Electronic Communications of the EASST
Volume 35 (2010)

Proceedings of the 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010)

Integrating Formal Methods with Informal Digital Hardware Development

Neil Evans

16 pages

Guest Editors: Jens Bendisposto, Michael Leuschel, Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Integrating Formal Methods with Informal Digital Hardware Development

Neil Evans

neil.evans@awe.co.uk
AWE, Aldermaston,
United Kingdom

Abstract: This paper presents some results from an industrial project to develop high-integrity digital hardware by integrating formal methods with a more traditional informal approach. The ultimate goal of the project team was to produce sythesisable VHDL that could be proven to meet given requirements for an embedded controller. The burden was on the formal methods experts to integrate themselves into the team. This paper describes the formal approach that was developed as a result.

Keywords: Refinement, VHDL, CSP, The B Method, CSP ‖ B

1 Introduction and Background

Integrating formal methods practitioners into teams comprising engineers from other disciplines is a challenge because formal methods tend to demand complete allegiance from their users, which is perhaps unrealistic in such a context. This paper presents firsthand experience of a multi-discipline project to build an embedded controller. Digital hardware experts employed an informal approach to write VHDL and, consequently, the formal methods experts were tasked to verify the resulting code with respect to a set of informal requirements.

Because of the nature of the requirements, CSP [Hoa85] was chosen as the formal language. A translation from VHDL to CSP is developed using the CSP ‖ B approach [ST05], which is a combination of CSP and the B Method [Abr96]. In particular, traditional CSP ‖ B 'lifting' techniques [ST05] are employed to produce a CSP representation of the VHDL code via B. The approach is repeatable, scalable and could be automated. Automation minimises the risks involved in adopting such an approach by eliminating the need for a deep understanding of formal methods.

1.1 Very High Speed IC Hardware Description Language (VHDL)

VHDL [IEE02] is a language for describing digital electronic circuits. The language is rich in structure and other syntactic sugar. *Elaboration* removes the syntactic sugar to leave a set of concurrent *processes* connected via *signals*. Each process typically includes signal assignment statements that change the value of signals, and is accompanied by a *sensitivity list* of signals which causes the process to react to changes to signals in the sensitivity list.

The semantics of elaborated VHDL code is based on the execution of the code during simulation (as defined in [IEE02]). Each simulation cycle consists of two phases: a process execution

phase and a signal update phase. The notion of a *delta-delay* distinguishes simulation cycles that occur at the same global clock time. A simulation begins by initiating a process execution phase in which every process is active. A process execution phase ends when all active processes have suspended. Signal updates (arising from signal assignments) take place during the subsequent signal update phase. Whenever a signal is updated with a new value, an *event* is said to have occurred. A process reacts to such an event if the signal is present in its sensitivity list, and it resumes its execution during the next process execution phase. In this way signal updates drive the execution of processes, and the execution of processes drive the signal updates. Note, it is possible for multiple processes to update the same signal during the same process execution phase. This situation is resolved in VHDL with so-called *resolution* functions. Although this does not arise in the example presented below, it is not precluded by the approach taken in this paper.

For illustrative purposes, we shall consider a road junction controlled by traffic lights. The VHDL controller can be in one of eight possible states:

- `both_red`, in which all traffic lights are red.
- `major_ready`, in which the major road traffic is forewarned that it can go.
- `major_go` allows traffic on the major road to proceed.
- `major_end`, in which the major road traffic is forewarned that it must stop.
- `swap`, in which control transfers from the major road to the minor road.
- `minor_ready`, in which the minor road traffic is forewarned that it can go.
- `minor_go` allows traffic on the minor road to proceed.
- `minor_end`, in which the minor road traffic is forewarned that it must stop.

The process `fsm` is responsible for assigning values to the light signals. It is defined as a case statement on the signal `state`. The (abbreviated) process is:

```
process fsm(state, ready_delay, grn_delay, end_delay)
begin
  case (state) IS
    when both_red =>
      red_maj <= '1';
      yel_maj <= '0';
      grn_maj <= '0';
      red_min <= '1';
      yel_min <= '0';
      grn_min <= '0';
      next_state <= major_ready;
    when major_ready =>
      yel_maj <= '1';
```

```
      if (ready_delay = '1') then
        next_state <= major_go;
      end if;
    when major_go => ...
    when major_end => ...
    when swap => ...
    when minor_ready => ...
    when minor_go => ...
    when minor_end => ...
  end case;
end process;
```

The `when` clauses delimit the cases, and signals are assigned using the <= operator. In addition to the light signals, `fsm` updates the value of `next_state`. This can depend on the value of one of the delay signals (described below) as well as `state`. Consequently, the sensitivity list of the process includes the signal `state` and the delay signals. Accompanying the process `fsm` is the process `new_state`:

```
process new_state(clock)
begin
  if (clock'event and clock = '1') then
    if (reset = '1') then
      state <= both_red;
    else
      state <= next_state;
    end if;
  end if;
end process;
```

which is sensitive to changes to the signal `clock`. The process checks for a rising edge of `clock` (i.e., a change from 0 to 1) and, depending on the synchronous `reset`, sets `state` to `both_red` or the value of `next_state`.

The `ready_delay` signal determines how long the controller waits in a ready state, and `end_delay` determines how long the controller waits in an end state. The following process increments the signal `yel_count` if one of the yellow signals is 1. The `ready_delay` and `end_delay` signals are dependent on its value.

```
process yellow_counter(clock)
begin
  if (clock'event and clock = '1') then
    if (reset = '1') then
      yel_count <= "000";
    else
      if (yel_maj = '1' or yel_min = '1') then
```

```
        yel_count <= unsigned(yel_count) + 1;
      else
        yel_count <= "000";
      end if;
    end if;
  end if;
end process;
```

Note that `yel_count` is defined to be a 3-bit bit vector. The `ready_delay` and `end_delay` signals are assigned by the following (one line) processes:

```
ready_delay <= yel_count(1);

end_delay <= yel_count(2);
```

The `ready_delay` signal is assigned to be the middle bit of the bit vector, and `end_delay` is assigned to be the leftmost bit. The final two processes are defined similarly, in which `grn_delay` (and `grn_count`) determine how long the controller remains in a green state.

```
process green_counter(clock)
begin
  if (clock'event and clock = '1') then
    if (reset = '1') then
      grn_count <= "0000000";
    else
      if (grn_maj = '1' or grn_min = '1') then
        grn_count <= unsigned(grn_count) + 1;
      else
        grn_count <= "0000000";
      end if;
    end if;
  end if;
end process;

grn_delay <= grn_count(6);
```

1.2 Communicating Sequential Processes (CSP)

The language of CSP has its own notion of *process* and *event*. A CSP process is a formal object which interacts with its environment by performing atomic events. A notion of input and output can be introduced by allowing structured events: the process $c!v \rightarrow P$ outputs the value v on channel c and then behaves as P, and the process $c?x \rightarrow P_x$ is prepared to input any value x (of c's type) and then behave as the process P_x. Both input and output can occur within the same event (as in the event $d?x!y$).

If P and Q are processes then $P \,\Box\, Q$ is a process that behaves as P or Q (the choice is made by the environment), and $P \,|[A]|\, Q$ is the parallel composition of P and Q such that they synchronise

on the events in set A. The hiding operator removes events from a process interface: $P \setminus A$ is the process that behaves like P except the events in A are no longer visible. Event renaming also changes the visible events of a process. *Relational renaming* [Sch99] uses a binary relation to transform events: if P can perform an event a and R is binary relation that contains the pair (a,b) then $P[\![R]\!]$ can perform b instead.

Many semantic models exist for CSP, but we focus on those associated with CSP's model checker FDR [For]: the *traces* model, the *failures* model and the *failures-divergences* model (see [Hoa85]).

1.3 The B Method

The B Method is a formal approach to specifying, designing and implementing software. A specification consists of one or more modular units called *machines*. Each machine comprises a set of local state variables, an invariant which defines the properties of the variables, and a set of operations which modify the variables. An **INITIALISATION** clause gives the variables their initial values.

Various structuring mechanisms are available that allow machines to affect other machines within a specification. A **SEES** clause gives read access to another machine, whilst **INCLUDES** also allows a machine to change the values of another machine's variables, but only via the included machine's operations.

1.4 CSP $\|$ B

In CSP $\|$ B the events of a CSP process trigger operation calls of a B machine. Structured events are used to pass values between the controller process and the B machine. An event $e!x?y$ corresponds to an operation call $y \longleftarrow e(x)$ that inputs x and outputs a value y. A CSP $\|$ B specification can have multiple process/machine pairs [ST02].

CSP $\|$ B primarily makes use of FDR for analysis. As we shall see in Section 3.3, to do so it is often necessary to 'lift' state information from a B machine to a CSP process. In order to mimic the behaviour of a B machine in CSP, we parameterise a process P with state information i, to produce an augmented process P_i. Then the events of P_i are decorated with assertions that relate i to their input values. For example, the assertion $\{x > i\}$ in a decorated event $e?x\{x > i\}$ of P_i says the value input on e must be greater than i, otherwise the process diverges. The purpose of such assertions is to expose problems in the original CSP $\|$ B model by using FDR to find behaviours that violate them.

2 Formalising Requirements

For small project teams, requirements engineers will typically have other roles in the implementation of the system and, hence, will have a preconceived bias towards a particular solution. This will probably result in an overly prescriptive set of requirements. CSP is a good language for formalising such requirements because its operators give an explicit representation of control flow. As an example, we construct a CSP specification of a traffic controller which simply describes our everyday experience of traffic lights (in the UK):

$$SPEC\_RR = both\_red\_to\_maj\_red\_yel \rightarrow SPEC\_MJRY(max\_rdy)$$

$$SPEC\_MJRY(0) = maj\_red\_yel\_to\_grn \rightarrow SPEC\_MJG(max\_grn)$$
$$SPEC\_MJRY(n) = tock \rightarrow SPEC\_MJRY(n-1)$$

$$SPEC\_MJG(0) = maj\_grn\_to\_yel \rightarrow SPEC\_MJE(max\_end)$$
$$SPEC\_MJG(n) \ = tock \rightarrow SPEC\_MJG(n-1)$$

$$SPEC\_MJE(0) = maj\_yel\_to\_both\_red \rightarrow SPEC\_SWAP$$
$$SPEC\_MJE(n) = tock \rightarrow SPEC\_MJE(n-1)$$

$$SPEC\_SWAP = both\_red\_to\_min\_red\_yel \rightarrow SPEC\_MNRY(max\_rdy)$$

$$SPEC\_MNRY(0) = min\_red\_yel\_to\_grn \rightarrow SPEC\_MNG(max\_grn)$$
$$SPEC\_MNRY(n) = tock \rightarrow SPEC\_MNRY(n-1)$$

$$SPEC\_MNG(0) = min\_grn\_to\_yel \rightarrow SPEC\_MNE(max\_end)$$
$$SPEC\_MNG(n) \ = tock \rightarrow SPEC\_MNG(n-1)$$

$$SPEC\_MNE(0) = min\_yel\_to\_both\_red \rightarrow SPEC\_RR$$
$$SPEC\_MNE(n) = tock \rightarrow SPEC\_MNE(n-1)$$

The constants *max_rdy*, *max_grn* and *max_end* represent the delays in the sequence of traffic light signals, and *tock* represents the passage of time [Sch99].

3 Formalising VHDL for Model Checking

By specifying desirable properties in a formal language and translating VHDL code to the same language, we are able to check its behaviour with respect to the properties. Therefore, we would like to translate the VHDL code to CSP so that it can be checked against the specification defined above. To achieve this, we begin by translating the VHDL code to CSP ‖ B. Then we can use traditional CSP ‖ B 'lifting' techniques to incorporate the B components into CSP. It should be noted that this intermediate representation is presented here only to justify the resulting CSP process definitions. It is not integral to the approach, and would be unnecessary in an automation of the translation.

3.1 Translating VHDL into CSP ‖ B

In addition to translating the user-defined VHDL processes, we need to model VHDL's simulation semantics. There is a natural correspondence between VHDL and B. However, to model signals using B variables it is necessary to model each VHDL signal as two B variables: one representing the current signal value, and one representing the next signal value. The B operations that model signal assignments do so by assigning values to the next state variables, but only via expressions that use current state variables. Hence the current state variables need to be

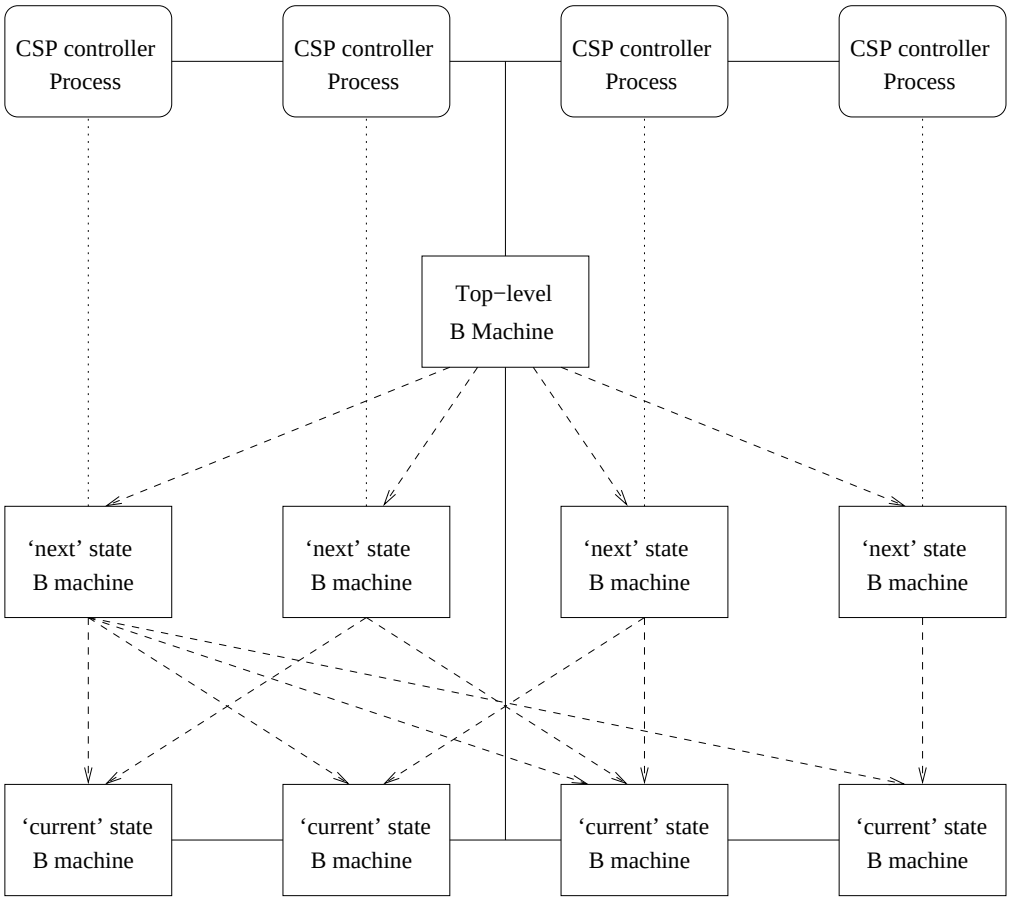

Figure 1: CSP || B architecture

seen by the next state B machines. An example of this structuring is shown in the two rows of B machines at the bottom of Figure 1.

CSP is used to model VHDL's signal update phase. CSP processes determine when all active B operations have completed (i.e., the B operations modelling the active VHDL processes in any given process execution phase). The CSP processes then synchronise to initiate the update of the current state variables with the values held in the next state variables. This requires an additional top-level B machine with write access to the current state B machines, and read access to all of the next state B machines. This structure is shown in Figure 1.

The Top-level B machine is connected to the 'current' state B machines with a solid line to indicate write access. It is connected to the 'next' state B machines with dashed arrows to indicate read access. The dotted lines from the CSP controller processes to the 'next' state B machines indicate their control over the operations within those machines. The solid line from the CSP processes to the Top-level B machine indicates that the CSP controllers synchronise to cause the update of the 'current' state B variables.

3.2 Translating the Traffic Controller

The mapping from user-defined VHDL processes to B operations is straightforward. The operation corresponding to the VHDL process `fsm` is as follows (only one case is shown to save space, but the others are very similar):

fsm $\widehat{=}$ **CASE** *s.state* **OF**
EITHER *all_red* **THEN**
red_maj := 1 || *yel_maj* := 0 || *grn_maj* := 0 ||
red_min := 1 || *yel_min* := 0 || *grn_min* := 0 ||
next_state := *major_ready*
OR *major_ready* **THEN**
⋮

Note, we have prefixed the current state variable *state* with a unique identifier (*s*) to distinguish it from the other (next state) instance of the same variable. The translations of the other VHDL processes follow a similar pattern with one or two subtleties. Consider the VHDL process `yellow_counter`:

yellow_counter $\widehat{=}$ **IF** ($clock\_event = TRUE \wedge clock = 1$) **THEN**
IF ($reset = 1$) **THEN**
$yel\_count := 0$
ELSE
IF ($yel\_maj = 1 \vee yel\_min = 1$) **THEN**
$yel\_count := y.yel\_count + 1$
ELSE
$yel\_count := 0$
END
END
END

Rather than using a bit vector, the variable *yel_count* is declared to be a natural number in the range 0 to 7 in the B model. In order to prove that **yellow_counter** maintains this typing invariant, it is necessary to add a precondition to say that the addition can happen only when *y.yel_count* is less than 7. Otherwise the operation could increment *yel_count* outside its range.[1] The precondition is generated automatically by taking the conjunction of the conditions that lead to the increment of *yel_count*.

yellow_counter $\widehat{=}$
PRE
($clock\_event = TRUE \wedge clock = 1 \wedge$
$reset = 0 \wedge (yel\_maj = 1 \vee yel\_min = 1)) \Rightarrow y.yel\_count < 7$
THEN
⋮

[1] Alternatively, we could use the *mod* operator to model 'wrap around' behaviour.

It is necessary to add a precondition to the B translation of `green_counter` for similar reasons but, in this case, *grn_count*'s range is 0 to 127. The two one-line VHDL processes that update `ready_delay` and `end_delay` are translated to (equally succinct) B operations. However, we are obliged to give them names:

set_ready_delay $\widehat{=}$ *ready_delay* := *bit* (*y.yel_count*, 1);

set_end_delay $\widehat{=}$ *end_delay* := *bit* (*y.yel_count*, 2)

The function *bit*(*x*, *y*), which returns the *y*th bit of value *x*, is not part of the B language. Hence it is necessary to define it within the B model.

In order to update the current state variables with their next state values, it is necessary to define operations within the current state B machines. These have a uniform structure in which the next state values are passed as input parameters, and the body of the operations are simple assignments of these values to the current state variables. Also, a list of Boolean values are output from the operation to indicate when the values of current state variables have changed. This information can be used to model the sensitivity lists so that the appropriate B operations are called in the next process execution phase. Consider the operation **update_fsm**:

dymj, *dgmj*, *dymn*, *dgmn* ⟵ **update_fsm** (*rj*, *yj*, *gj*, *rn*, *yn*, *gn*, *ns*) $\widehat{=}$
PRE
rj ∈ *Std_Logic* ∧ *yj* ∈ *Std_Logic* ∧ *gj* ∈ *Std_Logic* ∧
rn ∈ *Std_Logic* ∧ *yn* ∈ *Std_Logic* ∧ *gn* ∈ *Std_Logic* ∧ *ns* ∈ *State*
THEN
red_maj := *rj* || *yel_maj* := *yj* || *grn_maj* := *gj* ||
red_min := *rn* || *yel_min* := *yn* || *grn_min* := *gn* || *next_state* := *ns* ||
dymj := *bool* (*yel_maj* ≠ *yj*) || *dgmj* := *bool* (*grn_maj* ≠ *gj*) ||
dymn := *bool* (*yel_min* ≠ *yn*) || *dgmn* := *bool* (*grn_min* ≠ *gn*)
END

where *Std_Logic* is a B representation of VHDL's `Std_Logic` type and *State* is the B representation of the controller state. Note, there are no outputs concerning changes to the red signals because nothing is sensitive to such changes.

The Top-level B machine contains a single B operation called **dd** which calls all the current state update operations with the necessary input parameters. It also accumulates all of the Boolean outputs from these individual operations, and uses them as its own output. This single operation is capable of updating all current state variables simultaneously and provides the necessary information to say which variables have changed. The (abbreviated) definition of the top level B machine is shown below. Note, the operation **dd** calls **update_fsm** with the actual values of the next state variables in *NEXT_FSM*. (The other operation calls, input parameters, and output parameters are not shown.)

MACHINE *Top_Level*
SEES *n.NEXT_FSM*, ...
INCLUDES *CURR_FSM*, ...
OPERATIONS

$dymj, dgmj, dymn, dgmn, \ldots \longleftarrow$ **dd** $\widehat{=}$
BEGIN
$dymj, dgmj, dymn, dgmn \longleftarrow$ **update_fsm** $(n.red\_maj, \ldots) \parallel \ldots$

From Figure 1, we see that it is necessary to define CSP processes that control the calling of the various B operations. The idea is to mimic the simulation cycle within the CSP processes. Hence, each controller process begins by calling its associated next state B operation. On completion, the process is ready to call the Top-level B operation *dd* to update the current state B variables. However, it must wait until all processes are prepared to call *dd*. When *dd* occurs, each process can view the outputs from the *dd* operation to determine which variables' values have changed and, hence, whether it needs to call its associated next state B operation. If it does not then it waits for the next *dd* operation call. Consider the CSP process *FSM* (other controller processes are similar):

$FSM = fsm \rightarrow FSM'$

$FSM' = dd?bb_1?...?bb_n \rightarrow if\ (bb_{i1}\ or\ bb_{i2}\ or\ \cdots\ or\ bb_{im})\ then\ FSM\ else\ FSM'$

3.3 'Lifting' the B Components into CSP

In order to use FDR, *lifting* is the way important state information is taken from B into CSP. This entails, among other things, parameterising the CSP processes. The effects of the B operations are then modelled as updates to the parameters.

The lifted B operation **fsm** comprises a choice of guarded processes to represent the case statement(one process for each case):

$$
\begin{array}{l}
FSM(st, rdly, gdly, edly, rmj, ymj, gmj, rmn, ymn, gmn, nst) = \\
\quad (st = all\_red)\ \& \\
\qquad \text{let } rmj' = 1 \\
\qquad\quad ymj' = 0 \\
\qquad\quad gmj' = 0 \\
\qquad\quad rmn' = 1 \\
\qquad\quad ymn' = 0 \\
\qquad\quad gmn' = 0 \\
\qquad\quad nst' = major\_ready \\
\qquad \text{within } FSM'(st, rdly, gdly, edly, rmj', ymj', gmj', rmn', ymn', gmn', nst') \\
\quad \Box\ (st = major\_ready)\ \& \\
\qquad \text{let } ymj' = 1 \\
\qquad\quad nst' = if\ (rdly = 1)\ then\ major\_go\ else\ nst \\
\qquad \text{within } FSM'(st, rdly, gdly, edly, rmj, ymj', gmj, rmn, ymn, gmn, nst') \\
\quad \Box\ \ldots
\end{array}
$$

Note that the call to FSM' in this definition is parameterised with signal values. This is necessary to keep track of the values as they are distributed among the processes. In order to model the sensitivity list, a conditional statement is introduced into FSM' to check if the new values arriving

on *dd* differ from the values held by the process. If so then *FSM* is called, otherwise control is passed back to FSM'. Note, in the definition of FSM' below we show the relevant arguments of *dd* only. In this case, the values of `state`, `ready_delay`, `grn_delay` and `end_delay` are relevant because they appear in `fsm`'s sensitivity list.

$$
\begin{array}{l}
FSM'(st, rmj, ymj, gmj, rmn, ymn, gmn, nst, rdly, gdly, edly) = \\
\quad dd \ldots ?st' \ldots ?rdly' ?gdly' ?edly' \rightarrow \\
\quad\quad \mathit{if}\ (st' \neq st\ \mathit{or}\ rdly' \neq rdly\ \mathit{or}\ gdly' \neq gdly\ \mathit{or}\ edly' \neq edly)\ \mathit{then} \\
\quad\quad\quad FSM(st', rdly', gdly', edly', rmj, ymj, gmj, rmn, ymn, gmn, nst) \\
\quad\quad \mathit{else}\ FSM'(st, rdly, gdly, edly, rmj, ymj, gmj, rmn, ymn, gmn, nst)
\end{array}
$$

The VHDL process `yellow_counter` is sensitive to changes in the clock signal. Like most processes that are sensitive to the clock signal, the conditional statement within the process checks for a clock edge (in this case a rising edge). Consequently, such conditional statements will never evaluate to true unless they have been preceded by a signal update phase that changed the clock signal. In our formal model, the signal update phase is modelled by the CSP event *dd*. Hence, for *YELLOW_COUNTER* (and the formalisation of other clock-sensitive processes) we begin with the *dd* event.

The definitions of **yellow_counter** and **green_counter** in the B model included preconditions that, when evaluated to false, result in unpredictable (divergent) behaviour. It is necessary, therefore, to include these preconditions as diverging assertions in CSP in order to 'lift' the B operations completely.

$$
\begin{array}{l}
YELLOW\_COUNTER(yel\_count) = \\
\quad dd \ldots ?yel\_maj ?yel\_min ? \ldots \left\{ \begin{array}{l} clock\_event = true \wedge clock = 1 \wedge \\ reset = 0 \wedge (yel\_maj = 1 \vee yel\_min = 1) \\ \quad \Rightarrow yel\_count < 7 \end{array} \right\} \rightarrow \\
\quad\quad \mathit{if}\ (clock\_event)\ \mathit{then} \\
\quad\quad\quad YELLOW\_COUNTER'(clock, reset, yel\_maj, yel\_min, yel\_count) \\
\quad\quad \mathit{else}\ YELLOW\_COUNTER(yel\_count)
\end{array}
$$

$$
\begin{array}{l}
YELLOW\_COUNTER'(clock, reset, yel\_maj, yel\_min, yel\_count) = \\
\quad \text{let} \\
\quad\quad yel\_count' = \mathit{if}\ (clock = 1)\ \mathit{then} \\
\quad\quad\quad\quad\quad\quad \mathit{if}\ (reset = 1)\ \mathit{then}\ 0 \\
\quad\quad\quad\quad\quad\quad \mathit{else\ if}\ (yel\_maj = 1 \vee yel\_min = 1)\ \mathit{then}\ yel\_count + 1 \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{else}\ 0 \\
\quad\quad\quad\quad\quad\quad \mathit{else}\ yel\_count \\
\quad \text{within} \\
\quad\quad YELLOW\_COUNTER(yel\_count')
\end{array}
$$

The lifting technique is applied to the other B operations in a similar way.

This completes the lifting procedure. Unfortunately this is not in a form that can be accepted by FDR. In particular, each process contributes relatively few arguments to the event *dd*, and the eager manner in which FDR builds the transition graph means that the FDR compiler is overwhelmed by enumerating all possible instances of *dd* prior to parallel composition. This effort is

largely unnecessary because the parallel composition is such that very few of the instances will be exercised during the analysis. Hence, optimisation steps are needed.

3.4 Optimising the CSP for Model Checking

We begin by merging each unprimed/primed process pair into single process definitions. This results in processes with a uniform structure that will make optimisation easier. Consider the lifted CSP process *NEW_STATE*:

$$
\begin{array}{l}
NEW\_STATE(state) = \\
\quad dd \ldots ?clock\_event?clock?reset!state?next\_state \ldots \rightarrow \\
\quad\quad if\ (clock\_event)\ then\ NEW\_STATE'(clock, reset, state, next\_state) \\
\quad\quad else\ NEW\_STATE(state)
\end{array}
$$

We can remove the conditional statement by pattern matching on *clock_event*:

$$
\begin{array}{l}
NEW\_STATE(state) = \\
\quad\quad dd \ldots !true?clock?reset!state?next\_state \ldots \rightarrow NEW\_STATE'(clock, \ldots) \\
\quad \Box\ dd \ldots !false?clock?reset!state?next\_state \ldots \rightarrow NEW\_STATE(state)
\end{array}
$$

We then incorporate the process *NEW_STATE*′ by updating the parameter *state* in an appropriate way. This results in an external choice of *dd* events:

$$
\begin{array}{l}
NEW\_STATE(state) = \\
\quad\quad dd \ldots !true!1!1!state?next\_state \ldots \rightarrow NEW\_STATE(both\_red) \\
\quad \Box\ dd \ldots !true!1!0!state?next\_state \ldots \rightarrow NEW\_STATE(next\_state) \\
\quad \Box\ dd \ldots !true!0?\_!state?next\_state \ldots \rightarrow NEW\_STATE(state) \\
\quad \Box\ dd \ldots !false?\_?\_!state?next\_state \ldots \rightarrow NEW\_STATE(state)
\end{array}
$$

The first *dd* event handles the case when the clock is 1 and the reset is 1, and results in the *both_red* state. The second *dd* event handles the case when the clock is 1 and the reset is 0. This replaces the current state with the value bound to the *next_state* argument of *dd*. The remaining cases keep the current state value.

If we transform *YELLOW_COUNTER* and *GREEN_COUNTER* in a similar manner, we have to somehow incorporate the diverging assertions into the modified process definition. This is done by pattern matching on the processes' parameters. First recall the lifted definition of *YELLOW_COUNTER*:

$$
\begin{array}{l}
YELLOW\_COUNTER(yel\_count) = \\
\quad dd \ldots ?yel\_maj?yel\_min \ldots \left\{ \begin{array}{l} clock\_event = true \wedge clock = 1 \wedge \\ reset = 0 \wedge (yel\_maj = 1 \vee yel\_min = 1) \\ \quad \Rightarrow yel\_count < 7 \end{array} \right\} \rightarrow \\
\quad\quad \vdots
\end{array}
$$

When the parameter *yel_count* is not 7 the diverging assertion is true, and can be ignored. However, when it is 7 then the antecedent of the implication in the diverging assertion must be shown to be false (i.e., we negate the antecedent and use it as a new diverging assertion):

$$YELLOW\_COUNTER(7) =$$
$$dd \ldots ?yel\_maj?yel\_min \ldots \left\{ \begin{array}{l} clock\_event = false \vee clock = 0 \vee \\ reset = 1 \vee (yel\_maj = 0 \wedge yel\_min = 0) \end{array} \right\} \rightarrow$$
$$\vdots$$

Once again, we incorporate $YELLOW\_COUNTER'$ to produce a process comprising an external choice of unique dd events, but for each case we check whether the diverging assertion is true or false and define the process's subsequent behaviour accordingly:

$$\begin{array}{l} YELLOW\_COUNTER(7) = \\ \quad dd \ldots !true!1!1 \ldots ?\_?\_ \ldots \rightarrow YELLOW\_COUNTER(0) \\ \Box\ dd \ldots !true!1!0 \ldots !0!0 \ldots \rightarrow YELLOW\_COUNTER(0) \\ \Box\ dd \ldots !true!1!0 \ldots !1!0 \ldots \rightarrow DIV \\ \Box\ dd \ldots !true!1!0 \ldots !0!1 \ldots \rightarrow DIV \\ \Box\ dd \ldots !true!1!0 \ldots !1!1 \ldots \rightarrow DIV \\ \Box\ dd \ldots !true!0?\_ \ldots ?\_?\_ \ldots \rightarrow YELLOW\_COUNTER(7) \\ \Box\ dd \ldots !false?\_?\_ \ldots ?\_?\_ \ldots \rightarrow YELLOW\_COUNTER(7) \end{array}$$

The process DIV diverges immediately. Each branch that ends with this process begins with an instance of dd whose arguments falsify the diverging assertion. There are three cases that cause divergent behaviour, that would otherwise increment yel_count out of bounds. The instance of $YELLOW\_COUNTER$ when yel_count is not 7 is identical except the calls to DIV are replaced by calls to $YELLOW\_COUNTER(yel\_count + 1)$ (i.e., there is no divergent behaviour). There are corresponding definitions for $GREEN\_COUNTER$.

The (abbreviated) definition of FSM shown below is the result of incorporating FSM' into the 'lifted' definition of FSM. By pattern matching on the current state argument of dd we once again get a process that is an external choice of unique instances of dd, each of which is followed by a conditional statement to determine the next course of action. Note that each branch has the same structure as the body of FSM'.

$$\begin{array}{l} FSM(st, rmj, ymj, gmj, rmn, ymn, gmn, ns, rdly, gdly, edly) = \\ \quad dd \ldots !both\_red!rmj!ymj!gmj!rmn!ymn!gmn!ns?rdly'?gdly'?edly' \rightarrow \\ \quad\quad (if\ (st \neq both\_red \vee rdly \neq rdly' \vee gdly \neq gdly' \vee edly \neq edly')\ then \\ \quad\quad\quad FSM(both\_red, 1, 0, 0, 1, 0, 0, major\_ready, rdly', gdly', edly') \\ \quad\quad else\ FSM(st, rmj, ymj, gmj, rmn, ymn, gmn, ns, rdly, gdly, edly)) \\ \Box\ dd \ldots !major\_ready!rmj!ymj!gmj!rmn!ymn!gmn!ns!0?gdly'?edly' \rightarrow \\ \quad (if\ (st \neq major\_ready \vee rdly \neq 0 \vee gdly \neq gdly' \vee edly \neq edly')\ then \\ \quad\quad FSM(major\_ready, rmj, 1, gmj, rmn, ymn, gmn, ns, 0, gdly', edly') \\ \quad else\ FSM(st, rmj, ymj, gmj, rmn, ymn, gmn, ns, rdly, gdly, edly)) \\ \Box \ldots \end{array}$$

The delay processes (not shown) are transformed in a similar manner.

Now we are in a position to optimise the parallel composition of the above processes. The solution adopted in this paper is to merge process definitions to reduce the number of processes synchronising on dd and increase the number of arguments that individual processes contribute.

We illustrate this by merging the *NEW_STATE* process with the counter processes and delay processes because, individually, these process contribute one argument each to *dd*; collectively, they contribute six arguments. When merging processes, the divergent process *DIV* takes precedence: if any single process diverges in a given state then the entire process diverges. Merging preserves the structure of the processes derived in the previous step (i.e., an external choice of *dd* events), so merging can be repeated as often as necessary. In this instance, we shall call the resulting process *NS_YC_GC_DLY*. In its most general form, the merged process is:

$$
\begin{array}{l}
NS\_YC\_GC\_DLY(0, st, yc, gc, rdly, gdly, edly) = \\
\quad dd!true!1!1!st?\_?\_?\_?\_?\_?\_?\_!rdly!gdly!edly \rightarrow \\
\qquad NS\_YC\_GC\_DLY(1, both\_red, 0, 0, bit(yc,1), bit(gc,6), bit(yc,2)) \\
\quad \Box\ dd!true!1!0!st?\_!1!1?\_?\_?\_?ns!rdly!gdly!edly \rightarrow \\
\quad NS\_YC\_GC\_DLY(1, ns, yc+1, gc+1, bit(yc,1), bit(gc,6), bit(yc,2)) \\
\quad \Box\ dd!true!1!0!st?\_!0!1?\_!0?\_?ns!rdly!gdly!edly \rightarrow \\
\quad NS\_YC\_GC\_DLY(1, ns, 0, gc+1, bit(yc,1), bit(gc,6), bit(yc,2)) \\
\quad \Box\ dd!true!1!0!st?\_!1!0?\_?\_!0?ns!rdly!gdly!edly \rightarrow \\
\quad NS\_YC\_GC\_DLY(1, ns, yc+1, 0, bit(yc,1), bit(gc,6), bit(yc,2)) \\
\quad \Box \ldots
\end{array}
$$

Note that, due to the merging of the counter processes and the delay processes, the count parameters (*yc* and *gc*) are no longer needed outside this process. Hence, they are removed from the arguments of *dd*. There are three other (pattern matching) instances of this process that exhibit some divergent behaviour.

4 Model Checking

Our controller can now defined as:

$$
\begin{array}{rl}
CONTROLLER & = FSM(both\_red, 1, 0, 0, 1, 0, 0, major\_ready, 0, 0, 0) \\
& \quad [|\{|\ dd\ |\}|]\, NS\_YC\_GC\_DLY(0, both\_red, 0, 0, 0, 0, 0)
\end{array}
$$

The values of the parameters are the same values that would have been assigned by the next state B operations. Hence, the initial state of *CONTROLLER* corresponds to the VHDL prior to its first signal update phase.

We rename *CONTROLLER* so that we can use FDR to check that it refines *SPEC_RR*. For example, *dd*!*true*!1!0!*both_red*!1!0!0!1!0!0!*major_ready*?_ is renamed as *both_red_to_maj_red_yel*. If *REN* is the entire renaming then by hiding the remaining *dd* events we can perform a refinement check in FDR:

$$assert\ SPEC\_RR \sqsubseteq_T CONTROLLER[\![REN]\!] \setminus \{|\ dd\ |\}$$

which succeeds. It also succeeds if we perform the same check in the failures model. This shows that the controller is as 'live' as the specification that it implements. Another important check is to determine whether it is possible for the controller to cause a divergence by reaching a *DIV* state. This can be achieved by performing a livelock-free check on *CONTROLLER*. Note

that we are not hiding any events, so if the check fails then it is by virtue of a *DIV* process. The livelock-free check succeeds, and we conclude that the controller does not attempt to increment one of its counters out of bounds.

5 Conclusion

The challenge that prompted this work was to combine formal and informal approaches to produce verified hardware. The proposed solution allows the hardware developers to continue to use their informal approaches, but complements this with a method for translating VHDL to formal notations for analysis. Of all the related work, perhaps the most relevant is [ZT05] (in combination with [BGR00]) which translates B specifications and existing VHDL to ACL2 for analysis. However, in our case, the project requirements influenced the choice of formal approach considerably. CSP was used because it is a good language to formalise requirements of this form. The B Method was less suitable in this respect because, at its most abstract level, control flow is implicit.

The language of the B Method is, however, very similar to VHDL, and translating from VHDL to B is quite straightforward. CSP can also be used to provide the necessary control, which motivates a CSP $\|$ B approach. The ProB [LB03] tool is available to do trace refinement checks, but other kinds of checks (such as those described in Section 4) are not available. The approach presented here uses the existing CSP $\|$ B techniques to 'lift' the B components into CSP in order to use FDR. Another piece of work using CSP in a VHDL context is [CH97], but this focuses on synthesis.

The problem with using FDR lies in its eager approach to building transition graphs. An optimisation technique has been proposed in this paper to circumvent this problem. In this respect, ProB would be better because it uses a lazy approach. Scalability is always a key issue with formal methods. The optimisation can be repeated as necessary, so it works for larger specifications with more processes. Automation of the approach is ongoing work.

A lot of work has been done on the formal semantics of VHDL (including [FM95]). This work provides useful insights into VHDL itself, but cannot be deployed in an industrial context directly.

Acknowledgements: The author is very grateful to the anonymous referees and workshop participants for their useful comments.

Bibliography

[Abr96] J. R. Abrial. *The B Book: Assigning Programs to Meaning*. CUP, 1996.

[BGR00] D. Borrione, P. Georgelin, V. Rodrigues. Using Macros to Mimic VHDL. In Kaufmann et al. (eds.), *Computer-Aided Reasoning ACL2 Case Studies*. Kluwer Academic Publishers, 2000.

[CH97] R. Chapman, D. H. Hwang. A Formally Verified High-Level Synthesis Front-end: Translation of VHDL to Dependence Flow Graphs. In *EuroDAC '97*. ACM, 1997.

[FM95] M. Fuchs, M. Mendler. A Functional Semantics for Delta-Delay VHDL Based on Focus. In Kloos and Breuer (eds.), *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.

[For] Formal Systems (Europe) Ltd. Failures-Divergences Refinement: FDR2 Manual.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[IEE02] IEEE Comp. Society. IEEE Standards: VHDL Language Reference Manual. 2002.

[LB03] M. Leuschel, M. Butler. ProB: A Model Checker for B. In Araki et al. (eds.), *FM'2003: Formal Methods*. LNCS 2805. Springer, 2003.

[Sch99] S. Schneider. *Concurrent and Real-Time Systems: the CSP Approach*. Wiley, 1999.

[ST02] S. Schneider, H. Treharne. Communicating B machines. In *ZB2002*. LNCS 2272. Springer, 2002.

[ST05] S. Schneider, H. Treharne. CSP Theorems for Communicating B Machines. *Formal Aspects of Computing* 17:390–422, 2005.

[ZT05] Y. Zimmermann, D. Toma. Component Reuse in B Using ACL2. In *ZB2005*. LNCS 3455. Springer, 2005.