Component Reuse in B Using ACL2

Yann Zimmermann, Diana Toma

To cite this version:

Yann Zimmermann, Diana Toma. Component Reuse in B Using ACL2. ZB Formal Specification and Development in Z and B - 4th International Conference of B and Z Users - ZB 2005, Apr 2005, University of Surrey, Guildford, UK, pp.280-299, 10.1007/b135596 . inria-00000755

HAL Id: inria-00000755

https://hal.inria.fr/inria-00000755

Submitted on 16 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Component Reuse in B Using ACL2

Yann Zimmermann[1,2] and Diana Toma[3]

[1] KeesDA SA, 2 av. de Vignate 38610 Gières, France
yann@keesda.com
[2] LORIA, MOSEL team, 54506 Vandoeuvre-lès-Nancy Cedex, France
yann.zimmermann@loria.fr
[3] TIMA, 46 av. Félix Viallet, 38031 Grenoble CEDEX, France
Diana.Toma@imag.fr

Abstract. We present a new methodology that permits to reuse an existing hardware component that has not been developed within the B framework while maintaining a correct design flow. It consists of writing a specification of the component in B and proving that the VHDL description of the component implements the specification using the ACL2 system. This paper focuses on the translation of the B specification into ACL2.

1 Introduction

Electronic systems are becoming more and more complex and they are now involved in a lot of products. Malfunction of an electronic circuit may have financial consequences or take a heavy toll in human life. Some standards, as IEC 61508 [12] or RTCA Do-254/EUROCAE ED-80 [17, 6], have been developed to address this. Our approach using the B method may be used in the parts relating to specifications and validations.

Formal methods are needed to ensure correctness of systems. Formal verification of circuits is often based on model-checking that is limited by the number of states of the system. Symbolic methods such as symbolic trajectory evaluation [10] may improve the efficiency of model-checking. Theorem proving is not limited by the size of the state space that may be sometimes unknown (or parameterised). Examples of successful applications of theorem provers for hardware verification include: ACL2 [21, 18], HOL [7] or PVS [20].

The PUSSEE project [16, 15] has defined a methodology to develop electronic systems by refinement from a very abstract model to its implementation at the register transfer level and translation to hardware description languages (HDL). Event-B [1, 2] is used as formal framework and BHDL®[1] is an implementation level for electronic circuits defined for B (as B0 is an implementation level for software). An example development of a circuit in B can be found in [9]. One

[1] BHDL is a registered trademark of KeesDA.

H. Treharne et al. (Eds.): ZB 2005, LNCS 3455, pp. 280–299, 2005.
© Springer-Verlag Berlin Heidelberg 2005

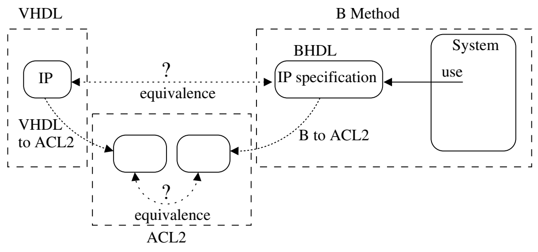

Fig. 1. Using ALC2 for IP reuse in B

issue of this development process is that it does not allow IP[2] reuse. To ensure correctness, components must be fully developed inside the B method and correctly translated into a HDL formalism. Reuse of existing components requires the opposite direction to be integrated into a formal B development. It is possible to translate the HDL description of the component in B, and perform proofs on the B model [4], but this is not the usual direction in B.

In this paper, we suggest using an intermediate approach (see figure 1) by specifying the circuit in B (actually in the sub-language BHDL) at a level of abstraction where the interface of the circuit corresponds to the interface of the IP. Then, this model is translated into ACL2. At the same time, the VHDL description of the IP is also translated into ACL2 [22] and we use ACL2 to prove the equivalence between both models.

This paper describes the translation from B to ACL2. The main step is to flatten the B model. It consists in building a B model where the evolution of each variable is specified using only references to inputs and registers of the circuit, without using any intermediate variable. This allows the construction of a compact model similar to the ACL2 model for a VHDL design. The advantage of doing this transformation in B and not in ACL2 is that we have proved that the flattening process is a B refinement (actually the flat model is equivalent to the original B model).

In the remainder of this paper, we first give a short introduction to circuits and to the three formalisms (VHDL, ACL2 and BHDL) of interest. The translation itself is presented in the next section by defining the notion of flat substitution, and flattening rules are explained. A sketch of the correctness of the transformation is given. We finish by summing up results obtained with case studies before the conclusion. Throughout the paper, we use the example of a simple counter to illustrate the theory.

2 Introduction to Synchronous Circuits

An electronic circuit is an assembly of elementary electronic components connected by wires. Wires carry electronic *signals* that can generally be in two

[2] Intellectual Property, this term refers to hardware sold by design companies. Usually a VHDL description is provided.

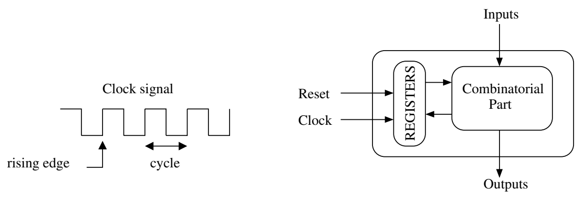

Fig. 2. The Clock signal and a schematic view of a synchonous circuit

states: low-level or high-level. In this case, a signal can be modelled by Boolean values. When a signal may have more than two states, we use other logics. Electronic signals propagate in one direction on a wire; regarding the circuit as a black box, some signals enter the circuit, they are the *inputs* and some signals go out, they are the *outputs* of the circuit. Inputs and outputs taking boolean values, the circuit is modelled by a Boolean function that relates inputs to the outputs. According to the function that it defines, a circuit can be *combinatorial* (outputs are entirely defined by inputs at the same instant) or *sequential* (outputs may depends on the history of inputs). Sequential circuits include memory elements, and are called *synchronous* or *asynchronous* according to the kind of memories being used.

We only address synchronous circuits (or combinatorial circuits if there is no memory at all). A synchronous circuit is a circuit where memory elements are *flip/flops*. There are several kinds of flip/flop, the common principle is that they are driven by a *clock* signal. We call *registers* flip/flops that are sensitive to the rising edge of the clock: the stored value changes only when the clock signal changes from low-level to high-level. Between two rising edges of the clock, the register output does not change, and modifications on the input of the registers are not taken into account until a rising edge occurs. The *clock* signal is cyclic: the time interval between two rising edges is constant (see figure 2).

All registers are supposed to be loaded under to the same clock condition and so they all evolve at the same time, in a *synchronous* way. Some flip/flops can also have another input, called *asynchronous reset* that permits to reset the stored value independently of the clock signal. This signal is set by the environment of the circuit. It must be set to high-level to initialise the circuit when the whole system starts. When initialisation of the system is finished, it must remain indefinitely at low-level (unless the system needs to be reinitialised).

3 VHDL

One of the most used Hardware Description Language is VHDL[11]. We focus on the subset of VHDL that models hardware components at the register transfer level, this means the level where circuits are described using registers and signals.

A VHDL description consists of a list of concurrent processes. The basic process is the signal assignment $s \Leftarrow E$, s is the name of the signal and E is the

```
-- component interface
entity counter is
   port (
      clock : in std_logic;
      reset : in std_logic;
      rst : in std_logic;
      alm3 : out std_logic);
end;
-- component architecture
architecture tab of counter is
-- type of the register
    type tc_type is
   array (0 to 7) of std_logic;
-- declarations of signals
   signal gd : std_logic;
   signal tc_0 : tc_type; ...
begin
-- process that models registers
process (clock, reset) begin
   if reset = '1' then
      tc_0 ⇐ "1000 0000"
   elsif clock'EVENT and clock='1' then
      tc_0 ⇐ tc_1;
   end if;
end process;
```

```
-- Combinatorial part

-- the output alm3 is connected to
-- the last cell of the register tc_0
      alm3 ⇐ tc_0(7);

-- the first bit of tc_6 is '1', other bits '0'
      tc_6(0) ⇐ '1';
   GEN1 : for k in 1 to 7 generate
      tc_6(k) ⇐ '0';
   end generate;

-- gd is '1' if the maximum not reached
      gd ⇐ not (tc_0(7));

-- tc_10 is tc_0 right-shifted
      tc_10(0) ⇐ '0';
   GEN2 : for k in 1 to 7 generate
      tc_10(k) ⇐ tc_0(k - 1);
   end generate;

-- tc_8 is tc_10 unless maximum reached
      tc_8 ⇐ tc_10 when gd='1'
            else tc_0;
-- the register is
-- reinitialised (tc_6) if rst='1'
-- right-shifted if maximum not reached
-- unchanged otherwise
      tc_1 ⇐ tc_6 when rst='1'
            else tc_8;
end;
```

Fig. 3. A VHDL description of a counter

expression that is assigned to the signal. The semicolon ";" denotes concurrent composition: $t \Leftarrow s; s \Leftarrow E$ is the concurrent assignment of s to t and of E to s. Compared to B, ";" has neither the same semantics as in B, nor the same semantics as $\|$ in B. It means that t is connected to s *and* s is connected to E: so t is also, indirectly, connected to E. Writting "A;B" or "B;A" is equivalent.

A more complex process is a block of sequential statements. Inside each process, statements are executed sequentially and it is possible to use local variables. Processes and signal assignments are concurrent. The semantics of concurrency are usually given using delta-delay (see [8] for example). The principle consists of applying assignments repeatedly until a fixed point is reached.

Notice that variables must not be confused with signals, they are two different kinds of objects. Signals usually correspond to wires in a circuit whereas variables are used in processes as a means of programming functionality. When a variable is assigned, its value changes immediately (as is usually the case in programming languages). When a signal is assigned, its value is not modified immediately, only the future value of the signal is modified. For example, we can write s ⇐ '1' after 10ns, '0' after 30ns that means the value of the signal will be set to '1' (high-level) after a delay of 10ns, then set to '0' after 30ns (this means 20ns later). The modification of a signal may also depend on an event, for example we can write s ⇐ '1' when t='0' else '0' that means the signal s will be set to '1' each time the signal t is equal to '0' and s is set to '0' in other cases.

The combinatorial part of the circuit is given by a list of concurrent signal assignments and a register is modelled by two signals and a process that is sensitive to *clock* and *reset* signals. One of these signals carries the current value of the register and the other one carries the next value of the register as specified by the combinatorial part.

As example, we give in Fig. 3 the VHDL code for a counter. The entity part is the interface of the circuit, the process corresponds to registers, and the part on the right corresponds to the combinatorial part of the circuit. In addition to the *clock* and *reset* signals, it has one input rst and one output $alm3$. The signal rst can be used to reinitialise the counter. If rst is '0', the counter is incremented by 1 at each cycle. Here the counter is not implemented as an integer and an adder to increment it. We use a vector of bits (tc) that contains one token. At each cycle (unless rst is set to '1'), the token moves to the next cell (GEN2 and tc_10(0) ⇐ '0'). When the last cell is reached, it does not move anymore until the rst signal is set to '1'. In this case, the token moves to the first cell (tc_6(0) ⇐ '1' and GEN1). Signals tc_0 and tc_1 are respectively the output and the input of the register. Other signals tc_x are intermediate signals.

When the asynchronous *reset* is set to '1', the process specifies that the token moves to the first cell. The output $alm3$ is an alarm set to '1' when the counter has finished to count, that is why $alm3$ is connected to the last cell of *tc_0*.

4 ACL2

ACL2 is a theorem prover based on a first order logic with equality and induction. We chose this theorem prover for its high degree of automation, and reusable libraries of function definitions and theorem proofs [13]. ACL2 is also a programming language based on Common Lisp. Therefore ACL2 models are both executable and provable. Before investing human time in a proof, it is thus possible to check the model on test vectors, a common simulation activity in design verification which helps debugging the formal model and gaining designer's confidence in it. ACL2 has already been used successfully for digital systems verification [14].

ACL2 Model of VHDL. The VHDL is automatically translated into a functional model using a method based on symbolic simulation developed by the VDS group, TIMA Laboratory [22]. The model is simulated symbolically for one clock cycle, actually corresponding to several VHDL simulation cycles, to extract the transition function for each output and state variable of the design. The body of a transition function is a conditional expression, an arithmetic or a

Boolean expression. The functions are translated into Lisp and used to define the Moore machine for the initial VHDL description. Standard VHDL operations on Boolean and bit vectors are replaced with corresponding operations defined and proved correct in ACL2.

Along with the functions above, information about inputs and state variables are translated to Lisp and two predicates are created: hyp-input (input), which states the type for each input element of the design, and hyp-st (st), which states the type for each state variable of the design.

A state of the Moore machine is the set of all internal memories and all the outputs of the design. A step is modeled as a function sim-step which takes as parameters the inputs of the design and the state of the machine at clock cycle k, and which produces the state of the machine at clock cycle k+1 (k is a natural number). The body of sim-step is the composition of the transition functions obtained by symbolic simulation.

Below, the corresponding sim-step function for the VHDL design implementing a counter.

```
(defun vhdl-sim-step (in st )
...
  (list (nextsig_tc_0 reset tc_1)
        (nextsig_tc_1 reset rst tc_1)
        (nextsig_tc_6)
        (nextsig_tc_8 reset tc_1)
        (nextsig_tc_10 reset tc_1)
        (nextsig_gd reset tc_1)
        (nextsig_alm3 reset tc_1))))
```

The nextsig_X function describes the behaviour of signal X during a clock cycle. For instance, here is the body of nextsig_alm3

```
(defun nextsig_alm3 (reset tc_1)
  (nth 7 (if (equal reset 1) (list 1 0 0 0 0 0 0 0)
         tc_1)))
```

The general state machine is defined as a recursive function system that takes a sequence of inputs *l-input* and an initial state *st* and returns the state obtained after consuming all inputs. *l-input* represents the list of symbolic or numeric values for the design's input ports at each clock cycle:

(*inputs_cycle*-1 *inputs_cycle*-2 ... *inputs_cycle*-*k*)

If the inputs list is empty (verified by the ACL2 function atom), the computation is finished and the function returns the state *st*. Otherwise, the next state is computed, and *st* is updated, by calling the step function sim-step. As we mentioned before, the model is also executable.

The funtion vhdl_counter below models the state machine function for the same VHDL design, over a time-sequence of inputs.

```
(defun vhdl-counter (l-input st )
   (if (atom l-input) st
      (vhdl-counter (cdr l-input) (vhdl-sim-step (car l-input) st ) )))
```

5 BHDL

The language BHDL [15–chapter 7] was defined during the PUSSEE project [16]. The goal of the project was to develop a methodology to elaborate systems (including electronic hardware) in B. Based on the same language of substitutions as B, BHDL can be used as a B implementation level for hardware, similar to B0 for software. During the project translators to SystemC and VHDL were also implemented using the logic solver of AtelierB.

We have extended the notion of frame introduced by Dunne [5] by defining for any substitution S a *write frame* (denoted by W_S) and a *read frame* (denoted by R_S). They are respectively the sets of variables that are written and read by the substitution.

5.1 Development of Circuits in EventB

For developing a circuit in B, one may use refinement and formal verification of a system from a very abstract model to the implementation level. The development process is summed up on figure 4. Classically, the initial specification is provided in natural language. Since such a specification is not formal, it may be incomplete, inconsistent on some points or ambiguous. A first step consists in developing a B model that corresponds to this specification. This formal specification is not made in one shot but using the refinement process provided by B. A first abstract model specifies the more general view of the system, then details are added to the specification by refinement. Each element of the specification is introduced at the most abstract level possible, because it is easier to understand for the designer (there is less details) and proofs are easier to handle.

Particularly, abstraction permits to prove algorithms and protocols (see [3] for example) of the design before being overflowed by the details of a hardware implementation level.

When cycle accuracy is needed (as late as possible), it is modelled in an abstract way by synchronising components of the system. *Synchronisation* models the chain of cycles, concurrency and communications between components. The system is refined again to obtain an implementable model. Requirements of the implementation level are: implementable data types, each component is modelled separately and works only with its own state and input/output ports. Cycle accuracy is needed at the end of the development, it is the basis of the semantics of BHDL.

Once an implementable model is reached, the B model is translated in BHDL. An implementable model is fully deterministic, it uses only SELECT guards. The substitution WHILE is not used in BHDL, an implicit global loop corresponds to the succession of cycles. From an eventB model, the BHDL model is obtained by *recomposing* events of each components, implementing the synchronisation and specifying which variables are input and outputs ports.

Recomposition is based on two rules. The first one merges two events into one when their guards are complementary. The second one creates a sequential composition of two events when the first one establishes the guard of the second one.

$$\frac{\begin{array}{l}\textsc{select}\ P \wedge Q\ \textsc{then}\ \mathrm{S}\ \textsc{end}\\ \textsc{select}\ P \wedge \neg Q\ \textsc{then}\ \mathrm{T}\ \textsc{end}\end{array}}{\begin{array}{l}\textsc{select}\ \mathrm{P}\ \textsc{then}\\ \quad \textsc{if}\ \mathrm{Q}\ \textsc{then}\ \mathrm{S}\ \textsc{else}\ \mathrm{T}\ \textsc{end}\\ \textsc{end}\end{array}} \qquad \frac{\begin{array}{l}\textsc{select}\ \mathrm{P}\ \textsc{then}\ \mathrm{S}\ \textsc{post}\ \mathrm{Q}\ \textsc{end}\\ \textsc{select}\ \mathrm{Q}\ \textsc{then}\ \mathrm{T}\ \textsc{end}\end{array}}{\textsc{select}\ \mathrm{P}\ \textsc{then}\ \mathrm{S};\ \mathrm{T}\ \textsc{end}}$$

The BHDL model has formal semantics[3] and can be translated to other formalisms. For example it can be translated to a hardware description language for simulation and synthesis or into a formalism that provides a better support than B for some verification activities, such as for temporal properties.

5.2 The BHDL Language

A BHDL design is an event-B model composed of only one event has no guard. An intuitive behaviour a BHDL design consists of:

- apply once the substitution of INITIALISATION, when the system starts
- then the substitution of the OPERATION clause is applied repeatedly

Relating this to synchronous circuits, the INITIALISATION clause specifies the initialisation of registers and the event specifies the combinatorial part of the circuit. Particularly, outputs of the circuit are specified by the operation clause even at the starting of the system: this means that the state between the initialisation and the first occurrence of the event is not observable. This corresponds to the fact that in a circuit, signals propagates inside the combinatorial part also during initialisation. Both signals *clock* and *reset* are not explicit in BHDL. The sequence of cycles is modelled by the repetitive application of the event.

Two kinds of objects carry values in a circuit: signals and registers. In a BHDL model, they are both modelled by variables. The distinction signal/register is not made explicitly, it is computed automatically using frames.

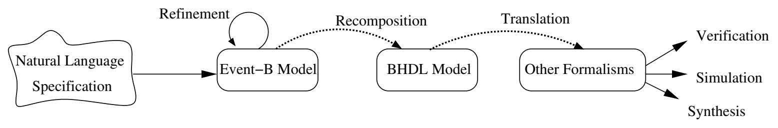

Fig. 4. Development Process

[3] Formal BHDL semantics are not yet published. They are based on before-after predicates: as a sub-language of B, BHDL inherits its semantics. These semantics has been used to ensure correctness of translations from BHDL to SystemC and VHDL.

BHDL uses a subset of B substitutions that is implementable by circuit at the RTL level. Below is a short grammar of substitutions used in this paper. The non-deterministic assignment ($x :\in \mathit{Const}$) is allowed only in the INITIALISATION clause and only constant set (Const) is allowed for this substitution. The term *BExp* refers to B expression and *BoolExp* to Boolean expressions.

$$\begin{aligned}\mathit{Subst} \leftarrow\ & x := \mathit{BExp} \mid x :\in \mathit{Const} \mid \mathit{Subst} \parallel \mathit{Subst} \mid \mathit{Subst}\ ;\ \mathit{Subst} \parallel \\ & \text{IF } \mathit{BoolExp} \text{ THEN } \mathit{Subst} \text{ ELSE } \mathit{Subst} \text{ END} \parallel \\ & \text{IF } \mathit{BoolExp} \text{ THEN } \mathit{Subst} \text{ END}\end{aligned}$$

5.3 Example of BHDL Design

We give the BHDL description of the counter (Fig. 5a). It has the same specification as the VHDL but the implementation differs. A register *compt* stores the current value of the counter. The output *alm* is set to *true* when the counter reaches 7. In this case, the counter remains at 7 until the *rst* input is set to *true*.

In the example of the counter, the register *compt* is initialised to 0.

(a)

```
INITIALISATION
    $compt := 0 \|$
    $rst :\in BOOL \| alm :\in BOOL$
OPERATIONS
    $alm := bool(compt = 7)$
    ;
    IF $rst = true$ THEN
        $compt := 0$
    ELSE
        IF $alm = false$ THEN
            $compt := compt + 1$
        END
    END
```

(b)

```
IF $reset = true$ THEN
    $compt := 0$
END
;
$alm := bool(compt = 7)$
;
IF $rst = true$ THEN
    $compt := 0$
ELSE
    IF $alm = false$ THEN
        $compt := compt + 1$
    END
END
```

Fig. 5. (a) BHDL model of a 3-bit counter; (b) merging of initialisation and operation clauses

6 Translation from BHDL to ACL2

A BHDL design has two important parts, the INITIALISATION clause that specifies how registers are initialised when the *reset* signal of the circuit is set, and the OPERATION clause that specifies the combinatorial part of the circuit.

The ACL2 circuit description used in our approach consists of one function per signal that computes the value of the signal at the end of a clock cycle. A function is also dedicated to simulate a clock cycle by calling all signal functions.

The translation process from BHDL to ACL2 is the following:

1. Convert the design into a design where the initialisation clause and the operation clause are merged into one substitution. A design with an initialisation clause *Init* and an operation clause *Op* is transformed into the substitution below. Moreover, the non-deterministic substitutions of *Init* are removed.

   $$\text{IF } reset = true \text{ THEN Init END ; Op}$$

   The input signal *reset* is introduced explicitly: if the signal *reset* is set then *Init* and *Op* are applied. In the other case, only *Op* is applied. This corresponds to the semantics of a BHDL model: the state between *Init* and the first application of *Op* is not observable. We give in Fig. 5b the event that corresponds to the BHDL description of Fig. 5a: the *reset* signal is made explicit and the initialisation is directly introduced inside the event.

   Notice that a requirement is that the signal *reset* is set to *true* when the system starts and then remains at *false*.
2. Flatten the resulting substitution
3. Translation of the flat substitution into ACL2.

6.1 Flat Form of a Substitution

The sequential substitution makes intermediate results available for reuse in some expressions. The ACL2 model is functional and, in our approach, the outputs are functions of inputs and registers, without any intermediate variables. To generate the ACL2 model, we first *flatten* the BHDL model to remove sequential substitutions. For example, the substitution $x := in + z; out := x + 1$ is first transformed into $x := in + z \| out := in + z + 1$.

In the definition of a flatten substitution, we only refer here to substitutions used in BHDL. A substitution is *flat* when :

– it contains no sequential composition,
– it is a parallel composition of substitutions, each one writing only one variable. Two of these substitutions cannot write the same variable and all variables must be written.

Notice that none of the composed substitutions can contain a parallel composition (because only one variable may be written), nor a sequential composition. So, according to the BHDL language, it can only be a tree of nested IF statements with simple substitutions of the form $v := E$ as leaves.

We can formalise this by giving the following grammar where *BoolExp* is the grammar of predicates, *Exp* of expressions and *var* of identifiers. The predicate $card(W_S) = 1$ is a well-formedness side-condition to ensure that each substitution of the parallel composition only writes one variable. In particular, in the conditional substitution, $S^{(1)}$ and $S^{(2)}$ must write the same variable. The requirement that two substitutions cannot write the same variable is ensured by well-formedness of the parallel composition.

```
IF $reset = true$ THEN
    $alm := bool(0 = 7)$
ELSE
    $alm := bool(compt = 7)$
END

$\|$

IF $reset = true$ THEN
    IF $rst = true$ THEN
        $compt := 0$
    ELSE
        IF $bool(0 = 7) = false$ THEN
            $compt := 0 + 1$
        ELSE
            $compt := 0$
        END
    END
ELSE
    IF $rst = true$ THEN
        $compt := 0$
    ELSE
        IF $bool(compt = 7) = false$ THEN
            $compt := compt + 1$
        ELSE
            $compt := compt$
        END
    END
END
```

Fig. 6. Flat substitution of the counter

$$\begin{aligned} \mathit{FlatS} &\leftarrow S \qquad\qquad\qquad\qquad card(W_S) = 1 \\ &\mid \mathit{FlatS} \parallel \mathit{FlatS} \\ S &\leftarrow \text{IF } \mathit{BoolExp} \text{ THEN } S^{(1)} \text{ ELSE } S^{(2)} \text{ END} \\ &\mid var := \mathit{Exp}. \end{aligned}$$

Example of Flat Substitution. To illustrate how a BHDL model is transformed, we give on figure 6 the flat form of the counter given in section 5.3. It consists of two substitutions composed in parallel. The first one specifies the evolution of the variable *alm* and the second one the variable *compt*. Each one depends only on inputs (*reset* and *rst*) and registers (*compt*). In particular, the expression of *compt* not longer depends on the variable *alm*.

6.2 Translation of the Flat Substitution into ACL2

The third step is easy after flattening, it just consists of rewriting the substitution using the ACL2 syntax. The syntax of the substitution respects the grammar given in previous section. Translation into ACL2 is done in this simple way by the operator *acl2* defined below. This operator applies on flattened substitutions. Substitutions S_1 ... S_n, S and T stand for substitutions that do not contain any parallel composition. They are simple substitutions or (flattened) conditional substitutions. We use u_k to denote the name of the variable written by the substitution S_k, it is used to give a name to the ACL2 created function ($\mathsf{B\_}u_k$).

$acl2(S_1\|...\|S_n) =$
  for each substitution S_k, this ACL2 function is created:
  (defun B_u_k $acl2(S_k)$)
    where $\{u_k\} = W_S$, u_k is the variable written by S_k
$acl2($IF C THEN S ELSE T END$) =$ (if $acl2(C)$ $acl2(S)$ $acl2(T)$)
$acl2(v := E) =$ ($acl2exp(E)$)
where $acl2exp$ is the translation of a B expression into an ACL2 expression.

The translation of the counter given in sections 5.3 and 6.1 producesthe following ACL2 functions.

```
(defun B_alm (compt reset)
  (if (equal reset 1)
    (if (equal 0 7) 1 0)
      (if (equal compt 7) 1 0))

(defun B_compt (compt rst reset)
  (if (equal reset 1)
    (if (equal rst 1)
      0
      (if (equal (equal 0 7) nil) (+ 0 1) 0 )
    )
    (if (equal rst 1)
      0
      (if (equal (equal compt 7) nil) (+ compt 1) compt )
    )))
```

7 Flattening

Flattening a substitution S builds another substitution that has the same effect as S but that is flat. The main transformation consists of removing sequential compositions $(S; T)$ by propagating effects of the first substitution (S) inside the second one (T). After the definitions of flattening rules, we give a sketch of the proof that the flattening process constructs a substitution that is equivalent to the original one.

7.1 Flattening Rules

The process is based on three operators. The main operator *flat* flattens a substitution. It uses operators *extract* and *integrate*. The operator $extract(v, S)$ gives a substitution that has the same effect as S on the variable v but that has exactly $\{v\}$ as the write frame. The operator $integrate(S, T)$ integrates the substitution S inside the substitution T: if T reads a variable v that is written by S, it reads v as it is after the application of S.

In this section, we use the notation $\|_{v\in E}S(v)$ to denote the parallel composition of substitutions $S(v)$ for each variable v in the set of variables E. If $E = \{v_1, ..., v_k\}$ then $\|_{v\in E}S(v) = S(v_1)\|...\|S(v_k)$.

Flattening of a Substitution. A simple substitution $v := E$ is already flat and a parallel composition is flat if both composed substitutions are flat.

$$flat(v := E) = v := E$$
$$flat(A \| B) = flat(A) \| flat(B)$$

In a conditional substitution, both alternative substitutions may write several variables. A flat conditional substitution writes only one variable. In consequence, the transformation rule creates one conditional substitution for each written variable and composes them in parallel ($\|_{v \in W_A \cup W_B}$). In the expression $extract(v, flat(A)))$, $flat(A)$ is flat, it is a parallel composition of substitutions, each one writing only one variable. The operator *extract* select the one that writes the variable v.

$flat(\textsc{if}\ C\ \textsc{then}\ A\ \textsc{else}\ B\ \textsc{end}) =$
$\quad \|_{v \in W_A \cup W_B} \textsc{if}\ C\ \textsc{then}\ extract(v, flat(A))\ \textsc{else}\ extract(v, flat(B))\ \textsc{end}$
$flat(\textsc{if}\ C\ \textsc{then}\ A\ \textsc{end}) =$
$\quad \|_{v \in W_A \cup W_B} \textsc{if}\ C\ \textsc{then}\ extract(v, flat(A))\ \textsc{else}\ v := v\ \textsc{end}$

Sequential composition does not exists in the flat form. It must be transformed into an equivalent flat substitution. This is achieved by propagation of transformations specified by the first substitution inside the second substitution.

The principle for flattening the substitution $S; T$ is the following. For any variable v written by S and read by T, the value of v used by T is the value of v after applying S, i.e. v is substituted in T by the expression specified by S. For example $x := E; x := x + 1$ is transformed into $x := E + 1$.

This transformation is achieved by the operator *integrate*, which is defined in the remainder of this section. It returns a flat substitution that has the same write frame as T. This means that variables that are written by S but not by T are not written by the result of the integration. So, we add the flat substitution $S_{/W_S - W_T}$ that have the same behaviour as $flat(S)$ on $W_S - W_T$ and for which the write frame is exactly $W_S - W_T$ (see operator *extract* below).

$$flat(S; T) = S_{/W_S - W_T} \| integrate(flat(S), flat(T))$$

Extraction from a Substitution. The operator $extract(v, S)$ gives a substitution that has the same effect as S on the variable v and for which the write frame is exactly $\{v\}$. The operator *extract* is defined here only on flat substitutions. This simplifies definitions because in a flat substitution, each substitution of the parallel composition writes only one variable. So, extraction simply consists of looking for the substitution that corresponds to the variable we want to extract.

$$extract(v, S) = v := v \text{ if } v \notin W_S$$
$$extract(v, S_1 \| ... \| S_n) = S_k \text{ where } W_{S_k} = \{v\}$$

Integration. The operator $integrate(S, T)$ propagates the effects of S inside T: if a variable is modified by S and used by T, T is transformed and uses the new value of this variable as specified by S. The operator *integrate* is defined here only for flat substitutions, this allows some simplifications in the definition.

Integrate a simple substitution $x := E$ consists of applying this substitution on all expressions. For example, propagate the substitution $x := 2$ inside $x := x + 1$ leads to the substitution $x := 2 + 1$.

$$\begin{gathered} integrate(x := E, y := F) = y := [x := E]F \\ integrate(x := E, \text{IF } P \text{ THEN } S \text{ ELSE } T \text{ END}) = \\ \text{IF } [x := E]P \text{ THEN } integrate(x := E, S) \text{ ELSE } integrate(x := E, T) \text{ END} \end{gathered}$$

If we integrate a substitution that writes some variables v inside a substitution that does not use v at all, the integration has no effect. For example, integrate $x := 2$ inside $x := y$ produces the substitution $x := y$.

$$integrate(S, T) = T \text{ if } W_S \cap R_T = \emptyset$$

Integrate a substitution S in a substitution that is a parallel composition consists of integrating S in all composed substitutions. For example, integration of $x := 2$ inside $x := x + 1||y := x - 2$ produces $x := 2 + 1\|y := 2 - 2$.

$$integrate(S, A\|B) = integrate(S, A)\|integrate(S, B)$$

Integration of a parallel substitution $A\|B$ into a substitution T consists of integrating A and B. Since some variables modified by B may be used by A and vice versa, we cannot first integrate A and then B. For example, if B contains $x := E$ and A contains $y := x$, integrating $A\|B$ in $z := y+x$ produces $z := x+E$. If we first integrate A, we obtain $z := x + x$ that produces $z := E + E$ after integrating B.

For confidentiality reasons, we do not give the exact rule we used to implement the integration of the parallel substitution. A possibility is to use transformation rules of the B Book [1] page 310 to transform $A\|B$ into a substitution in which only simple substitutions are composed in parallel ($x := E\|y := F$). The resulting substitution can be integrated using above rules (the case of the simple substitution can be easily generalised to multiple simple substitutions $x, y := E, F$).

For example, the integration of $S \equiv x := y + 1\|$IF $x = y$ THEN $y := x + 2$ ELSE $y := 1$ END inside $T \equiv x := x + y$ produces the substitution IF $x = y$ THEN $x := y + 1 + x + 2$ ELSE $x := y + 1 + 1$ END.

Simplifications. Flattening rules creates a lot of useless nested IF statements. This is because in B, except in the flat form, an IF substitution may contain several simple substitutions ($v := E$). The operator *flat* splits them into several IFs and *integrate* spreads and nests them. The result is a substitution that grows exponentially.

To simplify these substitutions, useless branches of an IF tree are cut and IF substitutions are simplified when both branches are equal. Substitution S^s (resp. T^s) is the result of the simplification of S (resp. T).

```
$simpl(C, NC, \text{IF } P \text{ THEN } S \text{ ELSE } T \text{ END}) =$
        if $P \in C$ then $simpl(C, NC, S)$
        else if $P \in NC$ then $simpl(C, NC, T)$
        else
         let $S^s = simpl(C \cup \{P\}, NC, S)$ and $T^s = simpl(C, NC \cup \{P\}, T)$ in
            if $S^s = T^s$ then $S^s$
            else IF $P$ THEN $S^s$ ELSE $T^s$ END
```

The parameter C is the set of conditions known to be true and NC the set of conditions known to be false. They come from the fact that the substitution under simplification takes place in a branch of an IF tree. Initially $C = NC = \emptyset$.

Implementation. Flattening has been implemented in Prolog. Experiments have shown the usefulness of simplifications. Without it, constructed substitutions become larger and larger during the flattening process. To be useful, simplifications must be applied regularly during the flattening process, or be directly integrated inside the flattening rules.

Without any simplification, flattening of a design of about 200 lines, needs more that 100MB of RAM and the process takes several hours to complete. With simplifications, flattening of the same design takes about 1 second.

7.2 Flattening is a Refinement

Flattening is an automatic refinement. Actually, the result of the flattening is a substitution that is equivalent to the original substitution. For any substitution S, and any predicate Q

$$[flat(S)]Q \Leftrightarrow [S]Q$$

For reasons of space, we cannot give the entire proof of this property here. However we give below a sketch of the proof for the interesting case $flat(S;T)$, based on the operator *integrate*.

Intuitively, the meaning of $integrate(S, T)$ is to produce a substitution that is consistent with $S;T$ but with the same write frame as T. With respect to the variables written by T, $integrate(S, T)$ is equivalent to $S;T$, but for the variables that are not written by T, $integrate(S, T)$ is equivalent to *skip*.

Let S and T be two flat substitutions. Let x_t be a variable written neither by S nor by T (x_t is a fresh variable) and w_t a variable written by T (possibly also written by S). We can prove the following property.

$$[S;T](x_t = w_t) \Leftrightarrow [integrate(S, T)](x_t = w_t) \quad (1)$$

The proof cannot be given here, it is based on a double recurrence on both arguments (on the structure of substitutions) of *integrate*.

Suppose there exists some variables written by S but not by T, we choose a variable w_{s-t}. Let x_{s-t} be a variable written neither by S nor by T. The property below holds.

$$[S;T](x_{s-t} = w_{s-t}) \Leftrightarrow [S](x_{s-t} = w_{s-t}) \quad (2)$$

Let Q be a predicate on variables written by S or T, we denote it by $Q(w_t, w_{s-t})$ where w_t is a variable written by T and w_{s-t} is a variable written by S but not by T (we make the assumption it exists). The predicate $Q(w_t, w_{s-t})$ may be rewritten $x_t = w_t \wedge x_{s-t} = w_{s-t} \wedge Q(x_t, x_{s-t})$, where x_t and x_{s-t} are two fresh variables. There is an implicit existential quantifier $\exists(x_t, x_{s-t}).(...)$.

We know that

$$[S;T]Q(w_t, w_{s-t}) \Leftrightarrow [S;T](x_t = w_t) \wedge [S;T](x_{s-t} = w_{s-t}) \wedge [S;T]Q(x_t, x_{s-t}) \tag{3}$$

Suppose we have a substitution S' such that its write frame is $W_S - W_T$ and that, for any predicate P on the same frame,

$$[S']P \Leftrightarrow [S]P \tag{4}$$

For any substitution A that has a disjoint frame from S' (W_T for example), we can say that $[S' \| A](x_t = w_t) \Leftrightarrow [A](x_t = w_t)$ because neither x_t nor w_t is written by S'. In particular, the following property holds.

$$[S' \| \mathit{integrate}(S,T)](x_t = w_t) \Leftrightarrow [\mathit{integrate}(S,T)](x_t = w_t) \tag{5}$$

In the same way, because variables x_{s-t} and w_{s-t} are not in the write frame of the substitution $\mathit{integrate}(S,T)$, we have to property below.

$$[S' \| \mathit{integrate}(S,T)](x_{s-t} = w_{s-t}) \Leftrightarrow [S'](x_{s-t} = w_{s-t}) \tag{6}$$

From (1) and (5), we deduce (7), and from (2), (4) and (6) we deduce (8).

$$[S;T](x_t = w_t) \Leftrightarrow [S' \| \mathit{integrate}(S,T)](x_t = w_t) \tag{7}$$

$$[S;T](x_{s-t} = w_{s-t}) \Leftrightarrow [S' \| \mathit{integrate}(S,T)](x_{s-t} = w_{s-t}) \tag{8}$$

Finally, from (3), (7), (8) and because there is no variable written by $S' \| \mathit{integrate}(S,T)$ in $Q(x_t, x_{s-t})$, we deduce the property (9).

$$[S;T]Q(w_t, w_{s-t}) \Leftrightarrow \begin{cases} [S' \| \mathit{integrate}(S,T)](x_t = w_t) \wedge \\ [S' \| \mathit{integrate}(S,T)](x_{s-t} = w_{s-t}) \wedge \\ [S' \| \mathit{integrate}(S,T)]Q(x_t, x_{s-t}) \end{cases} \tag{9}$$

From (9), we can recompose Q to obtain the property below.

$$[S;T]Q(w_t, w_{s-t}) \Leftrightarrow [S' \| \mathit{integrate}(S,T)]Q(w_t, w_{s-t}) \tag{10}$$

The predicate Q is on frame of $S;T$ (that is the same frame as $S' \| \mathit{integrate}(S,T)$). We can generalise to any predicate Q because $S;T$ and $S' \| \mathit{integrate}(S,T)$ have no effect on other variables. The reasoning above uses a predicate Q on two variables w_t and w_{s-t}, it can be generalised to two sets of variables. We made the assumption that $W_S - W_T$ is not empty, the case where all variables written by S are also written by T is a simpler case that leads to the same result.

We also made the assumption of the existence of S' that has $W_S - W_T$ as write frame and such that $[S']P \Leftrightarrow [S]P$ for any predicate P on the same frame. These requirements are met by $S_{/W_S - W_T}$ used in the definition of $flat(S;T)$. In consequence, with $S' \equiv S_{/W_S - W_T}$ and (10), we can conclude that $flat(S;T)$ is equivalent to $S;T$.

$$[S;T]Q \Leftrightarrow [flat(S;T)]Q$$

8 Case Studies

We present two case studies. The first one illustrates the methodology on a non trivial example. The second one uses the example of the counter to explain how the verification is done in ACL2.

8.1 Controller for a Serial Bus

The first case study concerns a controller for a serial bus (standard SAE J1708 [19]): several components linked by a serial bus may send messages to other components using the bus. Each component has a controller that is responsible for sending messages bit by bit on the bus and for dealing with contentions (when two controllers attempt to send a message at the same time).

The B development [23] consisted of first modelling the whole system at a very abstract level to specify important properties. Refinement was used to derive a model of the protocol (so, proved by refinement) and finally the system was refined again to obtain the description of controllers at the register transfer level. This is the level of BHDL. From BHDL, the circuit was translated to VHDL, simulated and synthesised.

The goal of this case study is not to validate the VHDL description of the circuit but to confirm the methodology with a non trivial example that we know to be correct. It is also a chance to associate three translators developed separately. We started from the BHDL description of the circuit validated in B. This description was translated twice: into ACL2 (as described in this paper) and in VHDL (using the translator developed by KeesDA). The VHDL description has around 400 lines and uses 140 internal signals. Then, this VHDL description was translated to ACL2 using the translator developed by TIMA.

At this point we have two ACL2 descriptions of the same circuit and we want to verify that they are equivalent. Equivalence is expected because both ACL2 descriptions come from the same BHDL description. The three translators are based on three different approaches and they have been validated separately.

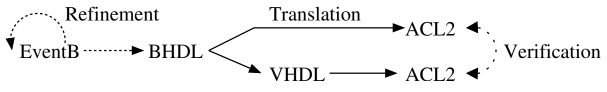

Fig. 7. Connection of translators

The fact that proof of equivalence can be done easily gives confidence in the implementation of these translators and confirms the methodology consisting of using ACL2 as an intermediary between B and VHDL.

The ACL2 model of the VHDL design has 148 functions and the ACL2 model of the BHDL description has 21 functions. For the equivalence proof we defined 65 theorems, and we used an already defined library about bit vectors and operations on bit vectors. The proof process was not difficult, it only took several hours of human time to complete it (against several weeks for the original B development). The proof itself is done in 17.23 minutes on a processor Ultra Sparc 3, 1.28 GHz, with 8GB memory.

Models were modified by hand to introduce some errors, particularly on types and arithmetic expressions. This permits one to check that not only purely functional errors are detected but also errors due to incompatible implementation of data (for example, an integer that may be valued to 8 cannot be implemented by a 3-bits vector). Errors were detected because some conjectures were shown to be false by ACL2.

8.2 Counter

We applied the ACL2 verification to the example of the counter. After the translation of the BHDL model to ACL2, the corresponding sim-step, system, hyp-input and hyp-st functions are defined.

We give below the sim-step and system functions for the BHDL model of the counter.

```
(defun b-sim-step (in st)
  (let ((reset (nth *b-reset* in)) (rst (nth *b-rst* in)) (compt (nth *b-compt* st)))
     (list (B_compt compt rst reset)
         (B_alm compt reset))))
(defun b-counter (input st)
  (if (atom input) st (b-counter (cdr input) (b-sim-step (car input) st))))
```

At this point, there are two ACL2 models : one corresponding to the VHDL design, and the other one to the BHDL model. Both models are cycle accurate.

To prove the bisimulation relation between the two models, a relation $Sim \subseteq ST_{VHDL} \times ST_{BHDL}$ is first defined, where ST_{VHDL} is the set of VHDL design states, and ST_{BHDL} is the set of BHDL model states. The proof that Sim is a simulation relation is done in two steps: (1) starting from arbitrary states, after a clock cycle, when reset is 1, both models are in similar states, conform to Sim; (2) starting from similar states, *st-b* and *st-vhdl*, where (*st-vhdl*, *st-b*) $\in Sim$, after consuming the same inputs (taking into considerations the necessary type conversions), the two models are in similar states :

$$(vhdl\text{-}system(inputs, st\text{-}vhdl),\ b\text{-}system(inputs, st\text{-}b)) \in Sim$$

This is proved by induction on the number of clock cycles, i.e. the length of the list of inputs. The base case states that sim-step functions preserve the similarity.

A second relation $Sim^{-1} \subseteq ST_{BHDL} \times ST_{VHDL}$ is defined and proved to be the inverse of Sim. Likewise, Sim^{-1} is proved to be a simulation relation between the BHDL model and the VHDL model.

Finally, Sim is proved to be a bisimulation relation between the BHDL model and the VHDL design.

For the counter, Sim is defined as follows:

$$(\mathit{st\text{-}vhdl}, \mathit{st\text{-}b}) \in \mathit{Sim} \Leftrightarrow ((\mathit{alm}3 = \mathit{alm}) \wedge (\mathit{get}\text{-}1\text{-}\mathit{pos}(\mathit{tc\_}1) = \mathit{compt}))$$

Where $\mathit{st\text{-}vhdl} = (\mathit{tc\_}0, \mathit{tc\_}1, \mathit{tc\_}6, \mathit{tc\_}8, \mathit{tc\_}10, gd, rst, \mathit{alm}3)$ and $\mathit{st\text{-}b} = (\mathit{compt}, \mathit{alm})$. The function get-1-pos takes a bit-vector as input and returns the position of 1 in the vector. For example, $\mathit{get}\text{-}1\text{-}\mathit{pos}((00100)) = 2$

Sim^{-1} is defined as follows:

$$(\mathit{st\text{-}b}, \mathit{st\text{-}vhdl}) \in \mathit{Sim}^{-1} \Leftrightarrow \\ ((\mathit{alm} = \mathit{alm}3) \wedge (\mathit{tc\_}1 = \mathit{construct\text{-}table}(\mathit{compt})))$$

The function *construct-table* takes a natural n as input and returns a bit vector of size 8 with bit 1 on the n-th position, all other bits being 0.

The proof uses ACL2 libraries about naturals and lists included in the public distribution of the theorem prover. It also uses a library about bit-vectors that was previously developped for hardware verification.

9 Conclusion

We have presented a new methodology to reuse existing components that have not been developed within the B framework.

The principle consists of writing a specification of the component in B and proving that this specification corresponds to the component using ACL2. To achieve this, both BHDL and VHDL descriptions of the component are translated into ACL2. ACL2 is used to prove that both models are equivalent.

Translation of the BHDL into ACL2 needs to flatten the BHDL model. Translation rules have been explained and we have proved that this transformation leads to a model that is equivalent to the original one.

The methodology has been applied to a non trivial case study to verify its efficiency. It was also a chance to combine three translators that have been developed separately and with different approaches. Experiments have shown that non equivalence of models is detected by this methodology.

References

[1] J.-R. Abrial. *The B-Book – Assigning programs to meanings.* CUP, 1996.

[2] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B-Method*, volume 1393 of *LNCS*, pages 83–128, 1998.

[3] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3):215–227, Apr 2003.

[4] Ammar Aljer, Philippe Devienne, Sophie Tison, Jean-Louis Boulanger, and Georges Mariano. B-HDL: Circuit Design in B. In *ACSD 2003, International conference on Application of Concurrency to System Design*, pages 241–242, 2003.

[5] Steve Dunne. A Theory of Generalised Substitutions. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002*, volume 2272 of *LNCS*, pages 270–290, 2002.

[6] European Organisation for Civil Aviation Equipment. , http://www.eurocae.org/.

[7] A.C.J Fox. Formal specification and verification of ARM6. In D. Basin and B. Wolff, editors, *TPHOLs '03*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.

[8] Max Fuchs and Michael Mendler. A Functional Semantics for Delta-Delay VHDL Based on FOCUS. In C. D. Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 9–42. Kluwer Academic Publishers, 1995.

[9] Stefan Hallerstede and Yann Zimmermann. Circuit Design by refinement in EventB. In *Proc. of FDL'04*, 2004.

[10] Scott Hazelhurst and Carl-Johan H. Seger. Symbolic Trajectory Evaluation. In T. Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 3–78. Springer-Verlag, 1997.

[11] IEEE, editor. *Standard VHDL - Language Reference Manual.* IEEE Computer Society Press, USA, 1988.

[12] International Electrotechnical Commission. , http://www.iec.ch/61508/.

[13] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided reasoning: ACL2 An approach*, volume 1. Kluwer Academic Press, 2000.

[14] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided reasoning: ACL2 Case Studies*, volume 2. Kluwer Academic Press, 2000.

[15] J. Mermet, editor. *UML-B - Specification for Proven Embedded Systems Design.* Kluwer Academic Publishers, 2004. ISBN 1-4020-2866-0.

[16] PUSSEE project IST-2000-30103. http://www.keesda.com/pussee/, 2004.

[17] Radio Technical Commission for Aeronautics. , http://www.rtca.org/.

[18] D. M. Russinoff. A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon Processor. In *FMCAD 2000*, 2000.

[19] SAE International. SAE J1708 revised OCT93, serial data communication between microcomputer systems in heavy-duty vehicule applications, www.sae.org, 1993.

[20] M. Srivas, H. Rueß, and D. Cyrluk. Hardware Verification Using PVS. In T. Kropf, editor, *Formal Hardware Verification: Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 156–205. Springer-Verlag, 1997.

[21] D. Toma and D. Borrione. SHA formalization. In *ACL2 Workshop*, USA, 2003.

[22] D. Toma, D. Borrione, and G. Al-Sammane. Combining several paradigms for circuit validation and verification. In *CASSIS*, 2004.

[23] Yann Zimmermann, Stefan Hallerstede, and Dominique Cansell. Formal modelling of electronic circuits using event-B, case study : SAE J708 serial communication link. In Jean Mermet, editor, *UML-B - Specification for Proven Embedded Systems Design.* Kluwer Academic Publishers, 2004.