A UML 2.0 Profile to Model Block Cipher Algorithms

Tomás Balderas-Contreras, Gustavo Rodriguez-Gomez, and René Cumplido

National Institute of Astrophysics, Optics and Electronics
Computer Science Department
Luis Enrique Erro 1, 72840 Santa María Tonantzintla, Puebla, Mexico
{balderas,grodrig,rcumplido}@inaoep.mx
http://ccc.inaoep.mx

Abstract. Current mobile digital communication systems must implement rigorous operations to guarantee high levels of confidentiality and integrity during transmission of critical information. To achieve higher performance, the security algorithms are usually implemented as dedicated hardware functional units attached to the main processing units of the embedded communication system. To save hardware resources, the designer usually performs a number of manipulations in the cipher algorithm lying at the core of the confidentiality and integrity operations to implement a simplified version of it that is suitable to be efficiently used in an embedded environment. This paper describes an extension to UML 2.0 to model the structure of contemporary block cipher algorithms, with the ultimate goal of synthesizing representations in a hardware description language from these models according to a model-driven development principle. This automated process should alleviate design complexity and increase the productivity of the developer during experimentation with different design alternatives.

Key words: Block cipher algorithm, UML 2.0 profile.

1 Introduction

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 designers ideas and objectives into a number of representations describing software modules and hardware components that can be tested and manufactured [11]. 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. There are a number of functionality aspects that have demanded attention from hardware/software engineers during the last years: communication, security, power management, multimedia processing, and fault tolerance.

When designing the digital hardware of a computer-based system the developers must deal with the challenge of making a trade-off between a number of design requirements, that can not be optimized all at the same time, while implementing the desired functionality. The digital hardware system must usually achieve a *high level of performance*, its operation should be *efficient in terms of power consumption*, and, when a large number of hardware resources is not available, its *circuitry must be small* and reutilize a component iteratively until operation completion. 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. In this document we focus our attention on the process of developing the digital hardware sub-system of a whole computer-based system.

1.1 Productivity gap

In spite of having more resources to design with, design complexity imposes serious limits to the ability of hardware 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 [11]. Flynn, et al. [6] 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.

1.2 Abstraction levels

An effective way to alleviate design complexity and to reduce the productivity gap during the design of digital hardware 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 first solid-state computers were built using *discrete transistors and other electronic components*, consumed several kilowatts of power, and became more complex to design as advanced architectural techniques to increase performance arose. Medium-Scale Integration (MSI) and Large-Scale Integration (LSI) integrated circuits that encapsulated whole computer modules within single dies allowed to design digital hardware systems as a set of *schematics* specifying the interconnection of a number of integrated circuits. Later, the behavior of a circuit started to be defined in terms of a *flow of signals (data transference) between hardware registers and the logical*

operations performed on those signals using hardware description languages like VHDL and Verilog. This representation was transformed into a description of the electronic components that made up the system and the interconnections between them (*netlist*), which could be implemented in a Very Large Scale Integration (VLSI) silicon platform like an Application-Specific Integrated Circuit (ASIC) or a Field Programmable Gate Array (FPGA). The current *Electronic System Level* (ESL) design trend proposes the use of high level languages, derived from languages like C and Java for instance, to describe the functionality of a digital hardware system and tools to automate the implementation process [2]; thus achieving a higher degree of comprehension and reutilization of the functional descriptions.

1.3 ESL and UML

At the ESL there are lots of similarities between the process of *describing the functionality of digital hardware systems* and the process of *developing software*. A research effort is needed to determine if we can take advantage of the recent advances in software engineering, like the *Model-Driven Engineering* (MDE) paradigm [7], to raise the level of abstraction even further, increase productivity, alleviate design complexity, exploit reuse of existing designs, and automate the production of representations of digital hardware systems at lower levels of abstraction.

Riccobene, et al. [10] propose a UML 2.0 profile containing the constructs of the SystemC language to allow the designer to build diagrams instead of writing code. Björklund, et al. [3] describe the use of an intermediate representation called SMDL to transform general-purpose state machine diagrams to VHDL. While these two proposals synthesize hardware description language code from UML, they do not customize UML to an application domain to allow the developer to describe a system in terms of the concepts he/she knows instead of the concepts of the implementation language or hardware platform.

This paper describes an extension to UML 2.0 [8] that includes abstractions to model the structure of block cipher algorithms with the purpose of them being implemented in hardware. The profile should allow the designer to modify the structure of the algorithm, without altering its operation, to design a hardware implementation that meets the required trade-offs between performance and resource consumption. 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 164.45 Mbps. [4], whereas a high-performance implementation of the same algorithm that requires 8.05 times more hardware resources (slices in a Virtex-E FPGA) has a performance of 5.32 Gbps [5]. This profile will be a crucial component of a model-based design flow that will transform a high level description in UML to a lower level VHDL representation that could be implemented in either an ASIC or a FPGA platform.

This document is organized as follows: section 2 documents the proposed profile to model block cipher algorithms, section 3 illustrates the application of the profile in a practical case of study, and section 4 concludes.

2 The Block Cipher Profile

Current versions of UML include a formal definition of the language's constructs and abstract syntax that is called meta-model (a 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 by the user to build the UML models of a system. A *profile* is an extension mechanism for UML, a kind of dialect that customizes 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.

2.1 Block ciphers

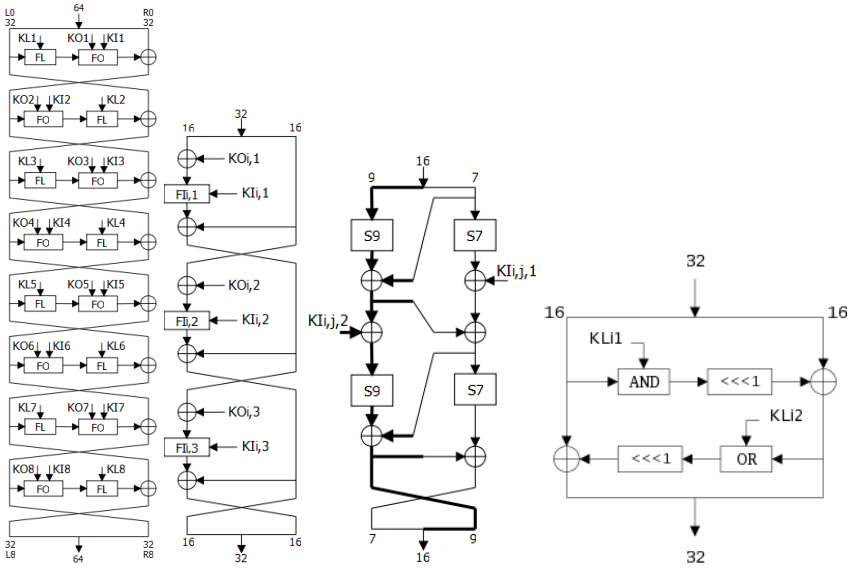
A *block cipher* is an algorithm that unvaryingly transforms a fixed-length group of bits, called plaintext block, into a different group of bits, called ciphertext block, under the control of a symmetrical secret key. The algorithm carries out the inverse process when it receives both the ciphertext block and the secret key as inputs.

Most block ciphers employ simple operations like bitwise logical operations (and, or, xor), shifts and rotations, n-bit substitution functions (referred to as S-Boxes), arithmetic operations, and permutations in an iterative manner until completion. The structure of these algorithms is usually shown as an iterative Feistel network, an structure whose iterations are called rounds and perform an internal round function.

As an example consider the KASUMI block cipher, illustrated in the block diagrams in Figure 1, used nowadays to implement security functions, like confidentiality and integrity, in modern 3G cellular communication networks [1]. Each of the eight rounds of KASUMI's Feistel network carries out a pair of operations called FL and FO, where FO is, in turn, a Feistel network with three rounds, each performing a function called FI that is made up of two seven-bit input S-Boxes (S7) and two nine-bit input S-Boxes (S9). The informal block diagram notation frequently used to describe this kind of algorithms does not represent either a digital circuit schematic or an UML diagram.

2.2 Defining the Block Cipher Profile

The UML Activity Diagram is used to describe procedural logic, business processes, and work flows. This diagram is conceptually similar to a flowchart, but



(a) Main Feistel network (b) FO function (c) FI function (d) FL function

Fig. 1. The components and full structure of the KASUMI block cipher (from [1]).

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 to fulfill the block cipher algorithms in a correct manner.

The Activity Diagram's modeling elements include: actions representing behavior execution, input/output pins working as parameters for 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 the existing modeling elements to derive specialized modeling constructs representing the operations required by block ciphers.

Figure 2 illustrates 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 with the intention of extending its behavior and defining a new modeling element. The stereotype's attributes shown in Figure 2 are called tagged values and define properties for the new modeling construct that are additional to the ones it inherits from its parent meta-class.

Our profile is encapsulated within a package that extends the package UML::-Activities::IntermediateActivities and uses the package UML::Actions::Basic-Actions in the Superstructure of UML [8]. IntermediateActivities was chosen

because it defines all the necessary meta-classes to base the new modeling elements on and is not polluted with other complex meta-classes. The profile derives several stereotypes from the **Action** meta-class to model the bitwise operations that are common to the block ciphers, as well as the S-Box components; it also derives a stereotype from the meta-class **ObjectFlow** to model edges transmitting bit-blocks; and it also derives a stereotype from the meta-class **ForkNode** to either distribute a bit-block along two or more different paths, or to partition a n-bit block into several bit-blocks of different lengths. An UML Activity Diagram built using this profile is called a Block Cipher Diagram.

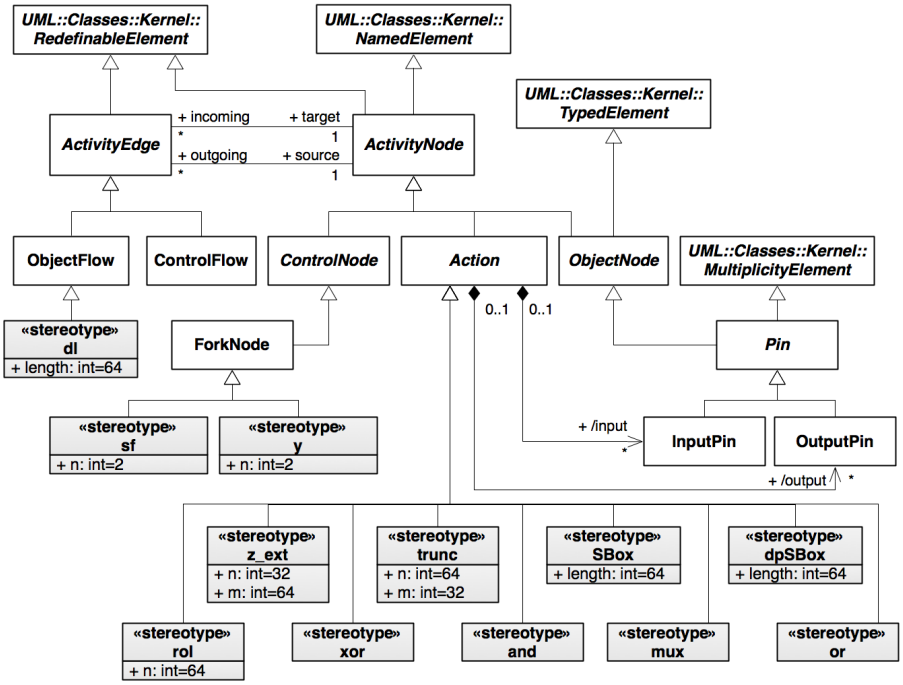


Fig. 2. Fragment of the UML 2.0 meta-model for Activity Diagrams extended with the stereotypes that make up the Block Cipher Profile.

Tables 1, 2, and 3 describe three stereotypes included in the Block Cipher Profile. Due to space limitations, it is not possible to describe all of the stereotypes that make up the profile in this document.

3 Applying the Block Cipher Profile

The hardware implementation of the KASUMI block cipher in its full structure is prohibitive for some embedded applications because it requires lots of hardware

Table 1. Definition of the **z_ext** stereotype in the Block Cipher Profile.

Name:	**z_ext**.
Generalizations:	**Action**.
Description:	An action that zero-extends the incoming bit-block.
Attributes:	n. An integer attribute indicating the length in bits of the incoming bit-block. Its default value is 32. m. An integer attribute indicating the length in bits of the outgoing bit-block. Its default value is 64.
Associations:	*input*: **InputPin**. A pin connected to the action that holds input bit-blocks to be consumed by the action. *output*: **OutputPin**. A pin connected to the action that holds output bit-blocks produced by the action.
Constraints:	$n \leq m$. There must be exactly two pins connected to this action; one of them must be an instance of the **InputPin** meta-class, whereas the other must be an instance of the **OutputPin** meta-class. The input pin must be attached to an edge that is an instance of the **dl** meta-class. The output pin must be attached to an edge that is an instance of the **dl** meta-class. The length of the bit-block in the incoming edge attached to the input pin must be equal to the n attribute. The length of the bit-block in the outgoing edge attached to the output pin must be equal to the m attribute.
Semantics:	Instances of **z_ext** are actions in a Block Cipher Diagram that receive an n-bit block as input and produces a m-bit block as output, with $n \leq m$. The output block's n least significant bits are set to the input block, and its $(m - n)$ most significant bits are all set to zero.

Table 2. Definition of the **dl** stereotype in the Block Cipher Profile.

Name:	**dl**.
Generalization:	**ObjectFlow**.
Description:	An edge that models the flow of bit-blocks between nodes.
Attributes:	*length*. An integer attribute indicating the length, in bits, of the block flowing along the edge.
Associations:	*source*: **ActivityNode**. The node the edge departs from. *target*: **ActivityNode**. The node the edge arrives to.
Constraints:	$1 \leq length \leq 128$. The edge must be attached to an instance of either the **Pin** meta-class or the **Action** meta-class or the **ForkNode** meta-class. See Figure 2. If the edge is attached to two pins then one of those pins must be an instance of the **InputPin** meta-class, the other must be an instance of the **OutputPin** meta-class.
Semantics:	Instances of **dl** (data line) are special edges intended to model transferences of bit-blocks between nodes in a Block Cipher Diagram. Data lines transfer bit-blocks whose length is greater than zero but less than or equal to 128 bits. When a **dl** instance's length attribute is set to 1 then the edge transfers a signal.

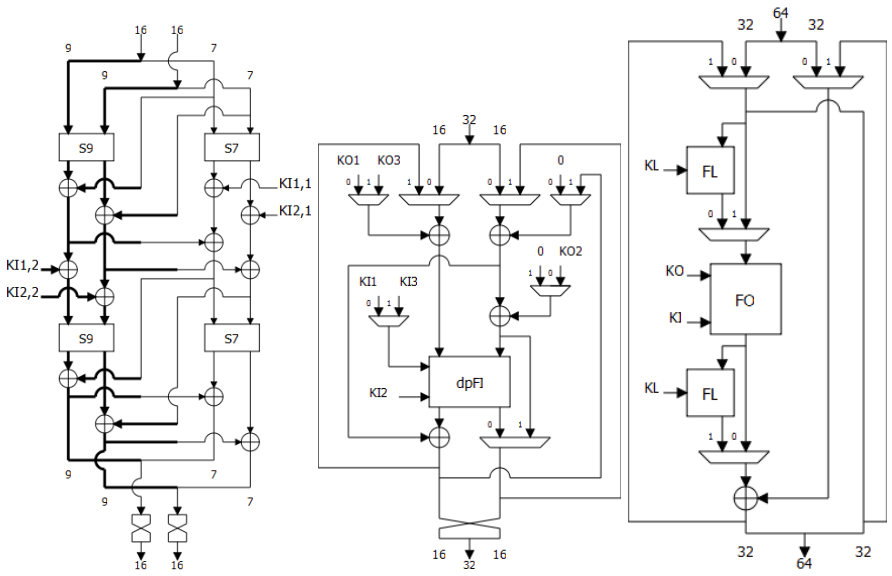
Table 3. Definition of the **sf** stereotype in the Block Cipher Profile.

Name:	**sf**.
Generalization:	**ForkNode**.
Description:	Splits an incoming bit-block into n bit-blocks of different lengths.
Attributes:	n. An integer attribute indicating the number of bit blocks outgoing the fork node.
Associations:	*incoming*: **ActivityEdge**. Edge that has the fork node as target. *outgoing*: **ActivityEdge**. Edges that have the fork node as source.
Constraints:	There must be exactly n outgoing edges, where n is the fork node's attribute. The incoming edge and all of the outgoing edges must be instances of the **dl** meta-class. The sum of the *length* attributes of each of the outgoing edges must be equal to the *length* attribute of the incoming edge.
Semantics:	Instances of **sf** (split fork) are special fork nodes that partition the bit-block in the incoming edge into n bit-blocks, and issue each of these bit-blocks through an independent outgoing edge. All of the outgoing edges are concurrent. The length of the incoming bit-block is indicated by the *length* attribute of the incoming edge. Similarly, the length of each of the outgoing bit-blocks is indicated by the *length* attribute of the corresponding outgoing edge. The sum of the *length* attributes for the outgoing edges must be equal to the length of the incoming edge.

components. In cases like this the designers usually manipulate the structure of the algorithm to obtain a representation that uses a minimal number of components. After a fixed number of successive iterations over this small set of components, by feeding back the result of the current iteration to the input of the design, the algorithm completes its task. Figure 3 illustrates the final result of a simplification process that is described in detail by Balderas, et al. in [4].

The simplified design combines two instances of the FI function into a single module that accepts two 16-bit inputs; see Figure 3(a). The four S-boxes internal to this dual-input FI function can be implemented either as combinational blocks that perform boolean functions over their inputs to generate their outputs, or as memories that store the correct value for each of the possible inputs. This dual-input FI function block is used by the simplified version of the FO function twice per round; see Figure 3(b). Therefore, the simplified KASUMI structure in Figure 3(c) requires two times eight equals sixteen iterations, as well as 16 clock cycles, to cipher a 64-bit block and has a throughput of 164.45 Mbps in a Virtex-E FPGA.

The profile is able to model the simplified structure of the KASUMI algorithm, as shown in the diagrams in Figure 4. The diagrams' modeling elements are labeled with a keyword containing the name of the stereotype they are instances of. For example, all of the edges in the diagrams are labeled with the keyword ≪dl≫to indicate that they are instances of the **dl** stereotype and, therefore, model the flow of bit-blocks. The profile is suitable to allow the designer to explore multiple design alternatives in a shorter period of time. The main idea is that the developer builds an initial model of the structure of the block cipher according to his/her architectural strategies, automatically synthesizes VHDL code from it, tests this code using a number of standard test benches, and computes the parameters of interest (performance, power consumption or area) to validate the design. If something goes wrong, or if the designer conceives a different architecture for the block cipher, it is always possible to directly



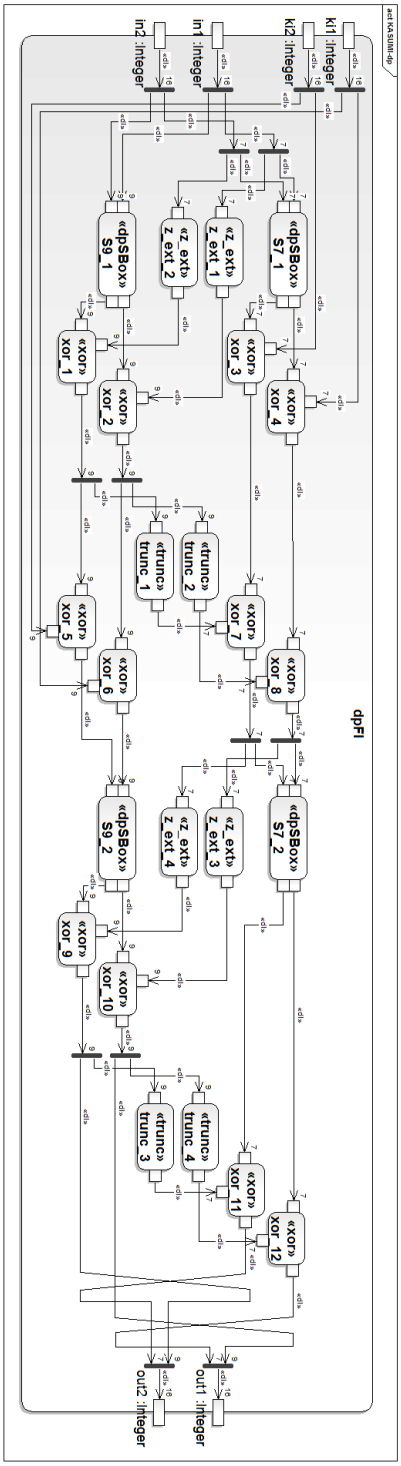
(a) Dual-input FI function (b) Simplified FO function (c) Simplified KASUMI structure

Fig. 3. The simplified structure of the KASUMI block cipher (from [4]).

manipulate the UML model to correct errors or to reorganize the architecture of the model, and then perform the test cycle again. The expectation is that handling domain-specific UML modeling elements and having a complete view of the design will be more productive than sketching the design and then writing the corresponding code in an implementation language like VHDL [9].

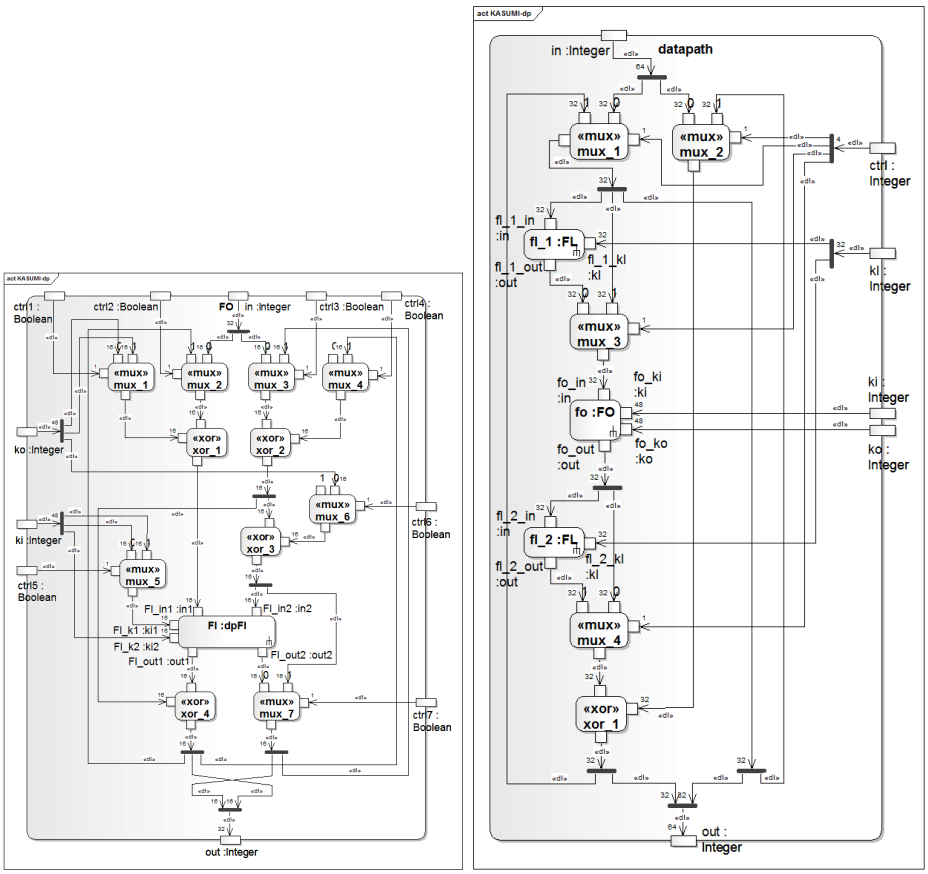
The models for the simplified FI and FO components, and for the simplified main Feistel structure, are self-contained and enclosed within an activity modeling element so that each can be subsequently reused by another model. This is the case of the activity containing the dual-input FI function, see Figure 4(a), which is used by the activity modeling the simplified FO function, as shown in Figure 4(b). The dual-input FI sub-activity within the simplified FO activity, denoted by the rake symbol (⫛), receives parameters and returns values through its input and output pins. The control signals expected by the activities in the models can be generated by state machine modeling constructs in UML 2.0.

It is important that the designer assigns correct values to the attributes of the modeling elements in the Block Cipher Diagrams. These attributes provide important information about the configuration of the modeling elements to a code synthesizer to produce correct VHDL code. Depending on the UML modeling tool, the attributes and the values assigned to them might be shown next to each modeling element, as tagged values, or not.



(a) Dual-input FI function

Fig. 4. Block Cipher Diagram for the simplified KASUMI structure.



(b) Simplified FO function (c) Simplified KASUMI structure

Fig. 4. Block Cipher Diagram for the simplified KASUMI structure (cont.)

4 Conclusions

This paper has discussed the convenience of being able to describe the functionality of digital hardware systems at higher levels of abstraction and let a number of transformation tools to synthesize an specific implementation from such descriptions, according to the model-driven engineering principle. This paradigm should have a positive impact on the alleviation of design complexity and the increase of the productivity of the developer.

The Block Cipher Profile described in this document is the first step towards the implementation of a design flow that will allow us to specify the structure and behavior of a digital communications system by means of UML 2.0 models, and derive a hardware implementation from the diagrams. One of the principles behind this design flow is the definition of domain-specific modeling languages that provide constructs and abstractions that are closer to the application domain than to the implementation technologies. Due to the extension capabilities of UML 2.0, as well as its graphical nature, we chose this modeling language as the base language for our domain-specific languages.

References

1. 3rd Generation Partnership Program: Universal Mobile Telecommunications System (UMTS), Specification of the 3GPP confidentiality and integrity algorithms, Document 2: Kasumi specification (3GPP TS 35.202 version 7.0.0 Release 7) (2007)
2. Bailey, B., Martin, G., Piziali, A.: ESL Design and Verification. A Prescription for Electronic System-Level Methodology. Morgan Kaufmann, San Francisco (2007)
3. Björklund, D., Lilius, J.: From UML Behavioral Descriptions to Efficient Synthesizable VHDL. In: 20th IEEE Norchip Conference, IEEE, Copenhagen (2002)
4. Balderas-Contreras, T., Cumplido, R.: An Efficient FPGA Architecture for Block Ciphering in Third Generation Cellular Network. In: Technical Conference of The International Embedded Solutions Event, Santa Clara, California (2004)
5. Balderas-Contreras, T., Cumplido, R.: High Performance Encryption Cores for 3G Networks. In: 42nd Annual ACM IEEE Design Automation Conference, pp. 240–243. ACM, New York (2005)
6. Flynn, M.J., Hung, P.: Microprocessor Design Issues: Thoughts on the Road Ahead. IEEE Micro 25(3), 16–31 (2005)
7. Kent, S.: Model Driven Engineering; In Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
8. Object Management Group: OMG Unified Modeling Language (OMG UML) Superstructure V2.1.2. OMG Document Number: formal/2007-11-02 (2007)
9. Picek, R., Strahonja, V.: Model Driven Development - Future or Failure of Software Development? In: Conference on Information and Intelligent Systems, Croatia (2007)
10. Riccobene, E., Scandura, P., Rosti, A., Bocchio, S.: A UML 2.0 Profile for SystemC. Technical report, ST Microelectronics (2005)
11. Semiconductor Industry Association: International Technology Roadmap for Semiconductors. Design Chapter. (2007)