile extensions for determinism

The DMOSES profile avoids these model ambiguities through two additional obligatory flow properties:

1) Any one of several parallel flows needs to be assigned a priority value as in Figure 3b, where a lower number means this flow is completely executed before the flow with the next higher priority number starts. This section of the model is sequentially executed; on multi-threaded machines, only a single thread is run for it.

2) Flows that do have no dependence to others can be labeled with the async property (see Figure 3c), which leads to concurrent execution. Multi-threaded platforms start executing separate independent threads.

While this looks on first sight like a bad use of parallel resources, it assures complete determinism and avoids unpredictable behavior in complex model implementations. A different hardware platform does not lead to different behavior.

C. C++ framework

The DMOSES C++ framework is a complete object-oriented hierarchy that has corresponding classes for most activity diagram components like generic actions, control nodes and pins. Using inheritance, specific action classes can then be generated, e.g. the following simplified example:

```c++
int class AddAction : public Action
{
    public: InPin<int> in1,in2;
    OutPin<int> out;

    bool calculate()
    {
        out.setValue( in1.getValue() + in2.getValue() );
        return true;
    }
};
```

A class like this emerges through the following typical steps:

- The designer models an activity and inserts a new action of type Add.
- After activity modeling is finished, the designer starts up code generation.
- The code of the whole activity type, including instantiating all nodes and connecting them, is generated. Also, for each new action type like the Add, a new class is generated.
- The designer specifies the action functionality by hand inside the calculate-method.
- The class definition is checked into the library for future re-use.

Note the use of templates for the input and output pin typing: this means that the same pin classes and transfer mechanisms can be used for any object type.

During execution, the two input pin objects separately get data from output pin objects they are connected to. As soon as both pins have data, the calculate method is engaged. When all output pins have been written, they transport their data to the next following nodes.
V. DMOSES VHDL FRAMEWORK

A. Pipeline analogies

Formal UML semantics define activity execution as a process of sequential steps that handover their results from input to output through the chain of processing actions. For a sequentially running CPU, which implements the actions as functions, this is an acceptable approach. To make use of FPGAs, especially in consideration of their much lower operating frequency, processing blocks (e.g., action nodes) should be implemented as hardware units. UML Activities as the one displayed in Figure 1 suggest to the digital designer pipeline-like characteristics. This way, results can be transported and temporarily stored in pipeline registers between the functional blocks. The potentially useful effect of this is that the pipelined units cannot just be used in sequence but in parallel, working on several data packets concurrently.

B. Autonomous FIFO-based data flow

Pipelines generally need a sophisticated flow control to avoid losing data when a functional stage unit takes longer than one cycle for its operation. At the same time, stages behind the stalling unit need to be made aware that no valid data to process is offered to them. While this can be finely tuned in a fixed design, control logic for a pipeline based on a freely modifiable model cannot easily be generated. Sufficient timing information about particular actions might not even be available at design time. Moreover, creating a control logic external to the node implementations conflicts with the goal to keep the generated code structurally close to the model.

We have instead decided on substituting the simple register stages with synchronous FIFOs. Figure 4a shows a FIFO with customary control signals Write and Read as well as indicators Full and Valid (=not empty) (For simplicity reasons, clock and reset signals have been left out). By extending each control signal with a gate, we make sure that the FIFO protects itself from illegal reads and writes. We further stipulate that such FIFOs are used to implement the input and output pins of our activity model. Figure 4b shows two connected action pins and their FIFO representations. By connecting the gated FIFO inputs we have created an automatic flow control: A data word is only transferred if the sender has valid data and the receiver is not full. The FIFO pin structure makes sure that no data is ever lost and no unit processes invalid data.

C. Action node implementation

The general structure of a VHDL action node is displayed in Figure 5. The action node interfaces to the outside through its input and output pin FIFOs. In addition to the pins the node also needs function blocks to administer incoming and outgoing control flows, called InTokenPort and OutTokenPort.

Control tokens are realized as 1-bit-words that are equivalent to the Valid-signal for data words. A token port for a single control flow can be understood as a FIFO that does not actually store data, but only keeps track of incoming and outgoing words/tokens. Since any number of control flows can enter and leave an action node, token ports are implemented as an array of such token counters dimensioned at synthesis time through generics.

Analogous to the C++ behavior, an action is supposed to start operation as soon as data has arrived in all input pins and tokens are offered by all incoming control flows. The enable signal for the operation is called calculate in the VHDL implementation. It is triggered when all input pins have active Valid signals and the combined token port reports tokens on all
incoming control flows. The function kernel is then responsible for reading the inputs, performing the operation and writing the outputs.

D. Control node implementation

The behavior of control nodes is completely defined in UML, and therefore their VHDL implementation does not need any additional designer input. Exemplary, Figure 6 shows the structure of a fork node for object flow. It has the obligatory pin FIFOs as input and output buffers. Since the only functionality lies in duplicating flow, the operative transfer signal becomes active when incoming data exists and all outgoing data flows have free capacity. Transfer triggers a read of the incoming data and writes it to all outgoing buffers at once. The equivalent fork node for control flows works analogously, with TokenPorts substituted for FIFOs.

E. Data types for CPU/FPGA hybrid systems

The implicit assumption up to this point was that data is handled and transferred in units of separate data words. The term object flow employed by UML suggests the use of more complex data structures, as does the templating used in the C++ framework. These include arrays, multi-word data like complex numbers as well as inhomogeneous data objects built by structs.

For data structures reaching a certain size, it becomes inconvenient to transfer all parts of them in a single cycle. For once, it takes immense routing resources to move data across the chip this way. Moreover, it implies that processing resources for the whole package are provided, possibly a large use of area. This is especially questionable with our later goal of crossing the CPU-FPGA boundary in mind, where even with DMA in place, a maximum communication bandwidth of 1 or 2 processor words per cycle is likely.

With an eye on array-like data structures, we propose the use of “two-dimensional” properties. Figure 7 shows the handling of a double precision quaternion, a number with one real and three orthogonal imaginary components (Quaternions are very useful in navigation applications). The data is handled with a width of 64 bit, corresponding to the underlying double precision floating point type. Furthermore, the data has a depth of 4 words, implying the processing and transfer of the data in four cycles and the use of four storage words in a FIFO.

Pin FIFOs implemented in VHDL need a typing mechanism equivalent to the templating that C++ pins use. Dependent on the type of data stored and transferred, storage size and access method need to be synthesized. We introduce an enumeration type called classtype. Its set of defined values includes names of data types needed to implement models in VHDL. The intention is to stay consistent with data classes defined in the model and used in the C++ implementation. Furthermore, we introduce two helper functions clwidth and clddepth, that allow the type properties to be retrieved in VHDL code. Table I gives examples including a complex floating-point number, a quaternion and a 1K-array of 8-bit samples.

As a result of those definitions, a pin FIFO can now be dimensioned by setting a classtype generic. Additionally, a buffer size generic can be specified if more objects than the default number of 2 should be stored in one FIFO. The FIFO can generate its data width by using the clwidth function and its depth in words by multiplying the buffer size with the clddepth return value.

F. Functional FIFO extensions for action node design

The functionality of the generated FIFO has not changed through the introduction of class properties. Multi-word objects and arrays are still read and written word by word, and Valid and Full signals still indicate on single-word granularity. To reinforce the handling of data as packets of words, we introduce the optional Object Mode. In Object Mode, a single-cycle write or read signal indicates transfer of a full multi-word object in consecutive cycles; correspondingly, Valid signals a complete available object and Full signals enough storage space for one object. While we consider Object Mode a semantical improvement, it can incur additional latencies;
therefore it can be individually deactivated for write and read sides of a pin FIFO.

As a further extension of Object Mode, we introduce Random Access Mode as a flexibility option to action kernel designers. Since complex packets of data will generally not be processable in purely sequential order, an available object initially stays stored in the input pin FIFO. Single data words of the object are accessible, however, through an address port (this port does not index the actual address of data in the underlying RAM, but only an object-relative offset). When all relevant data in the object has been accessed, the object is “flushed” by asserting the Full-In/Read signal once. Correspondingly, an output pin object can be randomly written in Random Access Mode, with a write offset port and an additional write enable signal. Through use of the Valid-In/Write signal, the object is declared fully written and “sent off”. The Random Access Mode saves area by avoiding to buffer a data object again inside the action kernel.

VI. MODELING AND GENERATING HYBRID SYSTEMS

A. UML modeling of hybrid systems

DMOSES makes it easier to employ the strength of each different platform. For that purpose, it offers the possibility to assign discrete function blocks of a modeled activity to different execution devices (see Figure 8). Each action or activity node has a resource property. It is based on a model of resource types and instances.

A resource type specifies a platform for which the same code can be generated. C++ and VHDL obviously warrant different resource types; also, different CPU or FPGA models or families (implying different toolchains, libraries and vendor-specific constructs) might need individual types. From these types, resource instances for all existing devices on the target hardware are then declared.

During activity modeling, the resource property of each node is assigned a specific instance (a default is usually defined, so that everything that is not explicitly marked otherwise is, for example, run in C++ on CPU1). During code generation, a Model-to-Code transformation template corresponding to the resource’s type is chosen and the generated code is stored in a source directory assigned to the instance device. All the code for a device is then built to a bitstream or binary and programmed into the target system.

B. Interface for an on-chip hybrid system

As a first cross-platform system, we have chosen to connect an FPGA on-chip PowerPC® system with logic based on DMOSES-generated VHDL. An overview of the system is displayed in Figure 9. The processor system composes an enclosed unit which connects to off-chip DRAM and peripherals. The unidirectional connections to and from VHDL components, implementing activity edges, are managed through processor bus peripherals called DMOSES OUT and DMOSES IN.

The peripherals are dimensioned at code generation time for the number of flow edges necessary and the object properties of each implemented edge. The essential element of each connection is again a FIFO that buffers the conversion from the 32-bit-single-word processor bus to the object dimensions or vice versa. For each edge, a register space of four 32-bit-words is reserved. It includes read-only registers for the object properties and a unique pin ID; Full or Valid bit; a 32-bit data port for the FIFO; and Reset and IRQ-Enable switches. The FIFO (an output example is depicted in Figure 10) is always dimensioned with the larger of the two bitwidths; the other side is multiplexed to its target bitwidth. It shows Object Mode behavior in its connection to the logic nodes as well as in its Full/Valid signaling towards the processor.

On the software side, the interface is implemented by generating special input and output pin code that accesses a
device driver. Our implementation is running a Linux kernel of version 2.6.29. On initialization, the system polls the DMOSES OUT and DMOSES IN register spaces for object properties and pin ID. It then installs and configures a character driver node for each flow edge into the device filesystem. Each connection can then be read respectively written by accessing the device with the corresponding pin ID, which is known in the generated C++ instance.

While the actual driver is specialized on the exemplified hardware interface, the access through the Linux device driver system is completely portable to any other cross-platform interface. Presumably, similar abstract interfaces are easy to realize for other embedded software platforms.

C. Execution issues

The implementation considerations for hybrid systems brought a set of cross-platform issues to the forefront. As displayed in Figure 8, the current model of software threads executes a dataflow up to the FPGA node, skipping the functionality implemented in VHDL. The CPU resumes execution with the output data from the FPGA-implemented node. This behavior subverts the high-throughput approach aimed for by using pipelined structures inside the FPGA. With any one C++ thread executing a flow like the one depicted, a maximum of one data object would be processed concurrently in the FPGA.

Furthermore, there are less clear-cut dataflows imaginable, e.g. when data-dependent control flows lead data to different output pins or dismiss obsolete data objects completely. In those cases, it would not be clear if, where and when a C++ thread had to pick up data and continue execution.

To forestall those issues, a dataflow from FPGA to CPU needs always to be declared async, resulting in an independent thread processing FPGA outputs; CPU-to-FPGA flows can suspend or restart from the beginning after submitting data to the FPGA.

D. Case Study

As a first cross-platform application, we have designed the configurable audio filter path displayed in Figure 11. We deliberately decided for an initial low-computation application to keep valid bandwidth issues on the sidelines. The design is targeted to a Xilinx ML507 board with a Virtex-5-FX70T chip. The VHDL side includes interfaces to an AC97 audio codec chip (peripheral inputs and outputs are modeled in DMOSES activities as arrow-shaped signal nodes). The received 20-bit stereo signals are combined to a mono signal (to keep the example simpler) and sent to a 32-point Fast Fourier Transform. The resulting Fourier coefficients are handled over into the CPU domain, where they are combined into 16 double-logarithmic spectral values to be displayed as a bargraph by an onboard LCD display. On the input side, the CPU system can receive a new set of coefficients for any filter characteristic through an RS232 line. The assembled coefficient object is sent over to the logic side and into the FIR filter. Different coefficient sets for all-pass, low-pass, high-pass, band-pass and band-stop were computed and sent from a PC to the demonstration system, eliciting the expected acoustic results. The filter characteristics were also clearly identifiable on the LCD bargraph.

VII. DISCUSSION

A. Determinism problems in FPGA systems

As raised in Section VI-C, the behavioral disparities in FPGAs and CPUs cannot be completely ignored. Another consequence of the chosen FPGA implementation is the inherent concurrency of flows: If a data flow is for example replicated by a fork node, all outgoing flows are processed at the same time. On a CPU, this would only be happening if the flows were labeled as async and if there were enough resources for multiple concurrent execution threads.

Furthermore, this forced concurrency in combination with pipelining can lead to correctness problems and race conditions as illustrated in Figure 12. A decision node routes data words to different processing nodes depending on the sign of the data. The three paths have different latencies. Therefore, the processed data is re-sequenced by the merge node in a different order than the one it entered in. In most cases, this will be undesired. Thus, the FPGA execution offers new potential for non-deterministic behavior. One of the declared intentions for the DMOSES development process is to prevent
exactly that. Hence, our next important venue of inquiry is to identify all possible problems in FPGA implementations and find semantic constraints that do not inhibit efficiency or sacrifice a large part of the cross-platform modeling commonalities.

B. Future Work

In addition to further examination of the determinism issue, the following extensions to the DMOSES VHDL framework are planned:

1) **State diagrams**: UML state diagram are an excellent orthogonal extension for behavioral description and can be effectively linked to activities. Similar determinism issues as for flows need to be considered.

2) **Testbench generation and component profiling**: To facilitate module verification and worst-case execution time (WCET) analysis, VHDL testbenches could be provided through the existing code generation process.

3) **Area, Speed and Bandwidth optimizations**: By introducing a compiler-like optimization phase into the development process, underused hardware units could be shared and high-latency units could be cloned.

VIII. CONCLUSION

We have introduced our VHDL framework for the DMOSES development process. It facilitates implementation of activity diagrams for digital systems. FIFO-based flow control allows the connection of independent components without a central control instance; the code structure can therefore stay true to the model structure.

We further introduced an infrastructure to let C++ and VHDL domains communicate transparently, so that activity diagram nodes can be assigned to the hardware platform more suited to the task. We demonstrated a functioning cross-platform audio path generated from a model.

The enhanced DMOSES process clearly has the potential to simplify the development of embedded systems, decrease the likeliness of instability or failure and ease maintenance of existing projects. With our VHDL framework and cross-platform interface, we have opened the process to FPGAs, an increasingly popular type of hardware platform. Being able to use integrated models for CPU and FPGA platforms bridges the programming model divide and frees the designer of interfacing choices.

We have identified potential semantic pitfalls that need to be addressed; we also plan to further expand the modeling instruments. We envision a reliable Model Driven Engineering toolchain with an expanding action node library that makes efficient cross-platforms designs a less daunting task.

ACKNOWLEDGMENTS

The authors would like to thank the European Union, European Regional Development Fund (ERDF) and the federal state of Baden-Wuerttemberg, Germany, for the support of our research.

REFERENCES


