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. Finally, we present a case study for cross-platform system implementation.

I. INTRODUCTION

The emergence of hybrid embedded systems, which combine conventional microcontrollers with specialized hardware like Digital Signal Processors and reconfigurable devices, adds new complications to an already complex and error-prone 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 framework in Section V. An infrastructure for consistent cross-platform modeling and a case study are presented in Section VI. Section VII concludes and mentions planned future work.

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 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 [3], the implementation of automata with state diagrams [4][5] or a combination of both [6]. Schatkowsky et al. [7] have used sequence and state diagrams to generate Handel-C. Fernandes et al. [8] have examined generating VHDL from Petri nets [9], 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

A typical Model-Driven Engineering flow is show 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-A 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 meta-model. 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. 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* 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. In contrast, activity nodes denote an instance of another activity diagram; they are distinguished from action nodes through a small fork-symbol in the corner.
- **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.

IV. THE DMOSES DEVELOPMENT PROCESS

The DMOSES process has been designed to guarantee embedded systems models and specifications that lead to stable and predictable 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 [10], which brings together modeling, model transformation and code generation in one place

A. Profile extensions for determinism

UML activities as shown in Figure 3a leave semantic ambiguities that can lead to race conditions. 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.
These extensions assure complete determinism and avoid unpredictable behavior in complex model implementations. A different hardware platform does not lead to different behavior.

B. 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:

```cpp
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.

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 functional stages take longer than one cycle for their operations. We have decided to substitute 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)`. 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. In reality, except for a few specific cases where the Valid signal is further constrained by UML semantics, the output FIFO is not implemented, since one input FIFO shows the same behavior as two FIFOs in series.

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 signals 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. 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. The data is handled with a width of 64 bit, corresponding to the underlying double precision floating point type. Further, 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, to define storage size and access mechanisms to be synthesized. We introduce an enumeration type called classtype and the properties cwidth and cdepth. Classtype’s set of defined values consists of names of data types needed to implement models. The intention is to stay consistent with data classes defined in the model and used in the C++ implementation. Table I gives examples.
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 `cldepth` return value.

**F. Extending FIFOs for Object Mode transfers**

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 multiword 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.

**VI. MODELING AND GENERATING HYBRID SYSTEMS**

**A. UML modeling of hybrid systems**

DMOSES offers the possibility to assign discrete function blocks of a modeled activity to different execution devices. 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. 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 8. 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.

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. Case Study

As a first cross-platform application, we have designed the configurable audio filter path displayed in Figure 9. We deliberately chose a low-computation application to keep valid bandwidth issues on the sidelines initially. 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-tap Finite Impulse Response filter. The filtered audio is then replicated by a fork node into three data streams, two for the output audio components and one that is consumed by a 1024-point Fast Fourier Transform. The resulting Fourier coefficients are handed over into the CPU domain, where they are combined into 16 double-logarithmic spectral values to be displayed as a bargraph on an onboard LCD display. The assembled coefficient object is sent over to the logic side and into the FIR filter.

VII. 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.

Planned future work includes the implementation of UML state diagrams in combination with activities, the automatic generation of testbenches for debugging as well as profiling, and profile-based area, speed and bandwidth optimizations.

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