On Model-Driven Engineering of Reconfigurable Digital Control Hardware Systems

INTRODUCTION

A digital control system is made up of a set of computer-based components whose mission is to coordinate, manage, and command the operation of another system. The controlled systems include navigation systems for both terrestrial and aerial vehicles, specific-purpose industrial machinery, automated chemical and thermal processes, and other man-made artifacts and dynamic processes. The algorithms that perform these control operations are usually implemented in software running on special microprocessors known as microcontrollers, which are present in plenty of devices, vehicles, and machines that are common in our every-day live.

Figure 1 illustrates the organization of a generic closed-loop digital control system. It is built around a digital computer that receives, as input, a discrete signal *e(kT)* that is the difference between the sampled versions of both the system's input (*r(t)*) and the controlled system's (the plant) output (*y(t)*). The computer produces another discrete (*u(t)*) that, by means of a D/A, is converted to a continuous signal to control the plant.

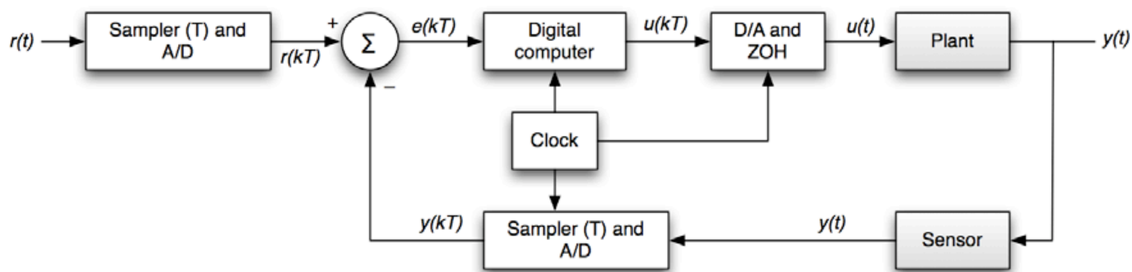

Figure 1. General block diagram of a digital control system.

According to [15], there are a number of capabilities demanded by modern software used by both civilian and military control systems. These capabilities are driven by modern technological trends and include the following:

- **Adaptability:** Depending on changes in the environment, changes in external inputs and/or the behavior of the system, the control software modifies its own behavior in a quick and seamless manner and without affecting the system's stability.
- **Extensibility:** The ability to include software components for new technologies (like control algorithms, sensors, or communication protocols) without redesigning the architecture of the control system software.

- **Interoperability:** Current control systems are distributed among a number of heterogeneous hardware/software platforms. These systems must be able to communicate with each other using different protocols and communication links, and meet a number of strict constraints related to response time and reliability.

This sort of software reconfigurability demands that the software architecture of the control system is flexible enough to support on-line changes without compromising the integrity of the system. There are a number of software technologies that might be useful to implement the previous capabilities, like component-based development [8] and self-adaptive software [13].

Alternatively, the control algorithms and operations performed by software may be implemented by a digital hardware system when strict performance and real-time requirements are a must. In this case, the capabilities listed above are still necessary features that must be supported in some manner by hardware-based control systems. This time we can take advantage of the current advances in reconfigurable hardware to allow a hardware-based control system to load different configuration bit-streams on the fly to perform a different operation.

Designing any kind of digital hardware system, whether dedicated or reconfigurable, is not a straightforward task; there are a number of complexities in the development process that must be addressed properly. The mechanisms to implement dynamic reconfiguration are specific of the hardware platform and are generally invisible to the end user and, to some extent, to the designer. However, the developer does deal with the challenge of implementing the numerous functional requirements of the system in shorter periods of time by meeting a number of operational constraints. This chapter discusses about the use of a model-driven principle to describe the functionality of hardware-based control systems at higher levels of abstractions, and to automatically synthesize a hardware implementation of those descriptions for either an Application-Specific Integrated Circuit (ASIC) or a reconfigurable platform.

In the following sections we focus our attention on the problem of digital design complexity and its direct consequence: the productivity gap phenomenon. We then describe modern approaches to raise the level of abstraction in the design process to increase productivity. Next, we describe how the model-driven paradigm for software development might help during the development process of digital hardware-based control systems by describing how to model difference equation block diagrams using a modeling language, and how these models could be implemented.

DESIGN COMPLEXITY

A computer-based system is a combination of hardware and software that implements a set of algorithms to automate the solution to a number of problems. Computer design technology transforms the designer’s ideas and objectives into a number of representations describing software modules and hardware components that can be tested and manufactured [14]. The design process is not straightforward; the developers always deal with the problem of alleviating the complexity of their designs to develop high-quality products within rigid time constraints. This problem arose as a consequence of the steady evolution of technology and the constant demand for new functionality.

Computer-based systems are not becoming easier to design as time goes by; on the contrary, the advancement of development and manufacturing technologies, and the need to meet new usage demand encourage the development of devices incorporating more and more functionality. This is a list of some key functionality aspects that have demanded attention from hardware/software engineers during the last years:

- **Communication.** A large number of computing devices must be connected to the Internet nowadays. This can be done by means of either broadband wired Ethernet, or local wireless WiFi, or global wireless WiMax, or a cellular telephone network. It is common that a single device supports a set of the previous standards, which makes it more flexible but more challenging to design efficiently.
- **Security.** Several computer systems must implement mechanisms to cipher information, authenticate users, guarantee the integrity and confidentiality of data, and protect against a number of attacks. Usually, it is needed to cipher lots of information in a short amount of time, so hardware accelerators that increase performance of encryption algorithms are pervasive.
- **Power management.** Computers must run operating systems that switch idle hardware components to an operation mode that consumes less power when needed. Thus, the hardware components must implement a set of power states, each corresponding to an specific requirement of power consumption.
- **Multimedia processing.** A wide range of mobile devices; every single video game console; and some desktop computers, workstations, and servers must execute software to visualize video streams and files, produce high definition sound, process 2D images, and render 3D images. In almost every case, the software is aided by hardware accelerators to increase performance of the algorithms that demand more computing power.
- **Fault tolerance.** High performance mission critical servers and supercomputers must incorporate algorithms to detect and correct errors, or, if the errors are uncorrectable, prevent them from spreading and compromising the whole system. In addition, those systems must implement algorithms to provide information redundancy and protect sensitive information as much as possible.

When designing the digital hardware of a computer-based system the developers must deal with the challenge of meeting a number of design constraints while implementing the required functionality. This is a list of the most common restrictions in hardware engineering:

- **Higher performance.** Very often it is not enough to solve a problem but to solve it fast. The functionality of the system must be implemented using algorithms that solve the corresponding problem with a high degree of performance. Performance is measured in different ways depending on the application.
- **Power consumption efficiency.** Portable devices must meet their operational requirements while providing long battery life. In this case the systems must be designed with the goal of consuming less power as possible.
- **Low area.** When a large number of resources (such as transistors) are not available, the developer must conceive small designs that reutilize a component iteratively until completion.

It is not possible to optimize all of these parameters at the same time because some of them contradict to each other; thus, the designer must make trade-offs between them. For instance, an

area-efficient hardware implementation of a block cipher algorithm for 3G cellular communications that reuses a basic function block iteratively until completion is able to encrypt information at a rate of 317.8 Mbps. [3], whereas a high-performance hardware implementation of the same algorithm that requires 6.3 times more hardware resources has a performance of 5.32 Gbps.

It is not possible to stop the evolution of technology or to prevent computer-based systems from implementing more and more functionality over time and becoming more complex. Hardware and software engineers are condemned to face the challenge of designing products that implement lots of functionality, while meeting difficult constraints, in shorter periods of time.

PRODUCTIVITY GAP

At this point we focus our attention on the challenging process of designing digital hardware systems that implement control systems, and propose a new method to improve productivity during their functional description phase. These functional descriptions can be tested and implemented in semiconductor platforms like Application-Specific Integrated Circuits (ASIC) or Field Programmable Gate Arrays (FPGA).

In spite of having more resources to design with, design complexity imposes serious limits to the ability of designers to develop high quality products that fully meet their requirements in a short period of time; that is, to their productivity. The productivity gap is the challenge that arises when the number of available transistors grows faster than the ability to meaningfully design with them [5, 14]. Flynn, et al. [5] illustrates the considerable separation between the exponential increase in the number of transistors per chip along the last 28 years and the increase in design productivity along the same period of time.

ABSTRACTION LEVELS

An effective way to alleviate design complexity and to reduce the productivity gap during the design of computer-based systems is to raise the level of abstraction at which developers carry out their activities. The goal is to design correct systems faster by making it easier to check for, identify, and correct errors.

The raise in the level of abstraction has been done many times in the past for both software and hardware development. The following is a brief description of the different levels of abstractions that have been used throughout the last decades to design digital hardware systems:

- **Transistor-level design.** The first solid-state computers were built using discrete transistors and other electronic components. These machines were relatively complex systems with little memory and consumed several kilowatts of power. As new architectural techniques to increase performance arose the hardware became so complex that turned design with discrete components impractical.
- **Schematic design.** When Medium-Scale Integration (MSI) and Large-Scale Integration (LSI) integrated circuits became pervasive the discrete components that made computer modules up were gathered together and encapsulated into a single silicon die. This allowed a high degree of miniaturization because now the hardware was described as a set of schematics specifying the interconnection of a number of integrated circuits.

- **Register-Transfer Level (RTL) design.** The behavior of a circuit is defined in terms of a flow of signals (data transference) between hardware registers, and the logical operations performed on those signals. This level of abstraction employs hardware description languages (like VHDL and Verilog) to create a more manageable description of a system. This representation can be transformed into a description of the electronic components that make up the system and the interconnections between them (netlist), which can be implemented in a Very Large Scale Integration (VLSI) silicon platform.
- **Electronic System Level (ESL) design.** The functionality of a digital hardware system is described by means of higher-level languages (some of them built from languages like C and Java) and graphical tools. The main goals are to achieve a high degree of comprehension and reutilization of the functional descriptions, and to fully automate the implementation process [2].

In spite of their advantages to describe the functionality of digital hardware systems at higher levels of abstraction, some ESL technologies have important drawbacks that prevent them from being used to design some kind of devices, like low-power embedded hardware, efficiently. There is a strong need for very high-level design languages and tools that are customized for different application domains and help to alleviate design complexity. ESL is a recent research trend that has been neither fully explored nor fully standardized [4]; there is still room for important innovations.

At the ESL there are lots of similarities between the process of functional description of digital hardware systems and the process of software development[1]. Thus, we can think of taking advantage of the recent advances in software engineering, like the Model-Driven Engineering paradigm (MDE) [7], to raise the level of abstraction even further, alleviate design complexity, increase reuse of existing designs, and automate the production of representations at lower levels of abstraction.

DESIGN FLOW BASED ON MODEL-DRIVEN ENGINEERING

MDE is a recent effort intended to raise the level of abstraction further when developing software systems. This approach is about conceiving the solution to a problem as a set of models expressed in terms of concepts in the problem's domain space, those that the designers and/or customers know very well, instead of concepts in the solution space, those related to software and hardware technologies. The intention is to translate the designer's models into the appropriate implementation for a specific platform[2], and to hide the complexities of such platform's hardware and software. The motivation to this paradigm is to improve both short-term productivity (increase functionality) and long-term productivity (lengthen longevity) during the development process [1]. Kent [7] describes a set of general aspects that characterize MDE: high-level modeling, multiple modeling dimensions, processes, tools, transformations, and meta-modeling.

The Model-Driven Architecture (MDA) technology, proposed by the OMG, is a realization of the MDE paradigm. It attempts to define an MDE-based framework using the OMG's standards,

[1] However, the divergence point is at the moment of implementing the digital hardware system's descriptions into silicon.
[2] In this context a platform is defined as a combination of hardware and software technologies.

most notably the Unified Modeling Language[3] (UML), to improve the process of software development [6, 10]. MDA separates a software system's functionality and requirements specification from the implementation of such functionality on a specific combination of hardware and software technologies (platform). The benefit of MDA is twofold: first, to enable the implementation of the same functionality on multiple computer platforms by means of automatic transformations; second, to allow the integration of different software systems by relating the corresponding models.

MDA categorizes models according to their level of abstraction, the only dimension in this MDE realization. A high-level functional model is called Platform-Independent Model (PIM), and its implementations, each for a particular platform, are called Platform-Specific Models (PSM). In a complete scenario, the designer should be able to create, execute, test and interchange PIMs before generating the corresponding PSMs. Figure 2 illustrates these concepts when applied to a number of software technologies based on Java, .NET, and CORBA.

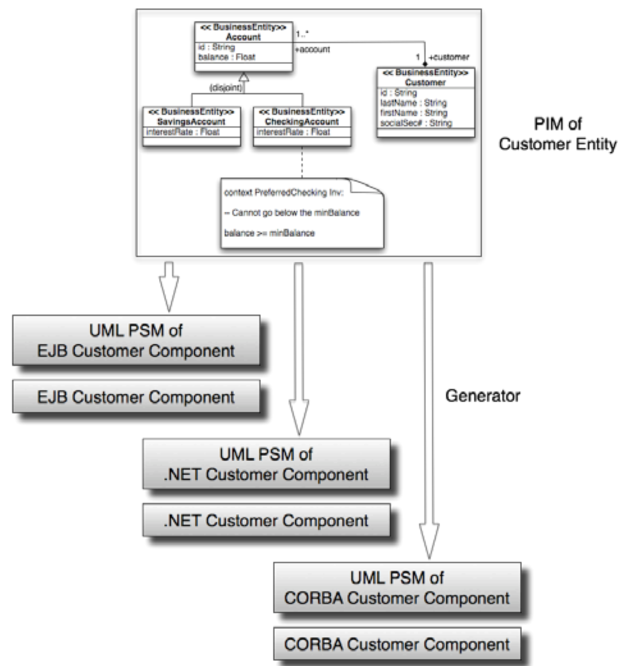

Figure 2. The MDA approach: transformation of PIMs into PSMs.

The following is a description of the main principles our proposed MDA-based design flow relies on:

- UML 2.0 [11, 12] has two main advantages that make it the best choice for building the modeling languages for our proposal: first, it was designed with the purpose of being extended and customized; second, its graphical nature allows a better comprehension degree than a textual language.

---

[3] UML is a graphical notation that has been used during the last decade to model and document object-oriented software systems. It allows specifying both the structural and the behavioral aspects of a software system. The last major revision of the language is the UML 2.0 revision.

- A Domain-Specific Modeling Language (DSML) is necessary for the designer to describe functionality more effectively because it allows the use of terms and abstractions within the problem domain, instead of letting the designer deal with the awkward details belonging to the implementation domain. Mernik, et al. [9] state that "by providing notations and constructs tailored toward a particular application domain, the domain-specific languages offer substantial gains in expressiveness and ease of use compared to general-purpose languages for the domain in question, with corresponding gains in productivity and reduced maintenance costs". This DSML is conceived as an extension to the UML 2.0 (a profile).
- The functional description in DSML (a PIM in the jargon of MDA) is, in the long run, automatically transformed into a lower level representation in VHDL that, in turn, can be implemented on either an ASIC or an FPGA. In between these two representations it is possible to introduce another UML-based description whose modeling constructs have the same semantics as the language constructs of VHDL, that describes the same functionality as the DSML model, and that allows the generation of VHDL code from it. The purpose of this description (a PSM in the jargon of MDA) is twofold: first, to ease the transformation process from DSML to VHDL by partitioning it into two simpler ones; second, to provide the user with a UML-based blueprint of the system that is closer to the final VHDL representation and allows a better comprehension of such VHDL code. The designer might be able to fine-tune this model before generating VHDL code from it; however, this is considered an inappropriate practice by some people.

Figure 3 illustrates how the previous principles define a design flow similar to that in Figure 2. We can observe both the PIM and PSM abstraction levels along with the VHDL code that is the ultimate result of the whole design approach. Figure 3 also shows the procedure to define the DSML and the UML 2.0 profile for VHDL. The two transformation algorithms that map high-level descriptions into lower level representation and that generate VHDL code are indicated in the figure as well.

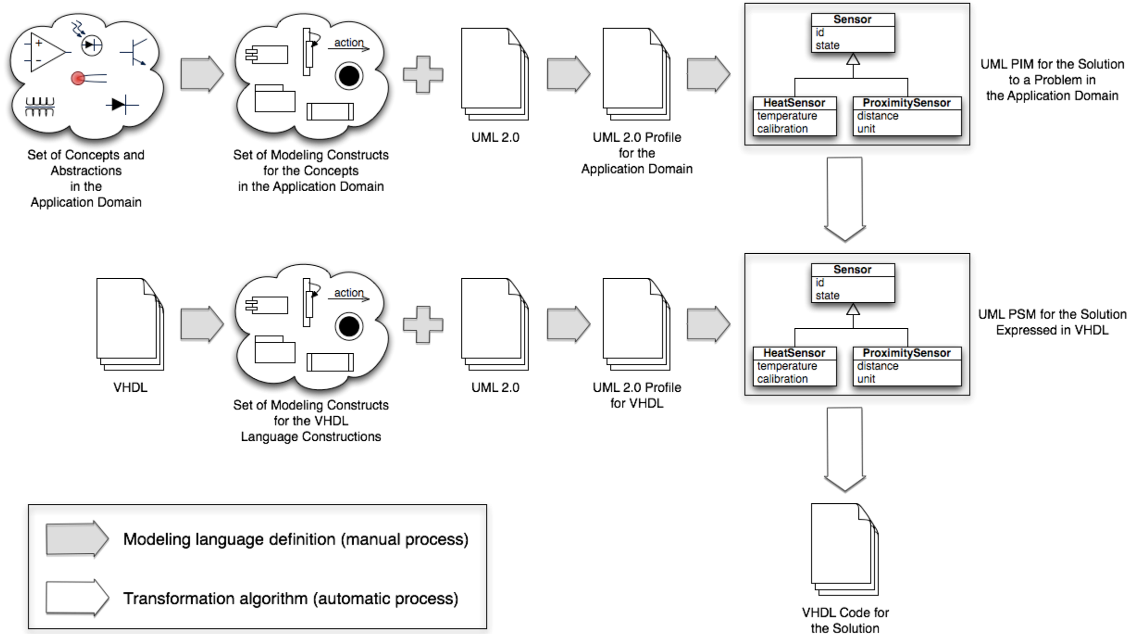

Figure 3. Proposed MDA-based design flow.

MODELING OF DIFFERENCE EQUATIONS

During the *k*-th sample, the digital computer has received the inputs $e_0, e_1, e_2, \ldots, e_k$ and has produced the outputs $u_0, u_1, u_2, \ldots, u_{k-1}$; it computes the current output u_k by means of a difference equation, a linear combination of *n* previous inputs and *m* previous outputs:

$$u_k = -a_1 u_{k-1} - a_2 u_{k-2} - \ldots - a_n u_{k-n} + b_0 e_k + b_1 e_{k-1} + \ldots + b_m e_{k-m} \quad (1)$$

A specific example of a difference equation that can be programmed in the digital computer to perform a practical operation is the trapezoid rule used for numerical integration with a sample period of *T*:

$$u_k = u_{k-1} + \frac{T}{2}(e_k + e_{k-1}) \quad (2)$$

By using the feedback line, the control system computes each of the e_k inputs provided to the computer, which stores them along with its previous outputs u_k and uses them during execution of Equation 2. Thus, with every step the computer gets closer to the estimation of the integral for a continuous function *e(t)*. Figure 4 illustrates the block diagram corresponding to this trapezoid rule difference equation; it denotes the delays in both the input and output signals, the arithmetic operations needed to compute the output signal, and the coefficients required.

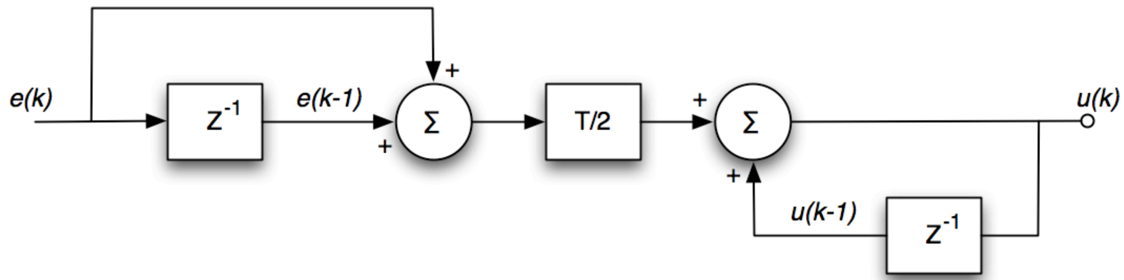

Figure 4. Block diagram for the trapezoid rule.

In this section we describe the first step of a process to implement simple difference equations into the computation engine of a digital control system by means of a model-driven development approach. This process departs from block diagrams like the one in Figure 4 and synthesizes either software to program a control system's digital computer, or VHDL code for a reconfigurable digital hardware platform for control applications. The block diagrams are first built using an extension to UML 2.0 Activity Diagram and then transformed automatically into software or VHDL code.

The UML 2.0 Profile

Unlike earlier versions of the UML specification, recent versions provide a formal model of the language's semantics called meta-model (model of a model). The meta-model contains a set of meta-classes that define the UML modeling elements, and describes the relationships between meta-classes that indicate how the modeling elements are assembled together to build the user's UML models of a system.

A profile is an extension mechanism for UML to get dialects that customize the language for particular platforms or application domains. Profiles are made up of stereotypes that extend particular meta-classes; tagged values that define additional attributes for the stereotype; and restrictions that specify rules, pre- and post-conditions for the extended modeling elements.

The UML Activity Diagram is used to describe procedural logic, business processes, and work flows. This diagram is conceptually similar to a flowchart, but differs from it in its ability to describe parallel behavior and model both control and data flows; these two distinctions make this kind of diagram the most adequate one to model the data flows and the operations required by the block diagrams representing difference equations in a correct manner. The Activity Diagram's modeling elements include: actions representing behavior execution, input/output pins working as connection points for data or control going in or out the actions, edges indicating the flow of either data or control, decision elements to choose one out of several paths, fork nodes to initiate parallel paths, asynchronous signaling mechanisms, and constructions to elaborate a hierarchy of sub-activity diagrams. Our profile's stereotypes extend the meta-classes of these modeling elements to derive specialized modeling constructs representing the operations required by difference equations.

Figure 5 shows a UML 2.0 class diagram illustrating the hierarchy of meta-classes from which we derive our profile's stereotypes, which are indicated by the shaded class boxes. A stereotype is a

meta-class labeled with the keyword «stereotype» that is derived from an existing meta-class within the meta-model with the intention of extending its behavior. The stereotype's attributes shown in Figure 5 are called tagged values and define properties for the new modeling element that are additional to the ones it inherits from its parent meta-class. Our profile derives several stereotypes from the `Action` meta-class to model the operations that are common to difference equation block diagrams; it also derives a stereotype from the meta-class `ObjectFlow` to model edges transmitting data; and it also derives a stereotype from the meta-class `ForkNode` to either distribute data along two or more different paths, or to partition an n-bit datum into several data of different lengths. Table 1 provides a brief description of the semantics of the profile's modeling elements, or stereotypes. An UML Activity Diagram built using this profile is called a UML Block Diagram.

Modeling Element	Description
delay	Models a delay operation, denoted in block diagrams as Z^{-1}.
MUL	Models a multiplication by a coefficient operation.
ADD	Models an addition operation.
dl	Models a data line that transfers intermediate results between modeling elements.
y	Allows sending an intermediate result along two or more data lines.

Table 1. Description of the modeling elements that make up the UML profile for difference equations.

Building UML Block Diagrams

Figure 6 shows the UML Block Diagram corresponding to the block diagram in Figure 4. The UML Block Diagram is enclosed within a main activity called TrapezoidRule having an integer input port to receive the sequence e_k, and an output port to send the output sequence u_k. There are two actions (`delay_1` and `delay_2`) that perform the delay of the incoming datum, and this is known because these actions are denoted by the «delay» stereotype. Similarly there are actions that perform integer addition, like `add_1` and `add_2`, denoted by the «ADD» stereotype, and actions that perform integer multiplication, like `T/2`, denoted by the «MUL» stereotype. Every operation is a special kind of action of the UML Activity Diagram; thus, the operations have pins attached to them for input and output data.

A very important aspect of the UML Block Diagram is the correct setting of each modeling element's attribute. Figure 5 indicates that each delay modeling element has an attribute called `n`, whose default value is one, that indicates the delay steps performed by the action; in the UML Block Diagram in Figure 6 there is no need to set this attribute to a different value. The integer multiplication modeling element has two attributes: `coefficient`, whose default value is one, stores the value of the coefficient the input datum is multiplied by; and `length`, whose default value is 32, indicates the length in bits of the value in the attribute `coefficient`. In the diagram in Figure 6 there is no settings for the multiplication action that is different from the default values; do not confuse the action's label `T/2` with the value of its coefficient. Similarly, the integer addition modeling element has only one attribute related with its precision. Depending on the specific UML modeling tool, the values of the attributes can be shown along with the corresponding modeling element or not.

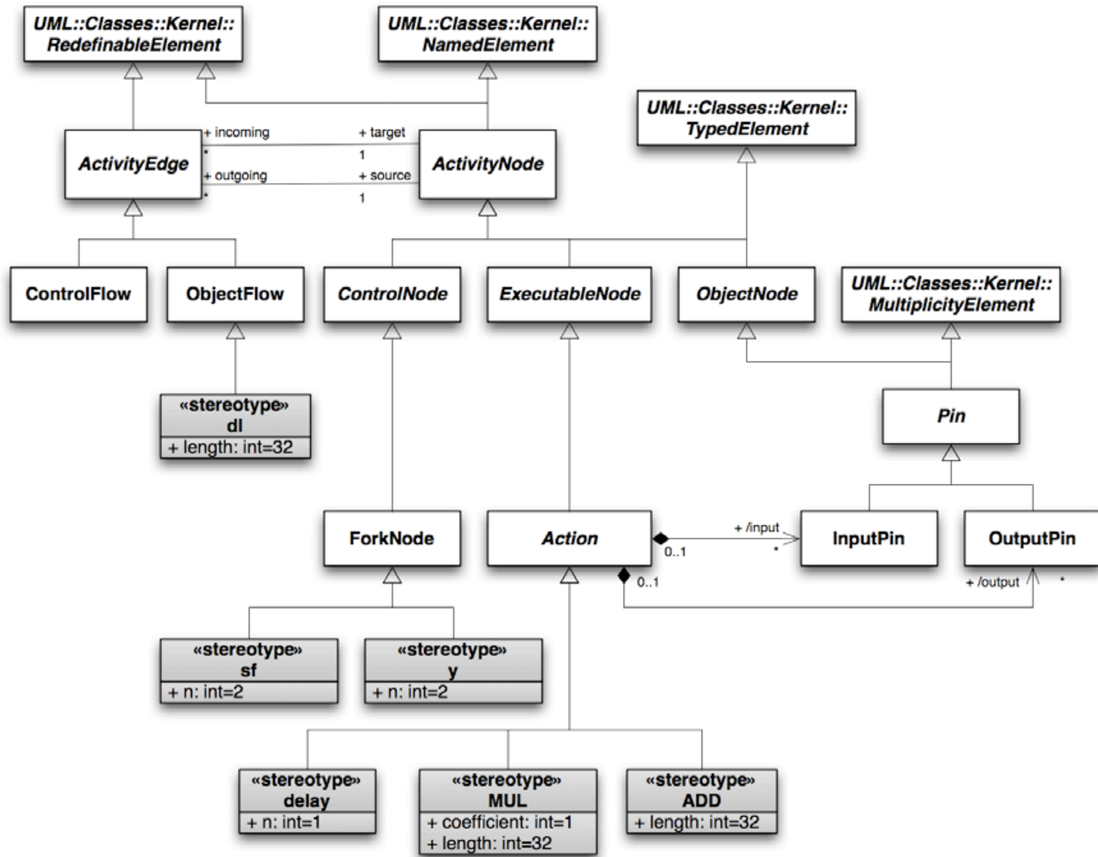

Figure 5. Fragment of the UML 2.0 meta-model for Activity Diagrams extended with the stereotypes that make up the profile for modeling difference equation block diagrams.

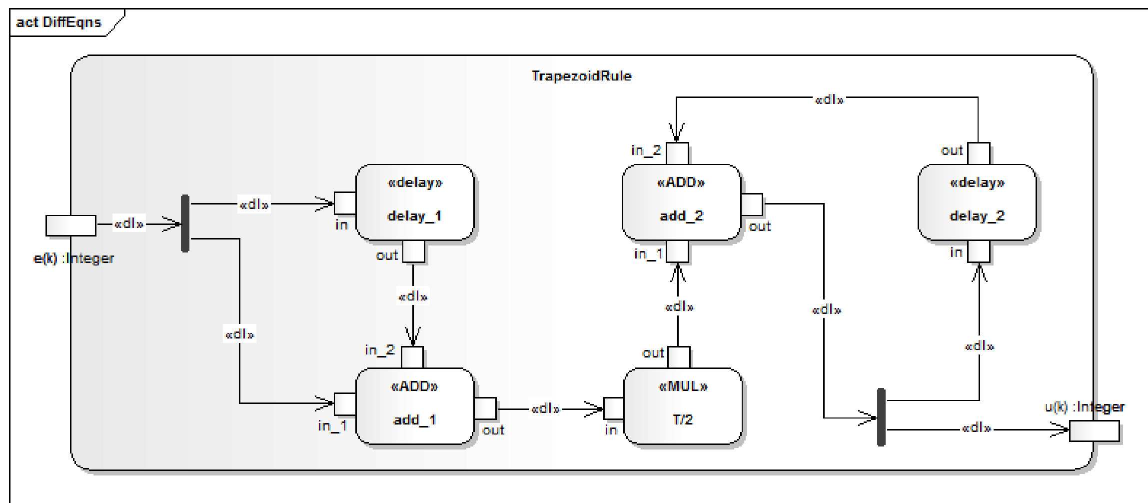

Figure 6. The UML Block Diagram for the Trapezoid Rule Difference Equation.

The data lines in the UML Block Diagram are modeling elements that are instances of the «dl» stereotype; they connect actions to one another by means of the actions' pins. The fork nodes that

appear in the diagram allow sending the incoming data flow along two, or more, different data paths that are semantically parallel.

Synthesis of UML Block Diagrams

In this sub-section we provide a number of ideas about what we can do with the UML Block Diagrams, about where we can go to with these representations. To do this we must take into account the ideas discussed previously about the MDE paradigm, and its realization in the MDA technology. We will link the UML Block Diagrams to the design flow in Figure 3.

A UML Block Diagram, like the one in Figure 6, can be stored and exchanged by multiple UML tools by using a standard representation called XML Metadata Interchange (XMI). One can develop a number of software tools that take these XMI textual representations and work with them to transform them into a new representation at lower levels of abstractions.

On the one hand, the models can be transformed into software for a microcontroller under the following rules:

- The delay operations are mapped to memory accesses to retrieve the proper sequence element.
- The coefficients of the multiplication actions can be stored in the microcontroller's registers.
- The addition and multiplication actions can be synthesized to a number of multiplication and accumulation operations whose operands come from the registers.
- Advanced processor micro-architectures might implement instruction-level parallelism mechanisms and the synthesis tools can produce code that exploits these micro-architectural features.

These rules are for the case in which the synthesis process produces assembly language code. Alternatively, the synthesis tools might generate C code that can be compiled by the microcontroller's specific compiler, which is a more portable solution.

On the other hand, a different synthesis tool might be able to build special-purpose hardware architectures from the UML Block Diagrams' XMI representation. In this case, the result of the synthesis process is either a functional description in a language like VHDL, like in the design flow described previously, or a netlist for a specific semiconductor platform. The functionality of the customized hardware architecture is fixed but it can be constructed to exploit parallelism and achieve higher performance. When implemented in platforms like FPGAs, it is possible to implement dynamic reconfiguration mechanism to change the functionality of the computing engine in the digital control system according to its environment.

CONLUSIONS

In this chapter we reviewed the motivation for design methodologies and tools at higher levels of abstractions to develop digital control systems in hardware; we focused our attention in current ESL. We examined the causes of the productivity gap phenomenon and suggested that the latest trends in software engineering, like the model-driven engineering paradigm, could have a positive impact on the improvement of productivity. We examined how the block diagrams for discrete difference equations could be modeled using a dialect of UML 2.0, and provided some hints on the processes required to generate descriptions that are closer to implementation from such higher

level models. We expect that this kind of high-level graphical modeling languages and automatic synthesis tools become pervasive in the near future, so they allow the designer to tackle the ever-increasing complexity of modern digital control systems in a more productive manner.

REFERENCES

1. Atkinson, C., & Kühne, T. (2003). Model-Driven Development: A Metamodeling Foundation. *IEEE Software*. Volume: 20, Issue: 5 (pp. 36-41).

2. Bailey, B., Martin, G., & Piziali, A. (2007). *ESL Design and Verification. A Prescription for Electronic System-Level Methodology*. San Francisco, CA: Morgan Kaufmann Publishers.

3. Balderas-Contreras, T. (2004). *Hardware/Software Implementation of the Security Functions for Third Generation Cellular Networks*. Master Thesis. INAOE. MEXICO.

4. Densmore, D., Sangiovanni-Vincentelli, A., & Passerone, R. (2006). A Platform-Based Taxonomy for ESL Design. *IEEE Design and Test of Computers*. Volume: 23, Issue: 5 (pp. 359-374).

5. Flynn, M. J., & Hung, P. (2005). Microprocessor Design Issues: Thoughts on the Road Ahead. *IEEE Micro*. Volume: 25, Issue: 3 (pp. 16-31).

6. Frankel, D. S. (2003). *Model-Driven Architecture. Applying MDA to Enterprise Computing*. Indianapolis, IN: Wiley Publishing, Inc.

7. Kent, S. (2002). Model Driven Engineering. *Lecture Notes in Computer Science*. Volume 2335/2002 (pp. 286-298).

8. Kozaczynski, W., & Booch, G. (1998). Component-Based Software Engineering. *IEEE Software*. Volume: 15, Issue: 5 (pp. 34-36).

9. Mernik, M., Heering, J., & Sloane, A. M. (2005). When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*. Volume: 37, Issue: 4 (pp. 316-344).

10. Miller, J,. & Mukerji, J. (2001). Model Driven Architecture. Document number ormsc/2001-07-01. OMG.

11. Object Management Group. (2007). *OMG Unified Modeling Language (OMG UML) Infrastructure V2.1.2*. OMG Document Number: formal/2007-11-04.

12. Object Management Group. (2007). *OMG Unified Modeling Language (OMG UML) Superstructure V2.1.2*. OMG Document Number: formal/2007-11-02.

13. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimhigner, D., Johnson, G., Medvidovic, N. Quilici, A., Rosenblum, D.S., & Wolf, A.L. (1999). An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*. Volume: 14, Issue: 3 (pp. 54-62).

14. Semiconductor Industry Association (2007) *International Technology Roadmap for Semiconductors; Design Chapter.*

15. Willis, L., Kannan, S., Sander, S., Guler, M., Heck, B., Prassad, J.V.R., Schrage, D., & Vachtsevanos, G. (2001). An Open Platform for Reconfigurable Control. *IEEE Control Systems Magazine*. Volume: 21, Issue: 3 (pp. 49-64).