Sergey Ostroumov | Leonidas Tsiopoulos | Juha Plosila | Kaisa Sere

Generation of Structural VHDL Code with Library Components from Formal Event-B Models

Turku Centre *for* Computer Science

TUCS Technical Report
No 1073, March 2013

Generation of Structural VHDL Code with Library Components from Formal Event-B Models

Sergey Ostroumov

TUCS – Turku Centre for Computer Science
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
Sergey.Ostroumov@abo.fi

Leonidas Tsiopoulos

Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
Leonidas.Tsiopoulos@abo.fi

Juha Plosila

University of Turku, Department of Information Technology
Joukahaisenkatu 3-5 B, 20014 Turku, Finland
Juha.Plosila@utu.fi

Kaisa Sere

Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
Kaisa.Sere@abo.fi

TUCS Technical Report

No 1073, March 2013

Abstract

We propose a design approach to integrating correct-by-construction formal modeling with hardware implementations in VHDL. Formal modeling is performed within the Event-B framework that supports the refinement approach, i.e., stepwise unfolding of system properties in a correct-by-construction manner. After an implementable deterministic model of a hardware system is derived, we apply an additional refinement step in order to introduce hardware library components in the form of functions. We show the mapping between these functions and corresponding library components such that a structural, i.e., component-based, VHDL description is derived. The application of functions binds unrestricted data types and substitutes regular operations with function calls. The approach is presented through examples that illustrate the additional refinement step and the code generation. We show the advantages in terms of occupied area and performance of the descriptions that incorporate hardware library components. In addition, we show generation of test cases from a formal model, which facilitates conformance or online testing.

Keywords: automated refinement, code generation, design flow, Event-B, formal methods, library components, structural VHDL, test cases generation

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Due to advances in Very-Large-Scale-Integration technology, designers can create increasingly complex systems on a single chip enabling energy-efficient execution of applications. These systems usually consist of a number of various components such that these components interconnected with each other constitute the functionality of the system. However, as complexity of a system and the number of components grow, it is rather infeasible to perform exhaustive testing in order to guarantee correct behavior of the system.

One of the appropriate approaches for developing correct systems is provided by formal methods. The application of formal methods can be categorized into two techniques. The model-checking [1] technique focuses on extracting a formal model from an implementation and checking some properties on this model. These techniques have been successfully employed (e.g., [2]) to identify errors that were undetected during normal design process. Modification and re-checking of the implementation should then be applied until the required integrity level is achieved.

Another technique to guarantee the correct behavior of a system is offered by a stepwise formal development. The formal modeling is performed following the refinement approach, i.e., unfolding system properties in a stepwise and correct-by-construction manner. Therefore, the derived formal model (specification) of the system is proved to be correct w.r.t. its functional requirements introduced as invariants. The utility of this approach can be further enhanced by automated code generation.

For the work in this paper, we utilize the latter approach and use the Event-B formalism [3] as the main framework for formal development. This formalism supports the refinement approach and has adequate tool support – the Rodin platform [4]. This platform is open source software offering the opportunity for an extension of its functionality in the form of plug-ins. Since code generation is a natural step for formal design flow, there are plug-ins that allow one to derive code in software languages such as C, Java, etc. [5]. However, due to the fact that hardware description languages (HDLs) differ in semantics and syntax from software languages, the same methods and techniques cannot be directly and completely applied to hardware design and code generation. Hence, we aim at facilitating the process of HDL description generation from formal models.

The target HDL is the VHSIC Hardware Description Language (VHDL). This language is standardized [6] and widely used in hardware design for systems based on field-programmable-gate-array or application-specific integrated-circuits technologies. VHDL supports the notion of library components allowing the designers to develop a system in a structural, i.e., component-based, manner and to derive possibly optimized code in terms of area and performance.

In this paper, we propose a design flow that integrates correct-by-construction formal modeling with hardware implementations in VHDL. We show the application of an additional refinement step to a deterministic implementable model. This refinement step serves as the middleware between a component-based formal model and its structural VHDL code. At this refinement step, we introduce VHDL (library) components as Event-B functions. We present a subset of library components and show the interconnection between them. The formal library can be further extended with the components used during the design. Additionally, we present the generation of test cases from the formal model using ProB animator and model-checker [7]. These test cases allow for automating the behavioral comparison between the formal model and the generated code or they can be deployed for online testing of the implementation [8].

To support our approach, we have developed a prototype of a plug-in [27] that automates the additional refinement step and allows one to generate a structural VHDL description in an automated manner. In addition, we have created another plug-in [27] to provide test cases generation.

2 Related Work

There exist several formalisms that provide specification and verification of hardware systems such as Signal [9], Esterel [10], ForSyDe [11] and others. Signal is dedicated to data-flow applications domain while Esterel is for control-flow ones. ForSyDe represents the design methodology targeting at covering both domains. The commonality of these languages is that they are all based on the perfect synchrony hypothesis. This hypothesis assumes a zero delay between consuming inputs and producing outputs. In addition, only Signal and ForSyDe support the notion of refinement. Refinement in Signal relies on checking if simulation of inputs and outputs preserves flow-equivalence (model checking) [12]. Refinement in ForSyDe stands for the mapping one process network onto another one restricting these networks to have the same inputs and outputs [11]. Moreover, these transformations have to be performed according to the predefined library.

BlueSpec [13] has been proposed as another solution to formal hardware verification and code generation. The language represents an extension of SystemVerilog and has a sound semantics allowing one to verify certain properties. It also supports design by refinement offering a possibility of integrating automated reasoning into the design flow [14]. However, automated verification of system correctness is provided by external theorem provers and/or model checkers such as PVS [14] and SPIN [25].

Evans [15] describes the mapping of VHDL to B and Communicating Sequential Processes (CSP) methods. The author proposes to derive a B model from VHDL and

formalize requirements with CSP. This approach uses a model-checking technique that requires modification and re-checking of the implementation until the desired integrity level is achieved.

In contrast to these approaches, we propose to use the Event-B formalism, which provides data and superposition refinement [16]. These types of refinement allow for stepwise unfolding of system functionality without restricting the model to have the same number of variables in refinements. Furthermore, one can postulate vital properties in terms of invariants for every refinement step. Following this approach, the discharging (proving) proof obligations serves as the guarantee that each refinement step preserves invariants and that concrete refinement step sustains their abstract counterparts. After the required model is derived and proved to be correct, a structural VHDL description is generated.

Another approach to deriving synchronous hardware systems proposed by Seceleanu [17] relies on Action Systems. The author describes the approach to modeling a synchronous system as read/write operations, where a combinational (asynchronous) circuit that consists of logic gates is followed by a synchronous component, namely a D-flip-flop, which operates on the clock signal. In addition, the author points out the mapping of such modeling to a behavioral VHDL description, where all operations are at one level of code, i.e., the description without components. Despite the fact that the Action Systems framework is similar to the Event-B formalism, it has a different underlying structure, which makes it infeasible to completely apply this approach to Event-B models. Furthermore, in contrast to this approach, we propose to derive component-based models and generate structural VHDL descriptions with library components.

Hallerstede and Zimmermann [18] proposed an approach to VHDL code generation from formal B models. The authors describe the mapping between B models and VHDL code through a middleware language B0, which allows one to generate code without components. This approach is adopted by AtelierB tool and supported by industrial partners [19]. Since Event-B is a descendant of B method that allows us to model reactive systems and has a different underlying structure, it is not straightforward how to apply this approach to Event-B models. Furthermore, we consider a component-based design flow, where components are injected into a formal model in the form of functions. This design flow allows for generating a structural VHDL description from such a model.

A similar approach to VHDL code generation has been proposed by Ostroumov and Tsiopoulos [20]. The authors suggest utilizing the conditional statement **if** condition **then** action **end if** in the process clause. This guarantees conformance of sequential VHDL behavior to the behavior of its formal counterpart enabling generation of a behavioral (i.e., without components) VHDL description from an implementable model following the usual proof-based design. We adopt and vastly extend the approach of

[20]. However, in contrast to this approach, we propose to apply an additional refinement step in order to derive a component-based model and, consequently, a structural VHDL description. The correctness of the additional refinement step is established through the proof obligations of the Event-B formal framework.

A BHDL tool has been proposed for digital circuit design [26]. The tool converts a VHDL description into B specification with two machines: an abstract that represents a VHDL entity and an implementation that corresponds to the architecture. Then, these two machines are verified using the B engine and the VHDL comments are interpreted as invariant properties. In contrast to this approach, we derive an implementable deterministic Event-B model following the usual refinement-based development. Then, components are injected into the model so that a structural VHDL description can be generated.

3 VHDL Description

3.1 VHSIC Hardware Description Language

VHDL, a standardized hardware description language [6], is widely used in hardware design and is supported by many Computer Aided Design tools (e.g., [22]). A VHDL description consists of two basic elements: an *entity* and an *architecture*. Every entity has a name and contains two clauses: *generic* that determines parameters for this entity and *port* that specifies inputs and outputs of this entity (an interface). The inputs and the outputs are distinguished by the keywords *in* and *out*, respectively.

The architecture attached to some entity has a name and a body that describes the behavior (the function) of a hardware component. Inside the architecture, a designer can introduce internal signals and other (e.g., library) components using the keyword *component* (Fig. 1).

A component is simply a predefined entity supplied with some architecture. The component entity has generic parameters that have to be instantiated using the keywords *generic map*. The connection between components is specified by the keywords *port map*. The keywords generic map and port map constitute the architecture body along with the *process* clause. The execution of the process is determined by a list of signals, namely the *sensitivity list*. In the process clause, we utilize the conditional statement **if** condition **then** action **end if**. If the condition holds, the action is executed.

The VHDL action in the process is an assignment to a signal of the form `s <= E`, where `s` is an internal or output signal and `E` is a constant or an expression over the input and/or internal signals. Every such an assignment is not instant. In other words, every signal has a buffer and the actual assignment takes place when the whole process

completes its execution. Hence, all the signals involved in a process are updated simultaneously.

```
entity Entity is
generic (--Parameters);
port (--Inputs : in std_logic/std_logic_vector,
       --Outputs : out std_logic/std_logic_vector);
end Entity;

architecture arch of Entity is
  -- Definitions of internal signals
  signal <signal_name> : <signal_type> := <initial_value>;
  -- Definitions of components
  component <component_name>
    generic (<component_parameters>);
    port (<component_interface>);
  end component;
begin
  -- Statements
  <component_label> : <component_name>
    generic map(<component_parameters_instantiation>)
    port map(<component_interface_mapping>);

  <process_label>:
  process (<sensitivity_list>) is begin
    if (<condition>) then <action>;
    end if;
  end process;
end arch;
```

Figure 1: VHDL entity and architecture

3.2 Hardware Library Components

Library components allow the designers to tackle complexity of a system facilitating faster design. Let us review a subset of library components available in Quartus-II software by Altera [22]. A small subset of them is presented in Table 1, where the components LPM_DIVIDE(DIVIDER) and LPM_DIVIDE(MODULO) differ in the output they produce and the abbreviations ALB, AEB etc. of the LPM_COMPARE component stand for A less than B, A equals to B etc., respectively. However, the library of formal components is not limited to the components presented in Table 1 and can be further extended since every library component has a unique definition.

The inputs and the outputs of the library components described here are bits or arrays of bits represented by STD_LOGIC and STD_LOGIC_VECTOR VHDL types, respectively. The type STD_LOGIC_VECTOR denotes a set of signals (a bus) whose number is determined by some constant (parameter defined in the generic clause). For the sake of brevity, we exemplify the mapping between a formal model and a structural code by the library component that performs the addition operation (Table 1, LPM_ADD_SUB(ADDER)). The others are interpreted in a similar manner.

The component has three parameters: LPM_WIDTH, LPM_DIRECTION and LPM_REPRESENTATION. LPM_WIDTH specifies the number of bits (the width) of the inputs and the output. LPM_DIRECTION determines the type of this component. If it

equals to ADD, the components is an adder. LPM_REPRESENTATION specifies the type of addition performed: signed or unsigned.

The adder operates on two inputs: the input port DATAA and the input port DATAB. It returns the result of addition of the two inputs to the output port RESULT as well as the carry flag to the output COUT. The input ports and the output port RESULT are of type STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0) while the carry flag is of type STD_LOGIC.

Table 1: A subset of library components

Components	Generic	Inputs	Outputs	Operation
LPM_ADD_SUB (ADDER)	LPM_WIDTH, LPM_DIRECTION = "ADD", LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	RESULT, COUT	RESULT=(DATAA+DATAB)(LPM_WIDTH-1..0), COUT=(DATAA+DATAB)(LPM_WIDTH)
LPM_ADD_SUB (SUBTRACTOR)	LPM_WIDTH, LPM_DIRECTION = "SUB", LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	RESULT	RESULT = (DATAA - DATAB)(LPM_WIDTH-1..0)
LPM_MULT	LPM_WIDTHA, LPM_WIDTHB, LPM_WIDTHP, LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	RESULT	RESULT = (DATAA * DATAB)
LPM_DIVIDE (DIVIDER)	LPM_WIDTHN, LPM_WIDTHD, LPM_NREPRESENTATION = "UNSIGNED", LPM_DREPRESENTATION = "UNSIGNED"	NUMER, DENOM	QUOTIENT	QUOTIENT = DATAA ÷ DATAB
LPM_DIVIDE (MODULO)	LPM_WIDTHN, LPM_WIDTHD, LPM_NREPRESENTATION = "UNSIGNED", LPM_DREPRESENTATION = "UNSIGNED"	NUMER, DENOM	REMAIN	REMAIN = DATAA % DATAB
LPM_COMPARE	LPM_WIDTH, LPM_REPRESENTATION = "UNSIGNED"	DATAA, DATAB	AGB, AGEB, AEB, ANEB, ALB, ALEB	AGB = bool(DATAA > DATAB), AGEB = bool(DATAA ≥ DATAB), AEB = bool(DATAA = DATAB), ANEB = bool(DATAA ≠ DATAB), ALB = bool(DATAA < DATAB), ALEB = bool(DATAA ≤ DATAB),

In the next section, we formalize library components as functions within Event-B to achieve correct-by-construction design flow. We show the one-to-one correspondence between formal and informal definitions of library components presented in Table 1.

4 Event-B Modeling

4.1 The Event-B Formalism

The Event-B formalism [3] allows designers to develop models in a correct-by-construction manner. A specification within Event-B consists of two main elements: a *context* and a *machine*. The context contains static data such as sets, constants, generic theorems and axioms. The machine models the dynamic part, which includes state

variables, theorems, system properties that must always hold (invariants) and events that modify the state variables. The context can be extended by another context and the machine can be refined by another machine. Moreover, the machine can refer to the data defined in a context, if this machine sees this context.

An event within the Event-B framework has the following structure:

$$e \triangleq \textbf{any } x \textbf{ where } g \textbf{ then } a \textbf{ end},$$

where x is a list of local variables, g stands for the guard and a represents an action of the event e, respectively. The guard is a conjunction of predicates that determine the execution of the action. If the guard holds, the action is fired.

The action represents a composition of parallel assignments (denoted by ||) that modify state variables. There are three types of assignments in Event-B: deterministic (denoted by :=), non-deterministic from a set (denoted by :∈) and non-deterministic specified by a predicate (denoted by :|).

Each event in Event-B is viewed as a before-after predicate ($BA_e = BA(v, v')$) [3] that links the values of the variables before (v) and after (v') the execution of the event e. This scheme allows us to prove the correctness (consistency) of the model w.r.t. postulated invariants by discharging proof obligations (POs). In particular, every predicate (i.e., an invariant, a theorem, a guard or an action) has to be well-defined [21], i.e., sound. Each event, in its turn, has to preserve postulated invariants [3, 21]:

$$Inv \wedge g_e \Rightarrow [BA_e]Inv, \qquad \text{(INV)}$$

where Inv is a model invariant whilst g_e and BA_e are the guard and the before-after predicate of event e, respectively. The expression $[BA_e]Inv$ stands for a substitution in the invariant Inv with the before-after predicate BA_e.

An Event-B model of a system is created in a stepwise manner following the refinement approach. At every refinement step, one adds details towards an implementable model. While refining the model, new variables, invariants, theorems and events can be added. However, the overall behavior of a more concrete model must conform to the overall behavior of its abstraction. This fact is guaranteed through discharging POs guard strengthening (GRD) and action simulation (SIM) [3, 21]:

$$Inv \wedge Inv_r \wedge g_r \Rightarrow g, \qquad \text{(GRD)}$$
$$Inv \wedge Inv_r \wedge BA_{er} \Rightarrow BA_e, \qquad \text{(SIM)}$$

where structures with the sub-script r represent refined versions.

To ease proving effort when discharging the above POs, one can postulate and prove theorems. Depending on the Event-B element (a context and/or a machine) where a theorem is stated, corresponding POs (THM_c for a context and THM_m for a machine, respectively) have to be discharged:

$$A \Rightarrow ThC, \qquad (THM_c)$$

$$A \wedge I \Rightarrow ThM, \qquad (THM_m)$$

where A is a set of axioms defined in a context, I is a set of model invariants, ThC is a theorem postulated in a context whilst ThM is a theorem introduced to a machine.

The Event-B tool support – the Rodin platform [4] – automatically generates and attempts to discharge the POs described above. The tool usually achieves high-level of automation (usually over 80%), sometimes requiring user assistance through an interactive prover.

4.2 Formalization of Library Components

To be able to prove that Event-B formalization conforms to the definitions of hardware library components shown above, we define a function that converts a non-negative decimal number into its binary image. This function binds infinite data types (e.g., naturals) to be suitable for hardware implementation since hardware bit images cannot be infinite.

Definition 1: A bijective function $conv(C, d) = k_b$ converts a non-negative decimal number into its binary image. The parameter $C \in \mathbb{N}1$ determines the upper bound (i.e., the width) on which the function operates. The parameter $d \in 0..2^C-1$ represents a non-negative decimal number within the range $0..2^C-1$, where 2^C stands for 2 to the power of C. The function returns a binary image of the number d, namely $k_b \in \{x \mid x \in \{0,1\}^* \wedge W(x) = C\}$, where W(x) stands for the number of bits (the width) of the binary number k_b. The function is defined recursively as follows:

$$conv(C,d) = \begin{cases} 0...0_b, \text{ if } d = 0 \\ conv(C,d-1) +_b 0...1_b, \text{ if } d > 0, \end{cases}$$

where $x...y_b$ is a binary number (e.g., 010_b) whose length (i.e., the number of bits) is determined by the constant C and $n +_b m$ is a binary sum defined as $0_b +_b 0_b = 0_b$, $0_b +_b 1_b = 1_b$, $1_b +_b 0_b = 1_b$, $1_b +_b 1_b = 10_b$.

Example 1. Suppose C equals to 3. Then, any non-negative decimal number from the set $0..2^3-1$ (i.e., 0..7) can be represented as a binary number from 000 to 111:

$conv(3,0) = 000_b$;

$conv(3,5) = conv(3,4) +_b 001_b = conv(3,3) +_b 001_b +_b 001_b = conv(3,2) +_b 001_b +_b 001_b +_b 001_b = conv(3,1) +_b 001_b +_b 001_b +_b 001_b +_b 001_b = conv(3,0) +_b 001_b +_b 001_b +_b 001_b +_b 001_b +_b 001_b = 101_b$.

End of example.

The formalization of library components is performed by using functions applied to an Event-B context. A function f in a context is a constant that has at least two axioms.

The first axiom defines the type of the function, i.e., the type of its arguments (T) and returning result (T′):

$$T_1 \times \ldots \times T_n \rightarrow T'_1 \times \ldots \times T'_m,$$

where $T_1 \times \ldots \times T_n$ is the Cartesian product, i.e., the set of all the pairs formed from the types T_1 to T_n.

The second axiom specifies the result returned by the function f:

$$\forall x_i \,.\, x_i \in T_i \Rightarrow f(x_1 \mapsto \ldots \mapsto x_n) = Exp(x_1,\ldots,x_n),$$

where $i \in 1..n$ and n is the number of arguments that the function f takes (determined by its type). The symbol ↦ represents an ordered pair and allows one to specify a number of arguments for a function. The function f produces the result defined by the expression Exp over x_i.

Table 2: Components as Event-B functions

Function	Constant(s)	Axioms
add_ unsigned	add_unsigned_ width	add_unsigned ∈ 0..2^add_unsigned_width-1 × 0..2^add_unsigned_width-1 → 0..2^(add_unsigned_width+1)-1 ∀dataa, datab . dataa ∈ 0..2^add_unsigned_width−1 ∧ datab ∈ 0..2^add_unsigned_width−1 ⇒ add_unsigned(dataa↦datab) = dataa + datab
sub_ unsigned	sub_unsigned_ width	sub_unsigned ∈ 0..2^sub_unsigned_width−1 × 0..2^sub_unsigned_width−1 → 0..2^sub_unsigned_width−1 ∀dataa, datab . dataa ∈ 0..2^sub_unsigned_width−1 ∧ datab ∈ 0..2^sub_unsigned_width−1 ⇒ (dataa ≥ datab ⇒ sub_unsigned(dataa↦datab) = dataa − datab) ∧ (dataa < datab ⇒ sub_unsigned(dataa↦datab) = 0)
mult_ unsigned	mult_unsigned_ width_a mult_unsigned_ width_b	mult_unsigned ∈ 0..2^mult_unsigned_width_a−1 × 0..2^mult_unsigned_width_b−1 → 0..2^(mult_unsigned_width_a+mult_unsigned_width_b)−1 ∀dataa, datab . dataa ∈ 0..2^mult_unsigned_width_a−1 ∧ datab ∈ 0..2^mult_unsigned_width_b−1 ⇒ mult_unsigned(dataa↦datab) = dataa∗datab
div_ unsigned	div_unsigned_ width_n div_unsigned_ width_d	div_unsigned ∈ 0..2^div_unsigned_width_n−1 × 1..2^div_unsigned_width_d−1 → 0..2^div_unsigned_width_n−1 ∀dataa, datab . dataa ∈ 0..2^div_unsigned_width_n−1 ∧ datab ∈ 1..2^div_unsigned_width_d−1 ⇒ div_unsigned(dataa↦datab) = (dataa ÷ datab)
mod_ unsigned	mod_unsigned_ width_n mod_unsigned_ width_d	mod_unsigned ∈ 0..2^mod_unsigned_width_n−1 × 1..2^mod_unsigned_width_d−1 → 0..2^mod_unsigned_width_d−1 ∀dataa, datab . dataa ∈ 0..2^mod_unsigned_width_n−1 ∧ datab ∈ 1..2^mod_unsigned_width_d−1 ⇒ mod_unsigned(dataa↦datab) = (dataa mod datab)
comp_ unsigned	comp_ unsigned_width	comp_unsigned ∈ 0..2^comp_unsigned_width−1 × 0..2^comp_unsigned_width−1 → BOOL × BOOL × BOOL × BOOL × BOOL × BOOL ∀dataa,datab . dataa∈0..(2^comp_unsigned_width−1) ∧ datab∈0..(2^comp_unsigned_width−1) ⇒ comp_unsigned(dataa↦datab)= bool(dataa > datab)↦bool(dataa ≥ datab)↦bool(dataa = datab)↦ bool(dataa ≠ datab)↦bool(dataa < datab)↦bool(dataa ≤ datab)

Following the approach of introducing functions into an Event-B context, we define a formal library of presented hardware components as shown in Table 2. For instance, let us consider the function add_unsigned in Table 2 that formalizes the VHDL adder component (Table 1, LPM_ADD_SUB (ADDER)) within Event-B. The type of this function is determined by the first axiom, where add_unsigned_width ∈ ℕ1 is the width. The returning result is specified by the second axiom that models the addition operation of two non-negative numbers.

Theorem (ADD): add_unsigned conforms to LPM_ADD_SUB, where add_unsigned_width = LPM_WIDTH and the parameters LPM_DIRECTION and LPM_REPRESENTATION of LPM_ADD_SUB equal to ADD and UNSIGNED, respectively (ensured by the code generation algorithm described in the next section).

Proof:

1. The function add_unsigned operates on the same input values in decimal as the library component LPM_ADD_SUB in binary:

   $$\forall inp \cdot inp \in 0..(2^{add\_unsigned\_width})-1 \Rightarrow (\exists inp_b \cdot inp_b = conv(add\_unsigned\_width, inp),$$

   where inp represents a decimal input to the function while inp_b is a binary image of inp supplied as an input to the component.

2. The result of the function add_unsigned ranges from 0 to 2^(add_unsigned_width+1)-1, i.e., one bit more than the width of the inputs. Hence, the function returns the result as well as the carry flag which corresponds to the value on the outputs RESULT and COUT of the component:

   $$\forall res \cdot res \in 0..2^{(add\_unsigned\_width+1)}-1 \Rightarrow (\exists COUT, RESULT \cdot COUT + RESULT = conv(add\_unsigned\_width+1, res)),$$

   where res represents the result of the function whereas COUT + RESULT is concatenation of the outputs COUT and RESULT of the component. Clearly, the overflow will never occur.

   Example 2. Suppose add_unsigned_width = LPM_WIDTH = 3, the input ranges of the function and the component are 0..7 and 000..111, respectively, while the result ranges are 0..15 and 0000..1111, respectively. The leftmost (the most significant) bit of LPM_ADD_SUB represents the carry flag.
   End of example.

3. Finally, the definition of the function add_unsigned models the addition operation of two inputs, namely dataa and datab, i.e., the function of the adder component.

Similarly, we can reason about other functions that specify other library components (Table 2) □.

While modeling a system in Event-B, one has to discharge POs (INV), (GRD) and (SIM) to show correctness of the system specification (Section 4.1). To ease discharging of these POs, we postulated and proved (discharged PO (THM_c)) the following theorems along with the definitions of functions in a context:

$$\forall n \,.\, n \in \mathbb{N} \Rightarrow 0 < 2^n, \qquad (ThC_1)$$

$$\forall x,y \,.\, x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x < y \Rightarrow 2^x < 2^y, \qquad (ThC_2)$$

$$\forall n \,.\, n \in \mathbb{N} \Rightarrow 2*2^n = 2^{(n+1)}. \qquad (ThC_3)$$

Theorem (ThC_1) states that 2 to the power of some natural number is a positive number. In other words, the set of values starting from 0 and ending in 2 to the power of some constant is not empty. Hence, the functions formalizing VHDL library components are well-defined on these values. Theorem (ThC_2) shows the order relation between numbers whose powers are in the order relation as well. Theorem (ThC_3) postulates inductiveness of 2 to the power of some constant.

5 The Design Flow and Code Generation Algorithm

The use of Event-B as a starting point in the design flow of hardware systems facilitates correct-by-construction development w.r.t. postulated properties and requirements. Code generation in an automated fashion enhances the utility of the approach reducing testing effort at later design phases. Hence, we propose the design flow shown in Fig. 2.

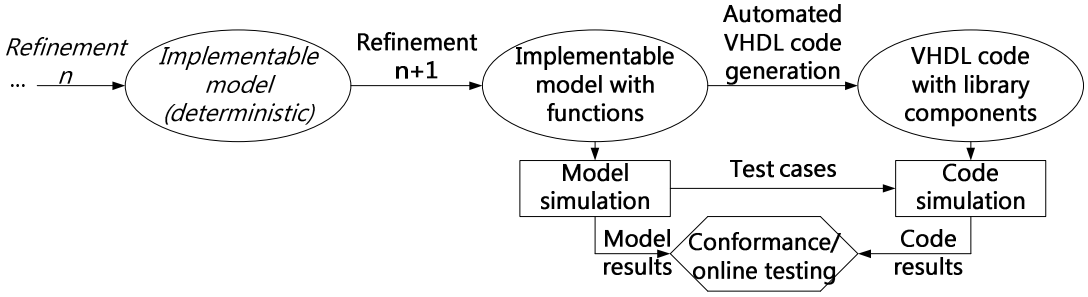

Figure 2: The design flow

An implementable deterministic model is derived following usual refinement-based development. Then, we apply an additional refinement step that serves as the middleware between a component-based formal model and structural VHDL description. The correctness of this refinement step is established by proving POs (INV), (GRD) and (SIM) using theorems of types (THM_c) and (THM_m) (Section 4). The Rodin platform [4] generates these POs and attempts to prove them automatically. The

algorithmic representation of the code generation utilizing the additional refinement step is as follows:

1. Refine an implementable model by extending the most definite context (if any) and refining the most concrete machine of the model.
2. Instantiate necessary functions to the newly created context by specifying the set of values they operate with (their width). This set is bounded by the corresponding constants. The necessary functions to be used are determined by operations used in the machine actions.
3. Restrict the types of the state variables according to the specified constants and functions where these variables are involved.
4. Replace regular operations in actions with calls to the corresponding functions.
5. To generate code, interpret each function in the context as a corresponding library component in VHDL according to the defined mapping.
6. Interpret the type of a variable which has been restricted by some constant as STD_LOGIC_VECTOR in terms of VHDL types. The length (the width) number is determined by the corresponding constant.
7. For every component instantiation, introduce an internal VHDL signal connected to the component output(s) in order to allow for chaining of diverse components.

To support the proposed design flow, we have developed a prototype of a plug-in that automates the additional refinement step and allows one to generate a structural VHDL description in an automated manner. The plug-in implements the algorithm described above and operates as follows. Firstly, it extends the most definite context of an Event-B project, if any, by copying theorems (ThC_1)-(ThC_3) to it. Secondly, the plug-in traverses the most concrete machine of the project. Each time it sees a regular operation that can be substituted with function call, the plug-in instantiates a corresponding function available in the library. A designer specifies the width of a function being instantiated. Thirdly, it refines the most concrete machine and replaces each regular operation with a function call. For instance, z := add_unsigned(x↦y) replaces z := x+y. Fourthly, for every variable involved in such an action, the plug-in generates a type invariant (PO (INV) in Section 4) in order to bind the values according to the instantiated function. Finally, it applies theorems (ThM, see PO (THM_m) in Section 4) of the form f(x↦y)=x op y to the machine, where x and y are the operands and f and op are the corresponding function and operation, respectively. For instance, if the function call add_unsigned(x↦y) replaces the expression x+y, then the theorem for this substitution is add_unsigned(x↦y) = x+y. These theorems allow for proving correctness of the additional refinement step. Hence, the behavior of the model is proved consistent.

A specification may contain several identical operations, e.g., two or more addition operations etc. In this case, for every function the name is formed from the function name, e.g., add_unsigned, and the suffix _n, where n is a number that starts from 0 and

is increased whenever another function definition is instantiated, e.g., add_0_unsigned, add_1_unsigned. Therefore, each function determines one library component such that the one-to-one mapping between formal model and VHDL code is feasible.

6 Experimental Results

Let us examine a couple of examples showing the application of our method to modeling within the Event-B framework and generating structural VHDL code. The examples show a sequential composition of components using different modeling styles in Event-B. Furthermore, we show test cases generation allowing one to perform conformance testing between the model and the generated code or to deploy online testing.

6.1 Component Chaining in Separate Events

This example illustrates the use of library components such that the result computed in one event is used as an input for the computations in another event (Fig. 3).

```
invariants
  Voltage_I ∈ ℕ ∧ Current_I ∈ ℕ1 ∧
  Resistance ∈ ℕ ∧ Inputs_Read ∈ BOOL ∧
  (Temp_Read = TRUE ⇒
              Resistance = Voltage_I ÷ Current_I) ∧
  // Gluing invariant with a mode abstract
  // model
  (Inputs_Read = TRUE ∧ Temp_Read = TRUE ⇒
                                  Temp_I = Resistance)
events
...
Resist_Comp refines Temp_Read ≙
where  Temp_Read = FALSE ∧
  Inputs_Read = TRUE ∧ Current_I ≠ 0
with  Temp = Voltage_I ÷ Current_I
  Then Resistance ≔ Voltage_I ÷ Current_I ||
  Temp_Read ≔ TRUE
end

Compare refines Compare ≙
where  Temp_Read = TRUE ∧
  Inputs_Read = TRUE
then Temp_Read ≔ FALSE ||
  Inputs_Read ≔ FALSE ||
  Result_O ≔ bool(Resistance ≥ Temp_Threshold)
end
```

```
invariants
  Voltage_I ∈ 0..2^div_0_unsigned_width_n−1 ∧
  Current_I ∈ 1..2^div_0_unsigned_width_d−1 ∧
  Resistance ∈ 0..2^div_0_unsigned_width_n−1 ∧
theorem
  div_0_unsigned(Voltage_I↦Current_I)=Voltage_I÷Current_I
...
events
...
Resist_Comp refines Temp_Read ≙
where  Temp_Read = FALSE ∧
  Inputs_Read = TRUE ∧ Current_I ≠ 0
then
  Resistance ≔ div_0_unsigned(Voltage_I↦Current_I) ||
  Temp_Read ≔ TRUE
end

Compare refines Compare ≙
where  Temp_Read = TRUE ∧ Inputs_Read = TRUE
then  Temp_Read ≔ FALSE ||
  Inputs_Read ≔ FALSE ||
  Result_O :| ∃agb,aeb,aneb,alb,aleb .
    comp_0_unsigned(Resistance↦Temp_Threshold)=
    agb↦Result_O'↦aeb↦aneb↦alb↦aleb
end
```

Figure 3: Component chaining in separate events

Here, we model the calculation of temperature using Ohm's law (event Resist_Comp), where temperature is proportional to resistance (variable Resistance). Then, the obtained value is compared to some threshold and the comparison result is promoted further (event Compare). An instance of this example is aerospace designs domain (e.g., [18, 21]) where the temperature sensor represents a high-quality resistor with a platinum or golden thread.

For this model, the Rodin platform generated 57 POs of which 51 were proven automatically. Three POs of type (THM_c) with the proofs were automatically derived for the context theorems (ThC_1)-(ThC_3) (Section 4) by the plug-in. One PO of type (INV) has been proved interactively using the theorem (ThC_3). One well-definedness PO has been discharged for a theorem of type (THM_m) generated by the plug-in and introduced into the machine. The remaining POs of type (SIM) has been proved using theorems (THM_m) generated by the plug-in.

We generated VHDL descriptions with and without library components from this model (see Appendix A for the description with library components). We then synthesized each description using Quartus-II [20]. The tool analyzed them and provided the information about occupied area and performance. The number of logic elements (LE) measures the area. The worst-case setup time (Tsu) and the worst-case hold time (Th) illustrate the performance of this example. The synthesis results are summarized in Table 3. They show the advantages and possible optimizations in terms of area (2,7%) and performance (13,7%) of the implementation with library components.

Table 3: Synthesis results for state holding implementations

LE, qt.		**LE, %**	**Tsu, ns**		**Tsu, %**	**Th, ns**		**Th, %**
w/ lib	***w/o lib***		***w/ lib***	***w/o lib***		***w/ lib***	***w/o lib***	
36	37	2,7	9.975	11.562	13,7	2.262	2.215	-2%

6.2 Replacing Infix Operators with Prefix Function Calls

This example illustrates the model, where a single event produces the result using different operators (Fig. 4). The computation of the result proceeds as follows (event Result). The variables Input1_I and Input2_I are multiplied, their result is summed up with the variable Input3_I and this sum is then divided by Input1_I. The order in which the operations take place specify the chain of the corresponding hardware library components.

For this model, the Rodin platform generated 53 POs of which 49 were proven automatically. Three POs of type (THM_c) with the proofs were automatically copied for the context theorems (ThC_1)-(ThC_3) (Section 4) by the plug-in. They have been used to discharge the only proof obligation of type (INV) interactively.

invariants Input1_I ∈ ℕ1 ∧ Input2_I ∈ ℕ ∧ Input3_I ∈ ℕ ∧ Result_O ∈ ℕ ∧ Read_Write ∈ BOOL ∧ Read_Write = FALSE ⇒ Result_O = (Input1_I ∗ Input2_I + Input3_I) ÷ Input1_I **events** ... Result ≙ **where** Read_Write = TRUE **then** Read_Write ≔ FALSE ‖ Result_O≔(Input1_I∗Input2_I+Input3_I)÷Input1_I **end**	**invariants** Input1_I ∈ 0..2^mult_0_unsigned_width_a−1 ∧ Input2_I ∈ 0..2^mult_0_unsigned_width_b−1 ∧ Input3_I ∈ 0..2^add_0_unsigned_width−1 ∧ Result_O ∈ 0..2^div_0_unsigned_width_n−1 ∧ **theorem** mult_0_unsigned(Input1_I↦Input2_I)=Input1_I∗Input2_I ... **events** ... Result **refines** Result ≙ **where** Read_Write = TRUE **then** Read_Write ≔ FALSE ‖ Result_O≔div_0_unsigned(add_0_unsigned(mult_0_unsigned(Input1_I↦Input2_I)↦Input3_I)↦ Input1_I) **end**

Figure 4: Replacing infix operators with prefix function calls

Analogously to the previous example, we generated VHDL descriptions with and without library components from this model (see Appendix B for the description with library components). Then, we used Quartus-II [20] to synthesize each description and acquire information about area and performance. The worst-case time required to propagate the value on the input pin to the output pin (W-C Tpd) reflects the performance metric for this example. Table 4 summarizes the synthesis results, which show the advantages in terms of area (12,5%) and performance (15,4%) of the description with library components.

Table 4: Synthesis results for nested function calls

LE, qt.		**LE, %**	**W-C Tpd, ns**		**W-C tpd, %**
w/ lib	***w/o lib***		***w/ lib***	***w/o lib***	
28	32	12,5	14,71	17,38	15,4

6.3 Test Cases Generation

To automate conformance testing using an Event-B formal model as a so-called golden reference, we implemented another prototype of the plug-in that simulates the model and generates test cases. The model simulation is performed using ProB animator and model checker [7]. A user can specify the number of simulations to be executed, i.e., the number of test cases to be generated affecting the test coverage. The test cases are of the TCL format supported by ModelSim simulation environment [24] that allows designers to simulate an implementation and obtain values on the signals. Let us show several examples of test cases generated from the models presented above.

The test cases for the example shown in Fig. 3 (Section 6.1) are the following:

force Current_I 'd6; force Voltage_I 'd3,

force Current_I 'd11; force Voltage_I 'd7,
force Current_I 'd1; force Voltage_I 'd8,

where force is a command that forces (drives) a value on the signal and 'dx is a conversion of a decimal number x into its binary image.

The simulation results of the model and the code for these test cases are identical as shown in Table 5.

Table 5: Simulation results of the model and the code

Event-B model	VHDL code
Result_O: 0, Current_I: 6, Temp_Read: 0, Resistance: 0, Voltage_I: 3, Inputs_Read: 0, ; Result_O: 0, Current_I: 11, Temp_Read: 0, Resistance: 0, Voltage_I: 7, Inputs_Read: 0, ; Result_O: 1, Current_I: 1, Temp_Read: 0, Resistance: 8, Voltage_I: 8, Inputs_Read: 0, ;	Result_O: 0, Current_I: 6, Temp_Read: 0, Resistance: 0, Voltage_I: 3, Inputs_Read: 0, ; Result_O: 0, Current_I: 11, Temp_Read: 0, Resistance: 0, Voltage_I: 7, Inputs_Read: 0, ; Result_O: 1, Current_I: 1, Temp_Read: 0, Resistance: 8, Voltage_I: 8, Inputs_Read: 0, ;

The test cases for the example shown in Fig. 4 (Section 6.2) are as follows:

force Input3_I 'd13; force Input2_I 'd0; force Input1_I 'd2,
force Input3_I 'd1; force Input2_I 'd1; force Input1_I 'd2.

The simulation results of the model and the code for these test cases are identical as well (Table 6).

Table 6: Simulation results of the model and the code

Event-B model	VHDL code
Result_O: 6, Input3_I: 13, Read_Write: 0, Input2_I: 0, Input1_I: 2 ; Result_O: 1, Input3_I: 1, Read_Write: 0, Input2_I: 1, Input1_I: 2	Result_O: 6, Input3_I: 13, Read_Write: 0, Input2_I: 0, Input1_I: 2 ; Result_O: 6, Input3_I: 13, Read_Write: 0, Input2_I: 0, Input1_I: 2

7 Conclusion

We have presented a design flow integrating the formal development of a hardware system within the Event-B framework with structural, i.e., component-based VHDL implementation. To support the proposed approach, we have developed a prototype of a plug-in that automates the additional refinement step and generation of structural VHDL description. We believe that the application of formal methods at early stages of the design flow with automated code generation can reduce testing effort at later design phases. In addition, we have shown experimental results that illustrate optimization provided by the code with library components (2,5% and 12,5% in area and 13,7% as well as 15,4% in performance).

The formal library of hardware components is not limited to the components presented in this paper and can clearly be extended. Hence, we will consider the formalization of other hardware components that are often used in hardware design to enhance correct-by-construction development of diverse hardware systems.

A subset of components presented in this paper is considered to be combinational, i.e., these components are clockless. However, there are combinatorial components that depend on the clock signal. Hence, another direction of our future work is to extend the approach to support modeling a system that contains clocked components. This will allow a designer to derive a time-aware model and generate synchronous code from this model.

8 Acknowledgement

This work is supported by Academy of Finland and the Research Institute of Åbo Akademi University. In addition, the authors would like to thank Adjunct Prof. Marina Walden for the fruitful discussions and valuable feedback.

References

[1] E. Clarke, Model Checking, Cambridge: The MIT Press, 2002.

[2] A. Roychoudhury, T. Mitra, S.R. Karri, Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol, Design, Automation and Test in Europe conference (DATE), IEEE, pp. 828-833, 2003.

[3] J.-R. Abrial, Modeling in Event-B. System and Software Engineering, Cambridge: Cambridge University Press, 2010.

[4] The Rodin platform. Available: http://sourceforge.net/projects/rodin-b-sharp/

[5] S. Wright, Automatic Generation of C from Event-B, Workshop on Integration of Model-based Formal Methods and Tools, 2009, p. 14.

[6] IEEE Standard VHDL Language Reference Manual, IEEE 1076, 2008.

[7] M. Leuschel, M. Butler, ProB: A Model Checker for B, Proc. FME, Springer, vol. 2805, 2003, p. 855-874.

[8] K. Kohno, N. Matsumoto, A new verification methodology for complex pipeline behavior, Design Automation Conference, IEEE, pp. 816-821, 2001.

[9] A. Benveniste, P. Le Guernic, Hybrid Dynamical Systems Theory and the Signal Language, IEEE Transactions on Automatic Control 35(5), 1990, p. 535-546.

[10] D. Potop-Butucaru, R. de Simone, Optimizations For Faster Execution Of Esterel programs, Proc. of MEMOCODE conference, 2003, pp. 227-236.

[11] I. Sander, A. Jantsch, System Modelling and Transformational Design Refinement in ForSyDe, Transactions on Computer Aided Design of Integrated Circuits and Systems, IEEE, Vol. 23, 2004, pp. 17-32.

[12] J. Talpin, P. Guernic, S. Shukla, R. Gupta, F. Doucet, Polychrony for Formal Refinement Checking in a System-Level Design Methodology, Application of Concurrency to System Design (ACSD), IEEE, pp. 9-19, 2003.

[13] BlueSpec Documentation. Available: http://www.ece.ucsb.edu/its/bluespec/index.html.

[14] D. Richards, D. Lester, A monadic approach to automated reasoning for BlueSpec SystemVerilog, Innovations System Software Engineering, Springer-Verlag, pp. 85-95, 2011.

[15] N. Evans, Integrating Formal Methods with Informal Digital Hardware Development, Proc. of AVoCS, 2010, p. 16.

[16] R. J. R. Back, K. Sere, “Superposition Refinement of Reactive Systems”, Formal Aspects of Computing, Springer, Vol. 8, 1995, pp. 324-346.

[17] T. Seceleanu, Systematic Design of Synchronous Digital Circuits, Turku: TUCS Dissertations, Turku Centre for Computer Science, 2001.

[18] S. Hallerstede, Y. Zimmermann, “Circuit Design by Refinement in Event-B”, FDL, pp. 624-637, 2004.

[19] M. Benveniste, A «Correct by Construction» Realistic Digital Circuit, RIAB Workshop, FMWeek, 2009. Available: http://www.bmethod.com/pdf/riab/st-marc-benvenisteproved-realistic-circuit-handout.pdf

[20] S. Ostroumov, L. Tsiopoulos, VHDL Code Generation from Formal Event-B Models. IEEE Digital System Design, 14th Euromicro Conference, Oulu, 2011, pp. 127-134.

[21] K. Robinson. (2011, June, 28). System Modelling & Designing using Event-B. Available: http://www.cse.unsw.edu.au/~cs9116/PDF/SMD.pdf

[22] Quartus-II software. Avaliable: http://www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html

[23] B. Nuckolls, Practical Low Resistance Measurements, 2004. Available: www.aeroelectric.com/articles/LowOhmsAdapter_3.pdf

[24] ModelSim for Altera. Available: http://www.altera.com/products/software/quartus-ii/modelsim/qts-modelsim-index.html

[25] G. Singh, E. Shukla, Verifying Compiler bsed Refinement of Bluespec Specifications using the SPIN model Checker, 15th International SPIN Workshop, Springer, pp. 250-269, 2008.

[26] A. Aljer, P. Devienne, S. Tison, BHDL: Circuit design in B, Conference on Application of Concurrency to System Design, IEEE, pp. 1-2, 2003.

[27] Event-B to VHDL. Available: http://eventb-to-vhdl.tk

Appendix A

Component Chaining in Separate Events

```
LIBRARY IEEE;
LIBRARY LPM;
USE IEEE.STD_LOGIC_1164.ALL;
USE LPM.LPM_COMPONENTS.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Separate_Events IS
	GENERIC
	(
		Temp_Threshold : NATURAL := 80;
		div_0_unsigned_width_n : NATURAL := 8;
		div_0_unsigned_width_d : NATURAL := 8;
		compare_0_unsigned_width : NATURAL := 8
	);
	PORT
	(
		--Input ports
		Voltage_I : IN STD_LOGIC_VECTOR(div_0_unsigned_width_n-1 DOWNTO 0) :=
			STD_LOGIC_VECTOR(TO_UNSIGNED(0, div_0_unsigned_width_n));
		Current_I : IN STD_LOGIC_VECTOR(div_0_unsigned_width_d-1 DOWNTO 0) :=
			STD_LOGIC_VECTOR(TO_UNSIGNED(10, div_0_unsigned_width_d));

		--Output ports
		Result_O : OUT STD_LOGIC :='0'
	);
END Separate_Events;

ARCHITECTURE a OF Separate_Events IS

-- Components declaration
component LPM_DIVIDE
	GENERIC(
		LPM_WIDTHN : NATURAL;
		LPM_WIDTHD : NATURAL;
		LPM_NREPRESENTATION : STRING;
		LPM_DREPRESENTATION : STRING
	);
	PORT(
		NUMER : IN STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
		DENOM : IN STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0);
```

```
		QUOTIENT : OUT STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
		REMAIN : OUT STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0)
	);
END component;

component LPM_COMPARE
	GENERIC(
		LPM_WIDTH : NATURAL;
		LPM_REPRESENTATION : STRING
	);
	PORT(
		DATAA : IN STD_LOGIC_VECTOR(compare_0_unsigned_width-1 DOWNTO 0);
		DATAB : IN STD_LOGIC_VECTOR(compare_0_unsigned_width-1 DOWNTO 0);
		AGB : OUT STD_LOGIC;
		AGEB : OUT STD_LOGIC;
		AEB : OUT STD_LOGIC;
		ANEB : OUT STD_LOGIC;
		ALB : OUT STD_LOGIC;
		ALEB : OUT STD_LOGIC
	);
END component;

-- Internal signals declaration
SIGNAL div_0_unsigned_res : STD_LOGIC_VECTOR(div_0_unsigned_width_n-1 DOWNTO 0);
SIGNAL compare_0_res : STD_LOGIC;

SIGNAL Resistance : STD_LOGIC_VECTOR(div_0_unsigned_width_n-1 DOWNTO 0) :=
	STD_LOGIC_VECTOR(TO_UNSIGNED(0, div_0_unsigned_width_n));
SIGNAL Inputs_Read : STD_LOGIC := '0';
SIGNAL Temp_Read : STD_LOGIC := '0';

BEGIN
	div_0_unsigned: LPM_DIVIDE
	GENERIC MAP (LPM_WIDTHN => div_0_unsigned_width_n,
		LPM_WIDTHD => div_0_unsigned_width_d,
		LPM_NREPRESENTATION => "UNSIGNED",
		LPM_DREPRESENTATION => "UNSIGNED")
	PORT MAP (NUMER => Voltage_I,
		DENOM => Current_I,
		QUOTIENT => div_0_unsigned_res);

	compare_0_unsigned: LPM_COMPARE
	GENERIC MAP (LPM_WIDTH => compare_0_unsigned_width,
		LPM_REPRESENTATION => "UNSIGNED")
	PORT MAP (DATAA => Resistance,
		DATAB => STD_LOGIC_VECTOR(TO_UNSIGNED(Temp_Threshold,
			compare_0_unsigned_width)),
		AGEB => compare_0_res);
```

```
        M1_Components_library:
        PROCESS (Voltage_I,Current_I,Resistance,Inputs_Read,Temp_Read,
                div_0_unsigned_res,compare_0_res) IS BEGIN

        Read_Inputs:
        IF (Temp_Read = '0') and (Inputs_Read = '0')
        THEN
                Inputs_Read <= '1';
        END IF;

        Resist_Comp:
        IF (Temp_Read = '0') and (Inputs_Read = '1') and
         (not (Current_I =
                STD_LOGIC_VECTOR(TO_UNSIGNED(0,div_0_unsigned_width_n))))
        THEN
                Resistance <= div_0_unsigned_res;
                Temp_Read <= '1';
        END IF;

        Compare:
        IF (Temp_Read = '1') and (Inputs_Read = '1')
        THEN
                Temp_Read <= '0';
                Result_O <= compare_0_res;
                Inputs_Read <= '0';
        END IF;
        END PROCESS;
END a;
```

Appendix B

Replacing Infix Operators with Calls of Functions

```
LIBRARY IEEE;
LIBRARY LPM;
USE IEEE.STD_LOGIC_1164.ALL;
USE LPM.LPM_COMPONENTS.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Infix_Operators_vs_Function_Calls IS
	GENERIC
	(
		mult_0_unsigned_width_a : NATURAL :=  2;
		mult_0_unsigned_width_b : NATURAL :=  2;
		add_0_unsigned_width : NATURAL :=  4;
		div_0_unsigned_width_n : NATURAL :=  5;
		div_0_unsigned_width_d : NATURAL :=  2
	);
	PORT
	(
		--Input ports
		Input1_I : IN STD_LOGIC_VECTOR(mult_0_unsigned_width_a-1 DOWNTO 0) :=
			STD_LOGIC_VECTOR(TO_UNSIGNED(1, mult_0_unsigned_width_a));
		Input2_I : IN STD_LOGIC_VECTOR(mult_0_unsigned_width_b-1 DOWNTO 0) :=
			STD_LOGIC_VECTOR(TO_UNSIGNED(0, mult_0_unsigned_width_b));
		Input3_I : IN STD_LOGIC_VECTOR(add_0_unsigned_width-1 DOWNTO 0) :=
			STD_LOGIC_VECTOR(TO_UNSIGNED(0, add_0_unsigned_width));

		--Output ports
		Result_O : OUT STD_LOGIC_VECTOR(div_0_unsigned_width_n-1 DOWNTO 0) :=
			STD_LOGIC_VECTOR(TO_UNSIGNED(0, div_0_unsigned_width_n))
	);
END Infix_Operators_vs_Function_Calls;

ARCHITECTURE a OF Infix_Operators_vs_Function_Calls IS
component LPM_MULT
	GENERIC(
		LPM_WIDTHA : NATURAL;
		LPM_WIDTHB : NATURAL;
		LPM_WIDTHP : NATURAL;
		LPM_REPRESENTATION : STRING
	);
	PORT(
```

```
                DATAA : IN STD_LOGIC_VECTOR(LPM_WIDTHA-1 DOWNTO 0);
                DATAB : IN STD_LOGIC_VECTOR(LPM_WIDTHB-1 DOWNTO 0);
                RESULT : OUT STD_LOGIC_VECTOR(LPM_WIDTHP-1 DOWNTO 0)
        );
END component;

component LPM_ADD_SUB
        GENERIC(
                LPM_WIDTH : NATURAL;
                LPM_DIRECTION : STRING;
                LPM_REPRESENTATION : STRING
        );
        PORT(
                DATAA : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
                DATAB : IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
                RESULT : OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
                COUT : OUT STD_LOGIC
        );
END component;

component LPM_DIVIDE
        GENERIC(
                LPM_WIDTHN : NATURAL;
                LPM_WIDTHD : NATURAL;
                LPM_NREPRESENTATION : STRING;
                LPM_DREPRESENTATION : STRING
        );
        PORT(
                NUMER : IN STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
                DENOM : IN STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0);
                QUOTIENT : OUT STD_LOGIC_VECTOR(LPM_WIDTHN-1 DOWNTO 0);
                REMAIN : OUT STD_LOGIC_VECTOR(LPM_WIDTHD-1 DOWNTO 0)
        );
END component;

SIGNAL mult_0_unsigned_res : STD_LOGIC_VECTOR((mult_0_unsigned_width_a+
                mult_0_unsigned_width_b)-1 DOWNTO 0);
SIGNAL add_0_unsigned_res : STD_LOGIC_VECTOR(add_0_unsigned_width DOWNTO 0);
SIGNAL div_0_unsigned_res : STD_LOGIC_VECTOR(div_0_unsigned_width_n-1 DOWNTO 0);
SIGNAL Read_Write : STD_LOGIC := '0';

BEGIN
        mult_0_unsigned: LPM_MULT
        GENERIC MAP (LPM_WIDTHA => mult_0_unsigned_width_a,
                LPM_WIDTHB => mult_0_unsigned_width_b,
                LPM_WIDTHP => (mult_0_unsigned_width_a+mult_0_unsigned_width_b),
                LPM_REPRESENTATION => "UNSIGNED")
        PORT MAP (DATAA => Input1_I,
```

```
            DATAB => Input2_I,
            RESULT => mult_0_unsigned_res);

        add_0_unsigned: LPM_ADD_SUB
        GENERIC MAP (LPM_WIDTH => add_0_unsigned_width,
            LPM_DIRECTION => "ADD",
            LPM_REPRESENTATION => "UNSIGNED")
        PORT MAP (DATAA => mult_0_unsigned_res,
            DATAB => Input3_I,
            RESULT => add_0_unsigned_res(add_0_unsigned_width-1 DOWNTO 0),
            COUT => add_0_unsigned_res(add_0_unsigned_width));

        div_0_unsigned: LPM_DIVIDE
        GENERIC MAP (LPM_WIDTHN => div_0_unsigned_width_n,
            LPM_WIDTHD => div_0_unsigned_width_d,
            LPM_NREPRESENTATION => "UNSIGNED",
            LPM_DREPRESENTATION => "UNSIGNED")
        PORT MAP (NUMER => add_0_unsigned_res,
            DENOM => Input1_I,
            QUOTIENT => div_0_unsigned_res);

        Component_test_library:
        PROCESS (Input1_I,Input2_I,Input3_I,Read_Write) IS BEGIN

        Read:
        IF (Read_Write = '0')
        THEN
            Read_Write <= '1';
        END IF;

        Result:
        IF (Read_Write = '1')
        THEN
            Result_O <= div_0_unsigned_res;
            Read_Write <= '0';
        END IF;

        END PROCESS;
END a;
```

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-2869-8
ISSN 1239-1891