Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the
Ninth International Workshop on
Automated Verification of Critical Systems
(AVOCS 2009)

Towards SMV Model Checking of SIGNAL (multi-clocked) Specifications

Julio C. Peralta and Thierry Gautier

15 pages

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

Towards SMV Model Checking of SIGNAL (multi-clocked) Specifications

Julio C. Peralta and Thierry Gautier

IRISA-INRIA, Centre Rennes-Bretagne Atlantique, 35042 Rennes cedex, France
Julio.Peralta@inria.fr Thierry.Gautier@inria.fr

Abstract: SIGNAL is a high-level data-flow specification language that equally allows multi-clocked descriptions as well as single-clocked ones. It has a formal semantics and is supported by several formal tools for simulation and static validation. This generality renders it useful for various specification, simulation, and verification tasks in embedded system design. SMV, in turn, is a language and model checker where synchronous models are single-clocked by definition. Roughly, we use standard techniques to describe clocks by Boolean variables, with the advantage that the number of such variables is kept to a minimum through a static analysis provided by the SIGNAL compiler. In particular, we propose a translation from possibly multi-clocked SIGNAL specifications into SMV specifications for their corresponding verification by model checking.

Keywords: Synchronous programs, Multiple-clocks, SMV, Model checking

1 Introduction

The increasing complexity of embedded systems and the costs associated with failures in their engineering and operation demand for models and tools that enable safe design and formal validation. In the past years, system design based on the *synchronous model* [BB91] has attracted the attention of many academic and industrial actors. This paradigm consists in abstracting the non-functional implementation details of a system, thus fostering a focused reasoning on the logic behind the instants at which the system functionalities should be secured. A benefit of designing with languages based on the synchronous model (e.g. ESTEREL [BG92], LUSTRE [HCRP91], or SIGNAL [LTL03]) is the availability of associated verification tools.

Among synchronous languages, a salient feature of SIGNAL is the notion of *polychrony*: the capability to describe systems in which components may have different clock rates. This expressivity coupled with its (compiler) ability to statically synthesize schedules (reasoning about the logic behind the source clock constraints) allows to embrace complex systems that arise in the form of GALS (globally-asynchronous locally-synchronous) or (loosely) time-triggered architectures, and thus renders the model checking of such specifications highly attractive.

SMV, in turn, is a language and model checker where synchronous models are single-clocked by definition. However, this apparent constraint does not prevent us from describing and verifying SIGNAL multi-clocked specifications as we demonstrate here. In order to describe multi-clocked computations using a single-clocked framework we use standard techniques to describe clocks by Boolean variables [BBG+00], with the advantage that the number of such variables is kept to a minimum through a static analysis provided by the SIGNAL compiler. Such analysis produces a hierarchy of clocks (ordered by set inclusion) which is useful to avoid proliferation of SMV state variables.

The paper presents in Section 2 syntactic and semantic highlights of our source SIGNAL programs and the target SMV programs. Section 3, in turn, describes a generic SMV translation for each SIGNAL kernel operator. Then in Section 4 we provide examples of translations for (possibly multi-clocked) SIGNAL specifications and show the use of the SIGNAL compiler analysis to reduce the number of SMV state variables. The behaviours of the translated examples will be examined in Section 5 by model checking with SMV itself. Next, in Section 6 some elements for comparison with related work on model checking for other synchronous languages are presented. Finally, some concluding remarks and pointers for future work are given in Section 7.

2 SIGNAL and SMV: Syntax and semantics

In this section we introduce the SIGNAL kernel language and a subset of the SMV language used for our translation, as well as highlights of each language semantics.

2.1 SIGNAL kernel

SIGNAL is a data-flow relational language that relies on the polychronous model [LTL03, BGL08]. It handles possibly infinite sequences of typed values called *signals*. A signal `x` is implicitly indexed by discrete time, thus denoting the sequence x_t where $t \in H$, $H \subseteq \mathbb{N}$. At any instant (arbitrary $t \in \mathbb{N}$) a signal may be *present*, at which point it holds a value, or *absent*. There is no actual value associated with a signal when it is absent, by contrast with the instants when it is present. The instants of absence of a signal are denoted with the special symbol $\perp$, in the semantics. Signals may be of standard types, e.g. Boolean, integer, real, etc. Additionally, there is a particular type of signal called `event`. A signal of this type is always *true* when it is present. The set of instants (index set H above) where a signal `x` is present represents its *clock*, noted $\hat{x}$ (which implicitly denotes a signal of `event` type). A *process* is a system of equations (also called elementary processes) over signals that specifies relations between values and clocks of the signals. A *program* is a process. SIGNAL relies on a few primitive constructs that define *elementary processes* from which bigger processes may be built. Next the definition of four elementary processes and two other constructs to build bigger processes, and to mask signals, respectively.

- *Function.* `y:= f(x1,...,xn)` $\overset{def}{\equiv} x1_t \neq \perp \Leftrightarrow \ldots \Leftrightarrow xn_t \neq \perp \Leftrightarrow \mathrm{y}_t = f(x1_t, \ldots, xn_t)$
- *Delay.* `y:= x $ 1 init c` $\overset{def}{\equiv} x_t \neq \perp \Leftrightarrow y_t \neq \perp \Leftrightarrow [(t > 0 \wedge y_t = x_k \wedge k = \max\{t' \mid t' < t \wedge x_{t'} \neq \perp\}) \vee (t = 0 \wedge y_t = \mathrm{c})]$; (`c` is a compile time constant).
- *Undersampling.* `y:= x when b` (where `b` is Boolean) $\overset{def}{\equiv} y_t = x_t$ if $b_t = true$, else $y_t = \perp$; (observe that expression `y:= when b` is equivalent to `y:= b when b`).
- *Deterministic merge.* `z:= x default y` $\overset{def}{\equiv} z_t = x_t$ if $x_t \neq \perp$, else $z_t = y_t$.

Table 1: Clock relations for primitives.

construct	clock relations
`y := f(x1,...,xn)`	$\hat{y} = \hat{x}_1 = \ldots = \hat{x}_n$
`y := x $1 init c`	$\hat{y} = \hat{x}$
`y := x when b`	$\hat{y} = \hat{x} \cap [b]$, $[b] \cup [\neg b] = \hat{b}$ and $[b] \cap [\neg b] = \emptyset$
`z := x default y`	$\hat{z} = \hat{x} \cup \hat{y}$

- *Parallel Composition.* `P1|P2` $\overset{def}{\equiv}$ union of equations of `P1` and `P2`.

- *Hiding.* `P where x` $\overset{def}{\equiv}$ `x` is local to the process `P`.

Derived operators are defined using the primitive operators above. For instance, a *synchronization* equation `x ^= y` specifies that `x` and `y` have the same clock. Moreover, the equation `x ^= y ^+ z` asserts that the clock of `x` is the union of the clocks of `y` and that of `z`. A *memory*: `y := x cell b init y0` allows to memorize in `y` the latest value carried by `x` when `x` is present or when `b` is *true*. Processes can be abstracted and declared, in a standard way, by explicitly designating their input and output signals (preceding their declarations with "`?`" and "`!`", respectively), with the sole constraint that the designated input signals cannot be defined (i.e. occur in the lhs of a `:=` symbol) inside such a process.

2.2 Static analysis of SIGNAL specifications

In order to assess the consistency of the clock relations associated with a program, and to organize the control of such a program, the compiler synthesizes a *clock hierarchy* [ABL95, BGL08]. A clock k_1 is said to be greater than a clock k_2 if k_2 is included in k_1 in terms of sets of instants.

Table 1 shows the *clock relations* implicit in each primitive construct of SIGNAL. For the undersampling construct, the clock of the Boolean signal `b` is partitioned into $[b]$ and $[\neg b]$. The sub-clock $[b]$ (resp. $[\neg b]$) denotes the set of instants where the Boolean expression `b` is present and *true* (resp. *false*). Clock relations are automatically added and (possibly) new relations between clocks are inferred by the compiler from any program to be analyzed. For a program $\texttt{P = P}_1\texttt{|...|P}_n$, its resulting relations between clocks are the result of applying the clock calculus on the conjunction of the clock relations associated with the sub-processes $\texttt{P}_k$, $k \in 1..n$.

The *clock calculus* [ABL94], in turn, seeks the greatest clock in the program, called *master clock*, from which all other clocks in the program can be extracted. In this case, the clock hierarchy is a tree. Nonetheless, in some programs, such a unique master clock may not exist. In this latter case, there are several local master clocks and the clock hierarchy is a forest. Note, however, that the root of a tree may not correspond with the clock of an input signal.

A program in which the clock hierarchy is a tree is *endochronous*. Such a program can be run in an autonomous way (its master clock plays the role of an activation clock). Otherwise, the program needs extra information from its environment to be run in a deterministic way.

The automatic code generation, for an endochronous program, relies on the synthesized clock

SMVpgr	→	*ModuleMain* | *ModuleStmt SMVpgr*	*AssignStmt*	→	ε | *lhs* := *rhs* ; *AssignStmt*
ModuleStmt	→	`MODULE id[(`*IdList*`)]`	*InvarStmt*	→	ε | *s_bool_exp*
		`VAR` *VarDclLst*	*lhs*	→	`init(id)` | `id` | `next(id)`
		`ASSIGN` *AssignStmt*	*rhs*	→	*cnst_exp* | *set_exp* | *case_exp*
		`INVAR` *InvarStmt*	*case_exp*	→	`case` *cases* `esac` ;
VarDclLst	→	ε | `id` : *Type* ; *VarDclLst*	*cases*	→	`1` : *rhs* ; | *bool_exp* : *rhs* ; *cases*

Figure 1: Subset of SMV language.

hierarchy. Each clock is represented by a Boolean variable (booleanization stage [BBG+00]) which is true when the clock is present, and false otherwise. For every signal, its value is meaningful (under a multi-clock interpretation) when the Boolean representing its clock has the value true. This allows to organize the control of the application following the clock hierarchy.

2.3 SMV: A subset

For our translation purposes we use only a subset of the SMV language. We present such a subset using the syntax of SMV that is compatible with the three versions [McM01, McM99, CCJ+05] of the language currently available on the web (SMV from CADENCE, SMV from CARNEGIE-MELLON UNIVERSITY, and NUSMV). We identify the following syntax with the oldest [McM01] of the three versions.

Syntax

Our SMV programs will consist of modules with parameters, except for the reserved module `main`. Module declarations may not be nested. Each module has a name, (possibly) a list of parameter names, and at most three sections: a section for variable declarations and/or module instantiation, marked at the beginning by the `VAR` reserved word; a section describing the variable values (initial, current or next instant), initiated by the `ASSIGN` keyword; and, a section describing invariants between the variables of the referred module, and whose beginning is marked by the reserved word `INVAR`. `DEFINE`, `FAIRNESS` and `SPEC` sections are not considered for the moment, but their use will be motivated when we present translation examples (Sect. 4), and some verification (Sect. 5) on them.

Figure 1 depicts the grammar of our subset of SMV. The possible type (*Type*) of an identifier (`id`) is `integer` (or intervals thereof), `boolean`, enumerated, or the name of another module; in this last case the identifier is used to refer to an instance of the referred module, and appropriate expressions should be given as parameters for the intended instance. Access to members of a module instance is through a dot notation (i.e. `id.var_id`).

An expression of an invariant is of type Boolean and may only contain module variables in its present form (i.e. no use of `init` or `next` operators are allowed). The right-hand-side of an assignment (*rhs*), for the case of a constant expression (*cnst_exp*), is a valid expression (e.g. containing arithmetic, Boolean or comparison operators) using any of the possible type values for a correct typing of the identifier in the left-hand-side; a right-hand-side, for the case of a set expression (*set_exp*), uses curly braces to extensively list the elements (separated by commas) of the desired set. Operations on sets are union and test of membership.

```
VAR
   f, h_x, h_y:  boolean;
ASSIGN
   init(y) := C;
   next(y) := case
                f & next(h_x) :  x;
                1 :  y;
              esac;
   init(f) := h_x;
   next(f) := f | next(h_x);
INVAR
   (h_y <-> h_x)
```

(a) `y := x$1 init C` in SMV

```
VAR
   h_x, h_y, h_b:  boolean;
ASSIGN
   init(y) := x;
   next(y) := case
                next(h_y) :  next(x);
                1 :  y;
              esac;
INVAR
   (h_y <-> (h_x & h_b & b))
```

(b) `y := x when b` in SMV

Figure 2: Function and undersampling operators in SMV

Semantics

Assignments for the first value (signaled by the use of `init` keyword on the *lhs*) of a program variable are only executed in the first instant of program execution, whereas assignments for the `next` instant are executed to obtain the value of the designated variable starting from the second instant. Assignments with no occurrences of `init` or `next` in their *lhs* are executed at all instants. The order in which assignments are executed is given by the data dependencies existing between the variables occurring in the right-hand-sides of the assignments to execute (among all assignments of a program including those added by process instantiation). The rule that dictates the (partial) order of assignment execution says that a variable is first assigned before its value is used in a right-hand-side evaluation. Invariants define (possibly) extra relations/constraints to those already imposed by the assignments, thus limiting the valid executions of the source program to those where the invariant expressions hold.

3 From SIGNAL to SMV

Let us now describe a possible translation from simple equations in the SIGNAL kernel, to SMV module fragments. We will assume, for simplicity of exposition, that there is only one kernel operator per equation. Also, the translation for each such SIGNAL source equation is an SMV program fragment where variable declarations will be omitted (whenever possible) to allow for a greater translation generality, provided that their translation depends on whether they are input, output or local in the presence of multiple SIGNAL processes. Roughly, the translation has an SMV variable to carry the value of each source signal, as well as a Boolean SMV variable to denote its clock. An instant of an SMV execution corresponds to an instant of SIGNAL execution. The multi-clock reading of an SMV generated program comes from reading pairs of SMV variables: one denoting its clock and another carrying its value (if any).

Delay See SMV translation in Figure 2(a). Variables `h_x`, `h_y`, and `f` were added by the translation. The first two represent the clock of `x` and `y` respectively, while the last variable is

```
VAR
   h_x, h_y, h_z:  boolean;
ASSIGN
   init(z) := case
                h_x:  x;
                1:  y;
              esac;
   next(z) := case
                next(h_x):  next(x);
                next(h_y):  next(y);
                1:  z;
              esac;
INVAR
   (h_z <-> (h_x | h_y))
```

(a) `z := x default y` in SMV

```
VAR
   h_x, h_y, h_z:  boolean;
ASSIGN
   init(z) := f(x,y);
   next(z) := case
                next(h_z):  next(f(x,y));
                1:  z;
              esac;
INVAR
   (h_z <-> h_x) & (h_x <-> h_y)
```

(b) `z := f(x,y)` in SMV

Figure 3: Merge and function operators

used to detect the first instant of signal `x`. The guard labeled with `1` in the `case` statement is the default choice if none of the offered options holds. It is important to note here that the previous value of `x` should be kept (in its SMV definition, not shown here) in case it is absent (typically the default case in a `next` assignment) since this SIGNAL operator will refer to the previous value in SIGNAL semantics, which is not necessarily that of SMV. Also, note here that the value of `y` is kept in case its first instant does not coincide with that of SMV. Variable `f` is needed to detect the first instant of `y` (or `x` since they are synchronous).

For the SIGNAL kernel operators that follow we decided to keep the previous value of the defined variable, considering a general schema of translation, but in some particular occurrences of such operators we may not need to keep the value. The use of assignments that keep the value by default, allows for stuttering steps in our translation: a fundamental property if compositionality is desired.

Undersampling The SMV translation is depicted in Figure 2(b). Here we (potentially) need three clocks, one for each signal. The `init` definition fixes the value to that of `x` disregarding clock `h_y`. This is correct, however, because if `h_y` holds in the first instant then the value is correct, and if it doesn't then the value is not important, thus any value is valid in this last case.

Merge Figure 3(a) depicts the translation into SMV. Here, as above, we have three (clock) SMV variables. Once again, the initial assignment definition for the default case (labeled with `1`) appears arbitrary; it is justified, however, with a similar reasoning as that used for the undersampling operator above.

Function Refer to Figure 3(b) for the SMV translation. The reason for the initial instant assignment is similar to that used for the `when` operator above. Whether the output is present or not, the chosen value will be good.

3.1 Improving the translation into SMV

So far we have proposed an intuitively correct translation from SIGNAL elementary processes into SMV modules. We anticipate/conjecture that this translation is correct given the straightforward coding style of data-flow and clock constraints into SMV assign statements and invariants. Nonetheless, scalability is another desirable feature. To this aim we would like to reduce the number of SMV (state) variables introduced by our translation, since the number of such variables may (sometimes) render the state space exponentially bigger. The natural candidates for elimination are the clock variables, and perhaps also the SMV variables corresponding to signal source variables.

In order to avoid state variables in the translation the reader should know that SMV allows to define a variable as a function of other variables without the use of `next` or `init` operators. That is, such assignments may only refer to the present values of other SMV variables. In order to identify such variable definitions SMV provides a section named `DEFINE`. Roughly, uses of the variables so defined are replaced by their definition thus sparing some state variables.

At first sight, we may think that there is no need to introduce state variables for signals defined through operator `when`, or `default`, or `function`, since they all refer to values in the same instant. It would be tempting to replace them by their equivalent in the `DEFINE` section, and thus their values would be arbitrary when absent. However, this replacement would be incorrect when the values they define are referenced through a SIGNAL `delay` operator. Recall that the clock of a SIGNAL variable coincides with the instants of the associated SMV Boolean variable when it has value `true`, which is *not necessarily* the previous SMV instant. Consequently, for those SIGNAL elementary processes using kernel operator `when`, `default`, or `function` that do not define a value used in a delay operator, one may replace the SMV translation proposed above by one referring to present values in the corresponding `DEFINE` section.

For the SMV (clock) variables introduced we propose to have one state SMV variable per tree root in the forest constructed (during clock calculus) by the SIGNAL compiler. The remaining SMV (clock) variables will be assigned in the `DEFINE` section. It is important to note here a shift in the translation. So far we translated clock relations as Boolean formulas in the `INVAR` section by pure constraint reasoning. Replacing such constraints with assignments (in the `DEFINE` section) renders the constraints *functional*. In summary, SMV variables that represent source SIGNAL clocks and are associated with an internal node in one of the trees found by the SIGNAL compiler may be translated using assignments in the corresponding SMV `DEFINE` section. In addition, the number of clock variables may be reduced by using one variable per synchronous equivalence class found by the compiler, as well as by elimination of those clocks (variables) found to be empty.

4 Translation examples

In the following we will provide examples of source SIGNAL specifications and their translation into SMV. Such SIGNAL examples will make part of a bigger specification describing a communication protocol for loosely time-triggered architectures [BCL+02].

4.1 A one-place FIFO

Consider a one-place FIFO in SIGNAL, `fifo_1` in Figure 4(a). Its content is the last value written into it. The output (signal `sx`) may only be read/retrieved after at least one instant that it was entered. The number of instants between a write and a read may increase non-deterministically. Each such instant is given by the (internal) clock of the local Boolean signal `b` (`interleave` process). Before translating into SMV we will give the `fifo_1` program to the SIGNAL compiler so that the hierarchy of clocks becomes evident as well as other optimisations applied by the compiler. For this program the compiler produces the SIGNAL program depicted in Figure 4(b). The hierarchy of clocks is made visually evident by the nesting of parallel[1] subprocesses (the only subprocess in this example comprises lines `5-13`). The root of the only tree is that of the clock defined at the top, line `3`. This line also indicates that the clock `h_b` is not fixed, but a free variable which could have any value at any instant. Line `4` gives the set of signals that share the same clock (`h_b`). Lines `6,8` indicate what is the name of the clock of signals `x,sx`, respectively, whereas lines `5,7` give their definitions. Finally, lines `9-11` provide the definition of the `fifo_1` output `sx` (through the use of the value of an intermediate variable `tmp`).

Now the translation of the compiled `fifo_1` program into SMV is in Figure 5. Translation of the negated delay spans lines `13-20`; translation of the `cell` operator lays between lines `5-12`; and, the `when` operator is translated into line `24` (if the type of `tmp` was not Boolean then a `case` statement would have been used). There is a clear depart from the translation schemes presented in Section 3. This stems from several improvements in the translation (already suggested at the end of Section 3), and with some conventions in the compiler program generation, Figure 4(b). A first convention exploited in our translation says that all uses of `when` operator have as first operand a synchronous expression (i.e. all its signals share the same clock) and second operand a signal denoting a clock *smaller or equal* to that of the first operand. As a result, our translation into SMV (Figure 2(b)) need not test the clock (`h_x`) of the first operand together with the clock (`h_b`) and value (`b`) of the second operand; it suffices to guard the use of the first operand value by the clock given as second operand (i.e. expression `h_y <-> h_x & h_b & b` becomes `h_y <-> h_b & b`). The next convention states that occurrences of the `default` operator have the standard form `x:= (a when h_f) default (b when h_g)` (with possibly more `default` and their corresponding operators) where `h_f,h_g` are clocks and `a,b` are synchronous expressions. Note here that clock signal `h_g` should be defined (implicitly or explicitly) as the difference between the clocks of `x` and `h_f`. Our translation of this operator (Figure 3(a)) won't have to translate the `when` operator occurrences in such equations, they serve to identify the clock guard for each `case` branch of the `default` operator. Finally, to discuss the `cell` operator recall that, in general, an equation `x := y cell z init C` is equivalent to the two equations `x := y default (x$1 init C)` | `x ^= y ^+ when z`. Uses of such `cell` operator have as second operand the clock of the defined signal. That is, `z` above will be a signal denoting the clock of `x`, and `y` is either a synchronous expression or a `when` operator.

An explanation of the simplification in the translation (lines `13-20`, Figure 5) of the negated delay (line `12`, Figure 4(b)) is in order. Because the definition of `bw` is quasi-circular (through a function operator and a one instant delay) we do not need the extra `f` variable to detect the first

[1] The SIGNAL parallel composition operator is commutative and associative.

```
process fifo_1 = (? boolean x;
                  ! boolean sx;)
  (| sx := current_1(x,^sx)
   | interleave(x,sx) |)
  where
  process current_1 = (? boolean wx;
                         event c;
                       ! boolean rx;)
    (| rx := (wx cell c init false)
                       when c
     |);
  process interleave = (? boolean x,
                          sx; !)
  (| x ^= when b
   | sx ^= when (not b)
   | b := not(b$1 init false)
   |) where boolean b; end;
  end;
```

(a) One-place FIFO in SIGNAL

```
process fifo_1 = (? boolean x;
                  ! boolean sx;)
  (| h_b := ^ h_b
   | h_b ^= tmp ^= b
   | (| h_x := when b
      | h_x ^= x
      | h_sx := when (not b)
      | h_sx ^= sx
      | sx := tmp when h_sx
      | tmp := (x when h_x) cell
               h_b init false
      | b := not (b$1 init false)
      |)
   |) where event h_b, h_sx, h_x;
        boolean tmp, b; end;
end;
```

(b) `fifo_1` after clock calculus

Figure 4: `fifo_1` source: Before and after applying clock calculus

instant, neither do we need an extra state variable (the `x` variable in Figure 2(a)) to guarantee that the delayed value is the correct one. All the information, clock-wise and data-wise, is comprised in the same signal, hence the compact SMV code generation. A straightforward generalisation of this reasoning allows us to translate in the same way all equations with form: `y := f(x$1 init C)`, where `f` is a SIGNAL function operator.

A two-place FIFO. Let us now consider the translation of the two-place FIFO resulting from composing two one-place FIFOs, as shown by process `fifo_2` in Figure 6(a). For reasons of space we won't show the compiled version[2] of `fifo_2` but use the generated SMV code (Fig-

[2] The compiler automatically inlines all process instances.

```
MODULE fifo_1(x,h_x)
VAR
 h_b, b, tmp:  boolean;
ASSIGN
 init(tmp) := case
                h_x :  x;
                1 :  0;
              esac;
 next(tmp) := case
                next(h_x) :  next(x);
                1 :  tmp;
              esac;
 init(b) := case
              h_b :  1;
              1 :  0;
            esac;
 next(b) := case
              next(h_b) :  !b;
              1 :  b;
            esac;
DEFINE
 h_x := h_b & b;
 h_sx := h_b & !b;
 sx := h_sx & tmp;
```

Figure 5: One-place FIFO in SMV.

```
process fifo_2 = (?  boolean x;
                  !  boolean xok;)
  (| xok := fifo_1(
                fifo_1(x))
   |) where
    process fifo_1 ...
     where
     process current_1 ...
     process interleave ...
     end;
   end;
```

(a) Two-place FIFO in SIGNAL.

```
MODULE fifo_2(x, h_x)
VAR
 ff11:  fifo_1(x, h_x);
 ff12:  fifo_1(ff11.sx, ff1.h_sx);
INVAR
 ff11.h_sx <-> ff12.h_x
DEFINE
 xok := ff12.sx;
 h_xok := ff12.h_sx;
```

(b) Two-place FIFO in SMV.

Figure 6: SIGNAL and SMV: A two-place FIFO

ure 5), and compose two instances of `fifo_1` accordingly. An interesting feature of the `fifo_2` SIGNAL program is that it is not endochronous (unlike `fifo_1`) and thus has multiple (master) clocks. Its translation into SMV (Figure 6(b)) uses the same schemas as for endochronous programs though. Yet another feature of the generated code is the existence of a clock constraint in the form of an SMV invariant. This expression was not translated as a clock definition since the SIGNAL compiler was unable to verify its validity, hence its form of constraint rather than a directed assignment (as those appearing in a `DEFINE` section, for instance).

4.2 The whole communication protocol

We've applied the mentioned simplifications for the complete specification of the protocol proposed by Benveniste et al. [BCL+02] (see ftp://ftp.irisa.fr/local/signal/publis/SIG2SMV/ for the whole protocol and its translation). Our simplification rules with the aid of the compiler reduced the number of state variables from 98 to 27 (disregarding any possible reductions in the SMV internal representation of such models), with the ensuing improvements in verification time.

5 Some model checking

Here we will pose some CTL [CGP00] queries (and LTL whenever possible, in order to ease the reading) to our previous SMV programs (Section 4) in order to elucidate some behaviour information from the SIGNAL source or the SMV translation. Also, our queries aim at illustrating the use of the SMV clock variables introduced by the translation.

5.1 The need for FAIRNESS constraints

Recall the `fifo_1` SMV module (Fig. 5). We are interested to know whether the SMV translation correctly assigns `true` for the first instant (in SIGNAL) of `b`, given that the default case assigns `false` (i.e. `0`). Also recall that the first instant of an SMV program does not necessarily correspond to the first instant of some SIGNAL clocks. An LTL query could be as follows: `(!h_b U b)`. Our formula states that along *all* paths from the initial state(s) of the system our signal may remain absent until it is first present with value `1`. However, the SMV model checker says that our model fails to follow this LTL specification, and gives us a one-state trace to support such a response. A close examination of the counter-example shows that it is a state with a loop transition to it; that is, a behaviour of our system where the signal (`h_b`) is forever absent. This is a valid behaviour and is desirable for compositional reasons. For model checking, however, it is best to ignore behaviours consisting only of such self loops. Fortunately, SMV provides ways of ensuring that our queries are verified on (possibly looping) behaviours where something interesting happens. This may be achieved using SMV `FAIRNESS` statements to *restrict the verification to paths where such statements hold infinitely often*. Hence, for our `fifo_1` example, we added the following line: `FAIRNESS h_b`, and then our model verifies our LTL query above. Let us assume that appropriate `FAIRNESS` constraints have been added to all our examples and our LTL/CTL goals are to be verified along fair paths. Clearly the correct `FAIRNESS` statements refer to the clocks of the root(s) of the tree(s) found during clock calculus.

Now, we can check whether the Boolean (guard) signal (`b`) is alternating, by posing the LTL query: `G( ( (h_b & b) -> X(!h_b U !b) ) | ( (h_b & !b) -> X(!h_b U b) ) )`. By such formula we mean that all states where the signal is present and true are always followed by a sequence of states where the signal may be absent until it first arises (is present) with value false, or the converse (for the signal values only). As expected, the SMV answer is affirmative.

5.2 Some non-determinism

Let us now query the `fifo_2` module (Fig. 6(b)) where a stored value can only be retrieved (at least) two instants after it has been written, and not before. A CTL formula for inspecting whether given an input event (`h_x`), in the next instant, an output event (`h_xok`) is possible could be expressed as `AG(ff11.h_x -> EX(h_xok))`. For this goal the model checker answers *no* and gives a counter-example where every arrival of the output occurs two instants after an input was received. One may think that the output is *always* available exactly two instants after an input is placed, and thus pose the LTL query `G(ff11.h_x -> X(X(h_xok)))`. Unfortunately this is not the case, as shown by another counter-example generated by SMV; the first output arrives four instants after the first input and then every three instants after another input. This (apparently) non-deterministic behaviour is due to the polychronous nature of the SIGNAL source by virtue of the two instances of the `fifo_1` process (and more specifically, of the `interleave` process). Nonetheless we may assert that in general, there is always a behaviour for which after exactly two instants the output will arrive, in CTL: `AG(ff11.h_x -> EX(EX( h_xok )))`. Alternatively, we may claim that given the input the output will always eventually arrive, in LTL: `G( ff11.h_x -> F(h_xok) ))` and thus verify this with the model checker. Note that (in part) due to the imposed `FAIRNESS` constraints, given an input, the output will eventually arrive, even when the constraint is not on the input or output variables.

5.3 Correctness of the whole communication protocol

Before verifying the correctness of the protocol we succeeded in verifying the correctness of a claimed specification property (property number 16 [BCL+02]) of the protocol implementation:

never two writing events between two successive bus/buffer sampling events. Finally, we posed the same two CTL goals (to prove correctness of the protocol) to our SMV translation and thus confirmed the answer previously reported [BCL+02].

6 Related Work

Here we provide some comparison elements for work on model checking for three synchronous languages: ESTEREL, LUSTRE and SIGNAL, and work on model checking multi-clocked specifications outside the synchronous paradigm.

From the language expressivity perspective it is worth noting that ESTEREL and LUSTRE assume a master clock[3], while SIGNAL does not impose such a constraint. We may say that *the subset* of SIGNAL programs that are found to be endochronous by the compiler coincide with those synchronous programs with a single master clock.

For LUSTRE alone there is a model checker called LESAR [Ray06]. It is based on symbolic model checking too, and is able to reason about numerical constraints (convex polyhedra) on the transition systems, unlike SMV. However, LESAR is unable to validate liveness properties; only safety properties can be proved. A case study [BWL06] comparing model checking using LESAR and SMV (among other validation tools), shows the improved power of SMV compared with LESAR. In such comparison some translation from LUSTRE is used, but unfortunately it is not provided. Nonetheless, a close examination of the LUSTRE sources for their example shows that their programs were already single-clocked, and thus the translation into SMV appears much simpler than ours. Also, a manual translation from ESTEREL to LUSTRE is mentioned (not provided) to reach the facilities of SMV.

Two other transformations from LUSTRE to SMV are mentioned in [MMM05, MAWW05]. Neither of them provide the transformation rules used, nor the LUSTRE subset that could be translated.

For model checking ESTEREL programs we know of a proposal [MHM+95] that first translates such programs into an intermediate representation called *Boolean automata*, and then translates such programs into SMV. However, the actual definition and transformation into Boolean automata is not provided and it appears that not all such Boolean automata could be described in their version of SMV.

SIGALI [MRLS01] is the model checker for SIGNAL. It is tightly integrated with the (SIGNAL) compiler internal representation and optimisations. In addition to model checking, SIGALI is also useful for controller synthesis [BBG+01]. However, it does not generate counter-examples (nor witnesses). From the beginning, it was conceived as a decision procedure and some limited form of counter-example generation is possible for safety properties only. The problem of generating counter-examples may be cast as controller synthesis to somewhat project the source program on all the behaviours that lead to a given unsafe set of states. Such program projection may be interpreted as a set of counter-examples. As regards the input language, Sigali only supports Boolean and event signals whereas SMV has some (limited form of) integer reasoning and offers the possibility to bridge to bounded model checkers too. Nonetheless, in SIGALI, signal clocks need not always be explicit in LTL/CTL goals, they may remain implicit unlike our proposal for SMV.

[3] Esterel V7 appears to be multi-clocked, though.

We argue that making clocks explicit fosters a good understanding of the source specification, besides the potential feedback provided by counter-examples.

Outside the synchronous approach there is the work of Clarke et al. [CKY03] and the work of Ganai and Gupta [GG07]. In the context of bounded model checking, the former considers linear relations (equality and/or inequality) between clocks as input and then synthesises an automaton that describes all possible schedules of the clock ticks. Even though the system of linear relations may reference precise clock frequencies the synthesised automaton refers to logical instants, as in SIGNAL. Stuttering transitions are problematic for the automaton representation since it is not evident which is(are) the master clock(s), if any. The authors appear to circumvent the problem for their experiments but a definitive answer is missing. Their proposal is tightly dependent on their bounded model checker and the kind of properties checked appears to be restricted to safety issues only, whereas we are not dependent on safety properties and bounded model checking remains one possibility amongst several.

In a refinement of this work, Ganai and Gupta [GG07] propose a specialised translation of LTL goals for clocked specifications, which apparently render the bounded model checking scalable, for multiple-clocks. By contrast, we do not propose any model checking technique, neither an optimised translation of clocked LTL/CTL formulas. However, we propose a tightly integrated (with the SIGNAL compiler) translation from specifications with multiple clocks into SMV where bounded model checking is one option.

Last, but not least, SMV itself provides a syntax for composing (single-clocked) modules asynchronously, using the `process` keyword. This language feature offers the possibility to express some coarse-grained multi-clocked specifications without the need for extra explicit signaling (as is the case of our SMV Boolean variables to denote clocks). By contrast, in our source SIGNAL multi-clocked specifications clocks are finely interwoven. The challenge here is to derive a so-called GALS (globally-asynchronous locally-synchronous) description from the SIGNAL source multi-clocked specifications in order to match and profit from this SMV language feature (i.e. asynchronous composition of single-clocked modules).

7 Concluding Remarks

We have shown a simple source-to-source translation from SIGNAL (multi-clocked) specifications to single-clocked SMV programs for the purpose of CTL verification. Then we refine the translation taking into account the compiler analysis of the source SIGNAL program, in order to reduce the number of state variables added by the translation. This optimisation allows us to eliminate signal variables as well as (Boolean) clock variables. We stick to a syntax compatible with the three versions of SMV currently available. We presume soundness of our translation given the semantic proximity of the two languages and because the SMV coding neatly reflects the clock relations (using invariants or definitions) and data-flow (with assignments). The generality of our proposed translation is exercised through modeling and verification of SIGNAL specification with multiple master clocks. There are two additions to common/standard use of SMV and CTL for model validation, namely, *(a)* clocks are explicit in SMV and LTL/CTL goals, and *(b)* fairness constraints are needed for ensuring reactivity of the model behaviours; such constraints refer to the clocks of all the roots found during clock calculus.

Boolean SMV variables are used to model SIGNAL clocks. The translation automatically adds the Boolean clocks, and it is the user who will be responsible for a correct combination of clocks and signals while querying (in LTL or CTL) the produced SMV model. The chief condition for a sound use of explicit clocks is that *the value of a signal is only meaningful when its clock evaluates to* `true`. As a result, the user is only concerned with knowing the name of the clock of a signal and for every occurrence of the signal name (in a temporal formula) add the conjunct to test its presence (by referring to a true occurrence of its clock variable).

Future work. Here we only used the LTL/CTL verification functionality of SMV. We plan to experiment with other functionalities (bounded model checking, bounds analysis, refinement checking, induction, and compositional verification). In order to reduce the load to the user on combining clocks and signal values while querying the SMV model, we envisage automating the addition of clocks to temporal formulas without them.

Bibliography

[ABL94] T. P. Amagbegnon, L. Besnard, P. Le Guernic. Arborescent canonical form of Boolean expressions. Technical report 2290, Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex, France, 1994.

[ABL95] T. P. Amagbegnon, L. Besnard, P. Le Guernic. Implementation of the dataflow synchronous language SIGNAL. In *Conference on Programming Language Design and Implementation, PLDI95*. ACM Press, 1995.

[BB91] A. Benveniste, G. Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*. Volume 79(9), pp. 1270–1282. September 1991.

[BBG$^+$00] L. Besnard, P. Bournai, T. Gautier, N. Halbwachs, S. Nadjm-Tehrani, A. Ressouche. Design of a Multi-formalism Application and Distribution in a Data-flow Context: An Example. In Gergatsoulis and Rondogiannis (eds.), *Intensional Programming II*. Pp. 149–167. World Scientific, 2000.

[BBG$^+$01] A. Benveniste, P. Bournai, T. Gautier, M. Le Borgne, P. Le Guernic, H. Marchand. The SIGNAL declarative synchronous language: controller synthesis & systems/architecture design. In *Conference on Decision and Control*. Pp. 3284–3289. 2001.

[BCL$^+$02] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P. Talpin, S. Tripakis. A Protocol for Loosely Time-Triggered Architectures. In *EMSOFT 2002*. Pp. 252–265. 2002.

[BG92] G. Berry, G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19(2):87–152, 1992.

[BGL08] L. Besnard, T. Gautier, P. Le Guernic. SIGNAL V4-INRIA version: Reference Manual. March 2008. http://www.irisa.fr/espresso/Polychrony/.

[BWL06] F. Boniol, V. Wiels, E. Ledinot. Experiences in using model checking to verify real time properties of a landing gear control system. In *Conference on Embedded Real-Time Systems*. January 2006.

[CCJ+05] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev. NuSMV 2.4 User Manual. ITC-irst, Via Sommarive 18, 38055 Povo (Trento), Italy, 2005. http://nusmv.irst.itc.it.

[CGP00] E. M. Clarke (Jr.), O. Grumberg, D. A. Peled. *Model Checking*. The MIT Press, 2000.

[CKY03] E. M. Clarke, D. Kroening, K. Yorav. Specifying and Verifying Systems with Multiple Clocks. In *International Conference on Computer Design (ICCD'03)*. P. 48. 2003.

[GG07] M. K. Ganai, A. Gupta. Efficient BMC for Multi-Clock Systems with Clocked Specifications. In *Asia and South Pacific Design Automation Conference*. Pp. 310–315. 2007.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*. Volume 79(9), pp. 1321–1336. September 1991.

[LTL03] P. Le Guernic, J.-P. Talpin, J.-C. Le Lann. Polychrony for System Design. *Journal of Circuits, Systems, and Computers*, March 2003.

[MAWW05] S. P. Miller, E. A. Anderson, L. G. Wagner, M. W. Whalen. Formal Verification of Flight Critical Software. In *AIAA Guidance, Navigation and Control Conference and Exhibit*. August 2005.

[McM99] K. L. McMillan. The SMV language. March 1999.
http://www.kenmcmil.com/smv.html

[McM01] K. L. McMillan. The SMV system (version 2.5.4). 2001.
http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps

[MHM+95] M. Müllerburg, L. Holenderski, O. Maffeïs, A. Meceron, M. Morley. Systematic Testing and Formal Verification to Validate Reactive Programs. *Software Quality Journal* 4(4), 1995.

[MMM05] M. Moy, F. Maraninchi, L. Maillet-Contoz. LusSy: An open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems* 10(2-3):73–104, 2005.

[MRLS01] H. Marchand, E. Rutten, M. Le Borgne, M. Samaan. Formal verification of programs specified with SIGNAL: application to a power transformer station controller. *Science of Computer Programming* 41(1):85–104, 2001.

[Ray06] P. Raymond. Vérification de programmes synchrones avec LUSTRE/LESAR. In Navet (ed.), *Systèmes temps réel 1*. Pp. 181–216. Hermes science publications, Lavoisier, 2006.