Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 264 (2010) 3–17
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Handling State-Machines Specifications with GATeL [1]

Benjamin Blanc[a] Christophe Junke[a] Bruno Marre[a]
Pascale Le Gall[b] Olivier Andrieu[c]

[a] *CEA, LIST – Software Reliability Lab (LSL)*
`firstname.lastname@cea.fr`

[b] *MAS Laboratory– École Centrale Paris & Université d'Evry*
`pascale.legall@ecp.fr`

[c] *Esterel Technologies*
`olivier.andrieu@esterel-technologies.com`

Abstract

GATeL proposes a testing environment for Lustre/Scade programs. Its main component is a resolution procedure based on a CLP interpretation of its input language. This paper presents a two-tier extension of GATeL in order to take into account state-machines descriptions. This extension relies on a compilation of these constructs into multi-clocked expressions. Our first contribution is a definition of explicit constraints to manage the clock type hierarchy in GATeL. The second one is the definition of constraints reflecting properties of state-machines built by the Scade compilation schema.

Keywords: Reactive systems, State-Machines, Test Generation, Constraint Logic Programming

1 Introduction

GATeL [3,16] belongs to the family of CLP-based test generation tools [2,5,11,13,19,21]. It allows the generation of complex test sequences for programs developed with Lustre [7] and its industrial implementation Scade. Such automatic generation tools are of great interest in the context of certification processes imposed to these programs (DO-178B, IEC-61508), and complementary to existing model-checking techniques [4,18,20]. The latest version of the language, Scade 6 [12], embodies a well known feature of I&C programs: their decomposition into different modes. Modes are represented at a high level using state-machines [15]. Currently, GATeL can only manage such state-machine descriptions through systematic boolean encoding [1]. The transition relation being hidden by the encoding, this solution is not efficient.

[1] This work is supported by project ANR-07-TLOG-019 and IRSN nuclear safety agency.

doi:10.1016/j.entcs.2010.12.011

However, for code generation purposes, Scade's state machine constructs are compiled onto an intermediate multi-clocked language [8]. This compilation process provides a structure to identify states and transition relations of automata in terms of clocked flows. Due to its proximity to the input language of GATeL, we therefore decided to handle this intermediate language and its multi-clocked operators. One way to tackle these new operators would be to define their CLP interpretation by a combination of existing single-clock operators. But this simulation also loses the information provided by the strong clock typing, and therefore is not efficiently handled by GATeL. Our claim is that exploiting clock types and operators within GATeL makes it possible to address realistic state-machine specifications, which includes Scade 6 models, but also clock-based translations from other block-diagram models (e.g., Stateflow/Simulink [6]).

The contribution of this paper is twofold: first, it describes an extension of GATeL for these new multi-clocked operators, then it proposes some improvements based on the compilation process to refine state-machine constraints. We first present the Lustre language and its associated notion of clocks in Section 2. A glimpse of the translation process between state-machines and a clocked program is then given. Section 3 introduces the basic mechanisms used in GATeL, then presents the major updates used for the clock extension. The next step is the definition of specific state-machine deduction rules in Section 4. Finally, in Section 5, a case study based on a simplified Cruise Control model demonstrates the efficiency of our proposal compared to the first two solutions.

2 The State Machine Extension of Lustre

Lustre Acts on Flows.

Lustre belongs to the synchronous data-flow family of languages used for the description of critical reactive systems. The basic structuring unit, called a **node**, allows to declare inputs and outputs with their type (**int**, **real**, **bool**, or user defined enumerated type), and gives for each output a value computed accordingly to its defining equation. Each iteration of the reactive loop consists in reading sensor values as inputs, then computing outputs. Any Lustre expression thus denotes an infinite data flow: the constant 3 corresponds to the flow (3,3,3...), the expression $A + B$ to the flow $(A_0 + B_0, A_1 + B_1, ...)$. Since Lustre follows the synchronous hypothesis, at each instant of the loop all flows have the same length. Most of the time, Lustre is used to model closed-loop programs: outputs are actuator commands that will influence next sensor values. It is then necessary to refer to past values in order to model complex behaviours. The synchronous hypothesis allows to uniquely refer to the past value of a flow: $\mathbf{pre}(X) = (nil, X_0, X_1, ...)$. In order to prevent the undefined nil value, this operator must be protected by an initialisation one: A `->` $B = (A_0, B_1, B_2, ...)$. Usual boolean (**and**, $\leq$,...) and arithmetic (+,-,...) operators are also available to define output expressions.

h	t	f	t	f	t	f	t
a	a_0	a_1	a_2	a_3	a_4	a_5	a_6
b	b_0	b_1	b_2	b_3	b_4	b_5	b_6
a **when** (h,t)	a_0		a_2		a_4		a_6
pre(a **when** (h,t)))	*nil*		a_0		a_2		a_4
pre(a) **when** (h,t)	*nil*		a_1		a_3		a_5
merge(h;a **when** (h,t);b **when** (h,f))	a_0	b_1	a_2	b_3	a_4	b_5	a_6

Fig. 1. Execution traces for clocked flows.

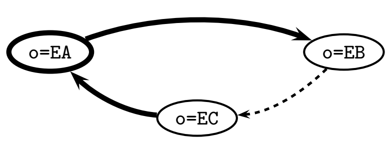
Flows Can Be Clocked.

The other fundamental notion of the Lustre kernel is its ability to design computations rated by clocks. Clocks allow to relax the synchronous hypothesis in a safe way: flows can have different lengths, but this desynchronisation only occurs in a limited way, precisely described by dedicated operators. Table 1 illustrates a clocked execution. A flow can be sampled with a **when** operator: X **when** (Y, v). When Y evaluates to v, the flow expression evaluates to X; otherwise it has no value. Pointwise application of standard operators is extended so that it only holds for flows of the same rate. A clock calculus is then defined following [10] in order to assign each expression with a clock type. Thus, the expression "a+(a **when** (h,t))" cannot be evaluated since its operands do not have the same clock type. The logical time represented by the successive iterations of the reactive loop defines the base clock. All other clock types are derived from this base clock through the sampling operator. Only boolean or enumerated identifiers are allowed for the sampling clock Y and v is a value belonging to its data type. Finally, flows on complementary clocks (*i.e.,* that rely on the same identifier but cover all the possible values of its data type) can be recombined using the **merge** operator. Note also the subtle distinction between the two **pre** expressions: the first one evaluates the **pre** only when h takes the value t, while the second evaluates the **pre** at every cycle, then samples the results when h is t.

Designing Control with Clocks.

One of the major drawbacks of data-flow languages is that they provide poor help in the definition of control structures. However, most reactive systems require a mix of control structures and data-flows (*e.g.,* different equations defining an output for each running mode). We consider the Scade 6 version of automata constructs, based on a conservative clocked extension of data-flows [8], for which two major design rules are ensured: at each cycle, there is only one active state and one fired transition. An important feature is that state-machines can be hierarchical: the body of a state can itself be defined as a state-machine.

Let us consider the state-machine in Figure 2 that defines some output *out* with respective expressions E_A, E_B, E_C. Bold edges correspond to *strong* transitions, that is those that may be triggered according to conditions on current inputs and all preceding values. Dashed edges correspond to *weak* transitions that can be triggered, assuming no strong one was fired during the current cycle, according to conditions on any input and output. The revised compilation process first defines



Sel	A	B	C	A
Act	B	B	A	B
Nxt	B	C	A	B
out	EB	EB	EA	EB

Fig. 2. A state machine

an enumerated type that represents the states (one enumerator per state). Three flows based on this type are created to carry out the dynamic semantics of the state-machine: at each cycle *Sel* identifies the state which can trigger strong transitions, *Act* the one which evaluates state bodies and may trigger weak transitions, and Nxt which defines the next selected state. In a state-machine without any strong transitions, we thus get *Act*=*Sel*; and reciprocally *Act*=Nxt if no weak transition occurs. These three flows are automatically defined from the structure of the state-machine.

```
Sel = A -> pre(Nxt);
Act = merge(Sel;B when (Sel,A);B when (Sel,B);A when (Sel,C));
Nxt = merge(Act;A when(Act,A);(if Sel=A then B else C) when(Act,B);
                C when(Act,C));
out = merge(Act;EA when (Act,A);EB when (Act,B);EC when (Act,C));
```

Sel is defined as the initial state (here A) at the first cycle and as the previous value of Nxt otherwise. *Act* is defined by a **merge** on *Sel*, and covers for each state the possible strong transitions. Nxt is defined by a **merge** on *Act* and considers weak transitions (here only in state B) when no strong one was fired at the same cycle. Outputs of the state machine are then defined by a **merge** on *Act*, gathering all defining expressions into a single one. Since each expression has a different clock type, the clock calculus ensures that at most one expression is computed within a cycle. Assuming that all transitions are unconditional in our example, one gets the execution of the table above.

3 A Multi-Clock Extension for GATeL

GATeL [3, 16] is a testing tool based on a dedicated resolution procedure. This procedure relies on an interpretation of Lustre operators through constraints programming. A first possible way to manage the clocked extension is to simulate the behaviour of **when**, **merge**, **pre** and -> by a combination of single-clock existing ones. For instance, X **when** (Id, v) can be simulated by **if** $Id{=}v$ **then** X **else** $(default$ -> **pre** $X)$, returning a $default$ value when its clock has never been present and maintaining the previous value since the last occurrence of its clock. This simulation exactly matches the original expression on each instant of the clock (Id, v). However, as it will be illustrated in Section 5, the CLP interpretation of this simulation is less efficient than a direct interpretation of these operators.

Internal Representation

Given a Lustre model and a reachability objective, GATeL automatically generates a sequence of inputs leading to this objective. The reachability objective is described as a boolean Lustre expression. This objective is stated using a dedicated directive, noted **reach**, which may be added in the body of a node. From a testing description gathering the model with an identified root node, and a reachability objective, GATeL first compiles these elements into a static logic program $\mathcal{P} = (\mathcal{V}ars, \mathcal{E}, \mathcal{R})$. $\mathcal{V}ars$ is a set of Lustre identifiers relevant to the given objective, $\mathcal{E}(Id)$ is a term corresponding to the defining equation of output or local identifier $Id \in \mathcal{V}ars$, $\mathcal{R}$ is a boolean term denoting the reachability objective. These terms are naturally built over an abstract syntax reflecting all Lustre operators and logical variables in place of Lustre identifiers. Each logical variable is interpreted in the following domains, according to its associated Lustre type : **bool** in $[f, t]$ finite domain, **int** in standard interval integer arithmetic. **real** either in double precision floating point interval arithmetic, or in real interval arithmetic, and enumerated types in their corresponding finite domain.

Clock Hierarchy

The multi-clock extension of GATeL consists in the use of an explicit notion of the clock types. Each Lustre identifier Id in $\mathcal{V}ars$ and reachability objective $\mathcal{R}$ has a clock type *cl* computed following the type system of [10], noted $(\vdash_{\mathcal{H}} Id : cl)$. A clock type *cl* is either the base clock **base** (representing the fastest flow rate of the program) or a Lustre flow identifier Id (the carrier flow of the clock type) belonging to an enumerated type, accompanied with its truth value v (the enumerator it relies on):

$$cl ::= \mathbf{base} \mid (Id, v)$$

Since the carrier flow of a clock type may be defined wrt. another clock type, this defines a hierarchy of clocks representing a partial order on clock types. This hierarchy $\mathcal{H}$ is synthesised by GATeL and added to the logical program $\mathcal{P}$. The root of $\mathcal{H}$ is the **base** clock type. Given the clock type *cl* of an expression X and of the identifier Id, X **when** (Id, v) sets (Id, v) as a sub-clock of *cl*. This hierarchy is equipped with two navigation functions giving for each clock type *cl* its unique ancestor cl^+ and the set of its sub-clocks cl^-. For a clock type *cl*, $cl.id$ represents its carrier flow identifier, and $cl.v$ its truth value. By convention, $\mathbf{base}.id$ is equal to $\mathbf{base}.v$.

Resolution Engine

This program $\mathcal{P}$ is used by the main component of GATeL: its resolution engine. The goal of this engine is to produce a matrix $\mathcal{M}$ whose rows are identifiers from $\mathcal{V}ars$ and columns are cycles. For each known cycle C, $\mathcal{M}(Id, C)$ stores the logical variable associated to Id whose domain is initialised according to its type. New cycles are created on demand by the resolution engine, extending $\mathcal{M}$ accordingly. As a standard CLP algorithm [14], the resolution engine alternates between deterministic and non-deterministic phases, the former being managed by a set of filtering rules based on the definition of Lustre operators (see section 3). These deduction rules involve two notions of cycles. The first one represents the current cycle C at which an equation holds. The second one is a vector representing the highest known cycle $C_M(cl)$ of each clock type cl in $\mathcal{H}$. Cycle numbering is indexed on the **base** clock numbering. A second vector of logical variables stores the status $S_M(cl)$ of each highest known cycle. A status variable has for domain $[init, non\_init]$. These two vectors will be useful to interpret the `->` operator. The filtering rules are denoted following an usual logical notation $\Gamma \vdash \Delta$. Δ is a set of couples (Eq, C) of equation Eq and cycle number C; Γ contains the matrix $\mathcal{M}$, the two vectors C_M and S_M, together with $\mathcal{S}$ made of the residual triples (Eq_i, C_i, V_i) whose couple (Eq_i, C_i) cannot be filtered out by our dedicated rules due to the insufficient information carried out by the variables V_i. These triples correspond to the constraints that remain to be solved.

The resolution algorithm first initialises the environment with an empty matrix $\mathcal{M}$ and residual store $\mathcal{S}$; the highest known cycle C_M of the **base** clock is set to zero and its status S_M is variable. It then starts by a first deterministic phase launched with (1) the clock type of the objective set to true, since the test sequence ends when the objective is realised and therefore its clock must be present, and with (2) the reachability objective set to true. The current cycle for these two equations is zero, which represents the final cycle of the sequence and the first column of matrix $\mathcal{M}$. Cycles are numbered backward from the last cycle to an initial one. The deterministic phase goes on until a fix-point is reached over the set of filtering rules. Note that this filtering may fail when detecting an unsatisfiability. Then, as with standard CLP algorithms, a variable is chosen among the ones occuring in the remaining constraints, along with a value in its allowed domain. The filtering rules are applied until another fix-point is reached. Any failure in the filtering rules cancels the previously made choice, by backtracking. In order to get a terminating algorithm, global bounds on the length of the sequences (noted Max_C) and on the numerical domains are positioned by the user. When the algorithm stops with no solution (failure for each valuation of $\mathcal{M}$), this only implies that there is no solution within the user bounds.

Filtering Rules

The filtering rules implement the operational semantics of the Lustre operators. Rules are applied according to a selection strategy that chooses a couple (Eq, C) from the current set Δ, then chooses a rule according to the head operator of the right member of Eq. When no rule can be applied on a chosen equation, a generic storing rule is applied:

$$\frac{\vdash_{\mathcal{H}} Eq : cl \quad V \subseteq FreeVars(Eq) \cup S_M(cl) \quad \Gamma \cup (Eq, C, V) \vdash \Delta}{\Gamma \vdash (Eq, C) \cup \Delta}\ \text{STORE}$$

The constraint (Eq, C, V) is added to the residual store $\mathcal{S}$ in Γ, waiting that a domain reduction occurs on a set of variables V defined as a subset of the free variables in Eq and the status of the clock type *cl* of the equation. Such a reduction leads to the introduction of the couple (Eq, C) back in Δ. We present in the sequel only rules involved in the management of the clock hierarchy. Those concerning arithmetic and boolean operators are managed by a general CLP library (COLIBRI) also used by other testing tools [2, 21].

Variables.

Rule VAR-I concerns input variables, while VAR-OL concerns output and local ones.

$$\frac{\vdash_{\mathcal{H}} Id : cl \quad \Gamma' \vdash \{(cl.id = cl.v, C)\} \cup \Delta}{\Gamma \vdash (\mathbf{var}(Id) = R, C) \cup \Delta}\ \textsc{var–i}$$

$$\frac{\vdash_{\mathcal{H}} Id : cl \quad \Gamma' \vdash \{(cl.id = cl.v, C), (\mathcal{E}(Id) = R, C)\} \cup \Delta}{\Gamma \vdash (\mathbf{var}(Id) = R, C) \cup \Delta}\ \textsc{var–ol}$$

where Γ' is Γ with $\mathcal{M}(Id, C)$ unified with R and the clock hierarchy possibly updated: if (Id, R) denotes a clock type in $\mathcal{H}$, and its highest known cycle $C_M(Id, R)$ is lower than C, then C becomes the new highest known cycle. The previous status is instantiated to *non_init* in order to inform residual constraints that could progress with this information. A fresh variable is then created to denote the new status $S_M(Id, R)$. Since a value for Id is needed at cycle C, its clock type is necessarily present at the same cycle. The corresponding equation is added to Δ. Finally, in rule VAR-OL, $\mathcal{E}(Id)$ is the defining equation for identifier Id. The update of global parameter C_M can then lead to a cascade of updates according to the following properties:

Proposition 3.1

(i) $\forall cl \in \mathcal{H} \quad C_M(cl) \le C_M(cl^+)$.

(ii) $\forall cl \in \mathcal{H}, S_M(cl^+) = init \Rightarrow (C_M(cl) < C_M(cl^+) \vee S_M(cl) = init)$

Indeed, the first item states that the highest known cycle of a clock type *cl* must always be greater than the highest known one for each of its sub-clocks $cl' \in cl^-$. To ensure this invariant, our engine systematically propagates updates upward in $\mathcal{H}$. Moreover, the second item allows to instantiate the fresh status to *init* when the status of the ancestor clock type is already set to *init* and its highest known cycle is the same as the updated one.

Sampling and Reconstruction Operators.

The filtering rule of the **when** operator simply traverses this operator. Note that surprisingly, there is no need to ensure the presence of the clock type of X at cycle C. This is a consequence of rules VAR and of the introduction of similar presence condition for the reachability objective. This will be the same for any operator in the sequel.

$$\frac{\Gamma \vdash (X = R, C) \cup \Delta}{\Gamma \vdash (X\ \mathbf{when}\ cl = R, C) \cup \Delta}\ \textsc{when}$$

The first rule of the **merge** operator introduces the definition of the clock identifier with a fresh variable R', and tags the operator so that it is done only once. The second rule states that if the clock identifier is instantiated with a value v_i at cycle C, then the filtering procedure switches to the corresponding expression.

$$\frac{\Gamma \vdash \{(\mathbf{var}(Id) = R', C), (\mathbf{merge}'(Id; ...; X_i \ \mathbf{when}\ cl_i; ...) = R, C)\} \cup \Delta}{\Gamma \vdash (\mathbf{merge}(Id; ...; X_i \ \mathbf{when}\ cl_i; ...) = R, C) \cup \Delta} \ \textsc{mer-i}$$

$$\frac{\mathcal{M}(Id, C) = v_i \quad cl_i = (Id, v_i) \quad \Gamma \vdash (X_i \ \mathbf{when}\ cl_i = R, C) \cup \Delta}{\Gamma \vdash (\mathbf{merge}'(Id; X_1 \ \mathbf{when}\ cl_1; ...; X_n \ \mathbf{when}\ cl_n) = R, C) \cup \Delta} \ \textsc{mer-s}$$

$$\frac{\begin{array}{c}\{j \mid v_j \in dom(\mathcal{M}(Id, C)) \wedge \Gamma \vdash (X_j, C) \rightsquigarrow R_j \wedge dom(R_j) \cap dom(R) \neq \emptyset\} \\ \Gamma' \vdash \{(\mathbf{merge}'(Id; ...; X_j \ \mathbf{when}\ cl_j; ..) = \sigma R, C)\} \cup \Delta\end{array}}{\Gamma \vdash (\mathbf{merge}'(Id; ...; X_i \ \mathbf{when}\ cl_i; ...) = R, C) \cup \Delta} \ \textsc{mer-r}$$

The third rule is used to discard some branches of the **merge** when their domain is incompatible with the domain of the result. $\Gamma \vdash (X_i, C) \rightsquigarrow R_i$ is a partial evaluation of X_i at cycle C returning R_i. Γ' is the update of Γ where the domain of $\mathcal{M}(Id, C)$ only contains valid v_j values. Finally σR denotes the reduction of $dom(R)$ to $dom(R) \cap \cup_j dom(R_j)$.

Temporal Operators.

The filtering rules for the **pre** operator require an increment to define the cycle at which its argument should be filtered. Two cases are possible, depending on the clock type of its argument.

$$\frac{\vdash_{\mathcal{H}} X : \mathbf{base} \quad \Gamma \vdash (X = R, C+1) \cup \Delta}{\Gamma \vdash (\mathbf{pre}\ X = R, C) \cup \Delta} \ \textsc{pre-b}$$

$$\frac{\vdash_{\mathcal{H}} X : (Id, v) \quad C' = min(j \mid j > C \wedge \mathcal{M}(Id, j) = v) \quad \Gamma \vdash (X = R, C') \cup \Delta}{\Gamma \vdash (\mathbf{pre}\ X = R, C) \cup \Delta} \ \textsc{pre-c}$$

When considering only **base** clock (rule PRE-B), the increment is one since the **base** clock represents the fastest flow rate. When argument X is based on a clock type (Id, v) the previous cycle corresponds to the smallest cycle C' where the flow identifier Id has taken the truth value v and never been unknown until C. In both cases, the correct initialisation of the program [9] ensures that $C + 1$ (resp. C') is lower than the highest known cycle of its corresponding clock type. When no such cycle can be found in the non **base** case, two sub-cases may be considered in order to allow the procedure to progress. Rule PRE-N1 states that if the carrier flow Id does not contain v in its domain until the highest known cycle of the **base** clock type, then a new cycle is needed. The filtering is then called recursively in environment Γ' in which $C_M(\mathbf{base})$ is incremented by one and a fresh status is associated, the old $S_M(\mathbf{base})$ being set to *non_init* in order to inform residual constraints. The recursive call intends to handle possible reductions provided by the creation of a new cycle on **base** clock. The user bound Max_C ensures a limited number of applications of this rule.

$$\frac{\begin{array}{c}\vdash_{\mathcal{H}} X : (Id, v) \quad \forall i \in [C..C_M(\mathbf{base})]\ v \notin dom(\mathcal{M}(Id, i)) \\ \Gamma' \vdash (\mathbf{pre}\ X = R,\ C) \cup \Delta\end{array}}{\Gamma \vdash (\mathbf{pre}\ X = R, C) \cup \Delta}\ \textsc{pre-n1}$$

$$\frac{\begin{array}{c}\vdash_{\mathcal{H}} X : (Id, v) \quad C' = min(j \mid j > C \wedge v \in dom(\mathcal{M}(Id, j))) \\ \Gamma \vdash \{(\mathbf{var}(Id) = R', C'), (\mathbf{pre}\ X = R, C)\} \cup \Delta\end{array}}{\Gamma \vdash (\mathbf{pre}\ X = R, C) \cup \Delta}\ \textsc{pre-n2}$$

Rule PRE-N2 calls the filtering procedure with the carrier flow set with a fresh variable R'. This equation should hold at the smallest cycle C' greater than the current cycle C where the domain of Id contains v. This intends to get more information in order to know whether cycle C' is really the previous cycle of clock type (Id, v).

Rule INIT-T states that if the current cycle is the highest known one of the clock type *cl* and its status is *init*, then the filtering procedure applies on its first argument. Rule INIT-F is complementary: when the current cycle is smaller than $C_M(cl)$, this means that C is non initial. Therefore the filtering procedure applies on the second argument.

$$\frac{\vdash_{\mathcal{H}} X : cl \quad S_M(cl) = init \quad C_M(cl) = C \quad \Gamma \vdash (X = R, C) \cup \Delta}{\Gamma \vdash (X\texttt{->}Y = R, C) \cup \Delta}\ \textsc{init-t}$$

$$\frac{\vdash_{\mathcal{H}} X : cl \quad C_M(cl) > C \quad \Gamma \vdash (Y = R, C) \cup \Delta}{\Gamma \vdash (X\texttt{->}Y = R, C) \cup \Delta}\ \textsc{init-f}$$

Rule INIT-NB allows to create a new cycle for the **base** clock: If the domain of the first argument is incompatible with that of the result at cycle C_M, then the status is necessarily *non_init*. Γ is updated to increment $C_M(\mathbf{base})$ and create a fresh status. The user bound Max_C on the length of sequences is used here to limit applications of this rule.

$$\frac{\begin{array}{c}\vdash_{\mathcal{H}} X : \mathbf{base} \quad C_M(\mathbf{base}) = C \quad dom(X) \cap dom(R) = \emptyset \\ C + 1 \leq Max_C \quad \Gamma' \vdash (Y = R, C) \cup \Delta\end{array}}{\Gamma \vdash (X\texttt{->}Y = R, C) \cup \Delta}\ \textsc{init-nb}$$

Considering the non **base** case, the status is instantiated to *non_init* in Γ' but the highest known cycle cannot be incremented here. Indeed, since the previous cycle depends on the instantiations of the carrier flow, this update is only performed by the rules for variables. The existential premise requires the existence of such previous significant cycle for the clock type of X. This can be handled by a dedicated constraint not shown here.

$$\frac{\begin{array}{c}\vdash_{\mathcal{H}} X : (Id, v) \quad C_M(Id, v) = C \quad dom(X) \cap dom(R) = \emptyset \\ \exists C' > C_M(Id, v) \ s.t. \ \mathcal{M}(Id, C') = v \\ \Gamma' \vdash (Y = R, C) \cup \Delta\end{array}}{\Gamma \vdash (X\texttt{->}Y = R, C) \cup \Delta} \ \textsc{init-nc}$$

Finally, rule INIT-I corresponds to the symmetrical case where the domain of the second argument is incompatible with the domain of the result.

$$\frac{\begin{array}{c}\vdash_{\mathcal{H}} X : (Id, v) \quad C_M(Id, v) = C \quad dom(Y) \cap dom(R) = \emptyset \\ \Gamma' \vdash (X = R, C) \cup \Delta\end{array}}{\Gamma \vdash (X\texttt{->}Y = R, C) \cup \Delta} \ \textsc{init-i}$$

where the corresponding clock status is instantiated to *init* in Γ'. This has many consequences on the status of the surrounding clock types in $\mathcal{H}$ and their carrier flows.

Proposition 3.2 *Let $cl = (Id, v)$ be the clock type of X. Assuming that $S_M(cl) = init$ the following facts hold:*

(i) $\mathcal{M}(Id, C_M(cl)) \equiv v$

(ii) $\forall cl' \in \mathcal{H}, (cl.id = cl'.id \wedge C_M(cl) = C_M(cl')) \Rightarrow S_M(cl') = non\_init$

(iii) $S_M(cl^+) = init \Rightarrow \forall i \in [C_M(cl) + 1 \, .. \, C_M(cl^+)], \ (\mathcal{M}(cl^+.id, i) = cl^+.v \Rightarrow cl.v \notin dom(\mathcal{M}(Id, i)))$

(iv) $\forall cl' = (Id', v') \in cl^-, \mathcal{M}(Id', C_M(cl)) = v' \Rightarrow S_M(cl') \equiv init$

The first property makes a direct link between $\mathcal{H}$ and $\mathcal{M}$: when a clock status is initial, then its corresponding carrier flow must be instantiated to its truth value. The second one states that all the clock types based on the same identifier but for a different truth value should be instantiated to *non_init* at the current highest known cycle of *cl* (when their status has not already been set to *init* at a lower cycle). Note that the existential hypothesis of rule INIT-NC should also be ensured for these clocks. Third property states that if the ancestor clock type cl^+ is already set to *init*, then for each cycle from $C_M(cl)$ to $C_M(cl^+)$, the carrier flow Id should not contain its truth value v. Note that this should also be ensured by a dedicated constraint. The last property states that for all the sub-clocks in cl^- whose carrier flow is already instantiated to its truth value, then their corresponding status should be set to *init*, since in this case the highest known cycles are necessarily equal. As for Proposition 1, these properties are handled by dedicated constraints not developed here.

Example 3.3 Consider again the tiny state-machine example of Section 2, with the following equations for output `out`:

```
EA = 0    EC = 0
EB = (0 when(Act,B)) -> pre(out when(Act,B)) + (1 when(Act,B))
```

Computed values for `out` are then: `0,1,0,2,3...` The samplings in equation E_B are introduced by the Scade 6 compilation process, while the user just typed in state B `out = 0 -> pre out + 1`. Assume that a reachability objective is: **reach** `out=3`, one can easily guess that at least five cycles are needed which is confirmed by the following derivation. $X : \{v_1 ... v_k\}$ is a notation for the domain of X, and we adopt the convention "**Rule Name** (Selected equation in Δ)".

MER-I (out_0:$\{3\}$ = **merge**(Act;E_A;E_B;E_C), 0)
Introduces the reachability objective at cycle 0. The clock identifier Act is added to Δ in order to fetch its domain.

VAR-OL (Act_0 = **var**(Act), 0)
Introduces the equation for Act at cycle 0, with the fresh variable Act_0.

MER-I (Act_0 = **merge**(Sel;B;B;A), 0)
Moreover the clock identifier, here Sel, is also added to Δ.

VAR-OL (Sel_0 = **var**(Sel), 0)
Introduces the defining equation for Sel at cycle 0.

STORE ($Sel_0 : \{A, B, C\} = A$ **->pre**(Nxt), 0, S_M(**base**)) in $\mathcal{S}$
No rule can be applied on this equation.

MER-R ($Act_0 : \{A, B\}$=**merge'**(Sel;B;B;A), 0)
The defining term for Act is then reduced, removing C from its domain since it not reachable by any of its branches.

STORE ($Act_0 : \{A, B\}$=**merge'**(Sel;B;B;A), 0, (Act_0,Sel_0)) in $\mathcal{S}$

MER-R (out_0:$\{3\}$ = **merge'**(Act;E_B), 0)
Similarly, the defining term for out is reduced to a single branch, since the other ones ($E_A = 0$ and $E_C = 0$) are incompatible with the result 3.

MER-S (3 = 0 **when**(Act, B) **->** **pre**(out **when**(Act, B)) + 1 **when**(Act, B), 0)
*The remaining branch of the **merge** is then explored with result 3.*

INIT-NC (3 = **pre**(out **when**(Act, B)) + 1, 0)
*The result is incompatible with the initial case. The clock status $S_M(Act, B)$ is set to non_init, and its C_M is not incremented yet. The existential hypothesis and Proposition 1 allow us to set C_M(**base**) to 1. The right member is then explored.*

PRE-N2 (2 = **pre**(out **when**(Act, B)), 0)
No past cycle can be found for clock type (Act,B). The definition of Act is then propagated at cycle 1, which is the smallest cycle at which Act is not known.

The fix-point of the filtering process is not fully described here, since it further alternate between propagation of equations for Act, Sel, Nxt and temporal rules until it deduces the existence of five previous cycles on the **base** clock.

4 Insights from the Compilation of State-Machines

The constraints defined in the previous section capture the whole semantics of state-machine constructs, as defined by the Scade 6 compilation process of Section 2. However, some specific rules may be added in order to optimise the resolution algorithm, at two levels: the clock hierarchy and the filtering rules.

Clock Hierarchy and State-machines

When applied to the state-machine translation, the clock hierarchy is used to represent the hierarchical structure of state machines. Each state is associated to two clock types whose carrier flows are the selected state (Sel) and the activated state (*Act*), and truth value is this state. An important point is that all these clocks appear at the same level of the clock hierarchy for a given state-machine. Moreover, the clock hierarchy contains as many levels as nested state-machines. The properties stated in Proposition 1 and 2 are thus widely used during derivations. We propose to strengthen these propositions in order to make a direct link between the transition function and the clock hierarchy. Given a state-machine (I, S, T), where I is the initial cycle, S the set of states and T the transition function that defines allowed transitions from one state to another, the following property states that the highest known cycle and status of the initial state I are the same as those of its ancestor clock.

Proposition 4.1 *Given a selected state identifier Sel, let* $cl_I = (Sel, v_I)$ *be the clock type corresponding to the initial state of the state machine and* cl^+ *its ancestor clock in* $\mathcal{H}$*. The following holds:* $(C_M(cl_I), S_M(cl_I)) = (C_M(cl^+), S_M(cl^+))$*.*

In the previous example, this proposition can be used to increment the C_M of clock type (Sel, A) while applying rule INIT-NC. The partial order relationship between the C_M of clock types can be extended when the reverse domain T^{-1} of a state v is a singleton state v': let $\{v'\} = T^{-1}(v)$; then $C_M(Sel, v') \geq C_M(Sel, v)$.

State Machine Constraint

The rules of section 3 do not take into account all the tight relationships between the three variables (*Sel*, *Act*, Nxt) created in the compilation process. These relationships mainly come from two major design rules: a single active state and a single transition per cycle. For instance, if the next state variable is instantiated to a given state v and the active state at the same cycle is a different state v', then the selected state is necessary equal to v' at this cycle, allowing some unification in $\mathcal{M}$. Indeed, the selected state cannot be set to a different state v'' since a transition would have already been fired between v'' and v', violating thus the second major rule. We therefore propose to add to the system a dedicated constraint called *sm* relating these three variables at a given cycle. This constraint is managed by the VAR rules: it is launched each time a new cycle C is created for a clock type based on the active or the selected state. In these rules X_C is a shortcut for $\mathcal{M}(X, C)$.

$$\frac{dom(Nxt_C) \cap dom(Act_C) = \emptyset \quad Sel_C \equiv Act_C}{sm(Nxt_C, Sel_C, Act_C)}$$

$$\frac{dom(Act_C) \cap dom(Sel_C) = \emptyset \quad Act_C \equiv Nxt_C}{sm(Nxt_C, Sel_C, Act_C)}$$

$$\frac{dom(Act_C) := dom(Act_C) \cap (dom(Nxt_C) \cup dom(Sel_C))}{sm(Nxt_C, Sel_C, Act_C)}$$

States			ON	OFF	STDBY
Disabled			F	T	F
En.	Interrupt		F	T	T
	Reg.	On	T	F	F
		StdBy	F	F	T

From	To	Condition
Dis.	En.	On
En.	Dis.	Off
Reg.	Int.	Brake
Int.	Reg.	Resume $\wedge\neg$ Brake
On	StdBy	Accel $\vee$ SpeedLimits
StdBy	On	$\neg$(Accel $\vee$ SpeedLimits)

Fig. 3. States, outputs and transitions for the *Cruise* example.

5 Case Study

Our Case Study is based on a comparison between three implementations of the same state machine. The first one, called *Cruise I*, implements a non-hierarchical version of this example with only boolean and single-clock temporal operators, following the previous Scade 5 compilation process for Scade SSM. The third one, *Cruise III*, uses the Scade 6 facilities for describing state machines, and the pre-compilation process of Section 2. The second one, *Cruise II*, is a code to code rewriting of the third one by replacing multi-clocked operators by a simulation using single-clock ones. All these versions have an equivalent behaviour when considering their outputs.

Presentation

The case study is a hierarchical version of a *Cruise Speed Controller*, which depends on five boolean inputs: *Brake* (brakes are pressed), *Accel* (accelerator is pressed), *Resume* (resume from interruption), *SpeedLimits* (*true* whenever speed is too slow or too fast), *On* and *Off* (start/stop requests). Our model consists in three nested state machines, each having two states: the controller can be either *Disabled* or *Enabled*; while enabled, the second automaton may be in states *Interrupt* or *Regulation*; a third automaton defines two computation modes when in state *Regulation*, namely *On* or *StandBy*. At each cycle, depending on its active state, the controller computes three boolean outputs: *ON*, *OFF* and *STDBY*, as shown in fig. 3. Following the hierarchical presentation, *Cruise II* and *Cruise III* versions have four possible states at each cycle. This leads to a search space of at least 4^{Max_C} possible sequences for Max_C cycles. On the other hand, the *Cruise I* version has six possible states at each cycle, searching in at least 6^{Max_C} possible sequences.

Coverage Objectives.

We propose to encode classical [17] coverage criteria in our setting by reachability objectives: (1) covering all the states, (2) covering all the transition-pairs, and (3) covering all the boolean outputs of the model, that is to obtain a sequence where *ON*, *OFF* and *STDBY* were at least one time set to true. Figure 4 shows the average time, over a hundred executions, needed by each implementation to generate a test sequence for a given objective with Max_C set to 4. The *Cruise III* implementation which uses all the features described in this paper is clearly the most efficient.

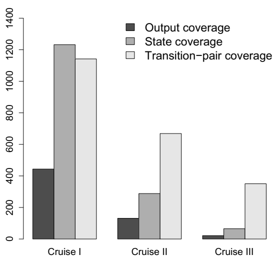


Fig. 4. Comparison of coverage resolution times (ms).

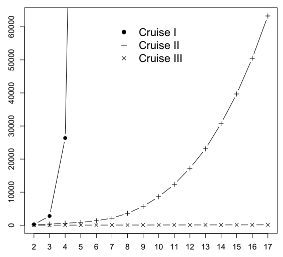


Fig. 5. Comparison of refutation times (ms) for increasing values of Max_C.

Safety Objectives.

The second kind of objective is to reach an unsafe state defined by an impossible combination of outputs. Since there is no solution in this case, no sequence is generated. The bound on the length of sequences, Max_C, is gradually increased to illustrate the behaviour of each implementation. Figure 5 shows the bad behaviour of the *Cruise I* version that would certainly be best handled by an adequate SMT solver. On the contrary, the response time for the *Cruise III* version is almost immediate and varies linearly with Max_C. In between, the response time for *Cruise II* version varies quadratically with Max_C. The main difference between *Cruise II* and *Cruise III* versions comes from the explicit notion of clock type within the filtering rules and the resolution algorithm. The guidelines provided by this clock typing dramatically prunes the search space by discarding states and transitions of inappropriate clock type.

6 Conclusion

We presented in this paper a CLP interpretation of the multi-clocked kernel of the latest version of the Scade 6 language. This interpretation uses the clock hierarchy as a global structure to perform powerful deductions. The case study illustrates that a simple encoding of the semantics of the multi-clocked operators does not provide such deduction ability. Moreover, as shown in Section 4, this interpretation can be improved with specific properties ensured by the Scade 6 compilation process. The extension of GATeL presented here is currently experimented on more complex case studies in the context of the SIESTA project. Previous single-clock version of GATeL provides basic mechanisms to assist the definition of used-defined test selection criteria [3]. These mechanisms must be extended to manage the multi-clock context. Furthermore, insights from the data-flow interpretation of state-machines could be used to ease the definition of state-machine based criteria. Further work concerns the handling of another kind of transitions, that is those which reset the behaviour of a state while entering in it. As stated in [8], reset conditions may be triggered asynchronously, which thus requires a full revision of our synchronous setting.

References

[1] *Defining and translating a "safe" subset of Simulink/Stateflow into lustre*, Technical Report TR-2004-16, Verimag Technical Report (2004).

[2] Bardin, S. and P. Herrmann, *Structural Testing of Executables*, in: *ICST*, 2008, pp. 22–31.

[3] Blanc, B. and B. Marre, *Test Selection Strategies for Lustre Descriptions in GATeL*, ENTCS **111** (2005), pp. 93–111.

[4] Bouali, A. and B. Dion, *Formal verification for model-based development*, in: *Society of Automotive Engineers, 400 Commonwealth Dr, Warrendale, PA, 15096, USA*, 2005.

[5] Bouquet, F., B. Legeard and F. Peureux, *CLPS-B: A constraint solver to animate a B specification*, STTT **6** (2004), pp. 143–157.

[6] Caspi, P., A. Curic, A. Maignan, C. Sofronis, S. Tripakis and P. Niebert, *From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications*, ACM New York, NY, USA, 2003.

[7] Caspi, P., D. Pilaud, N. Halbwachs and J. Plaice, *Lustre: A Declarative Language for Programming Synchronous Systems*, in: *POPL*, 1987, pp. 178–188.

[8] Colaço, J.-L., B. Pagano and M. Pouzet, *A Conservative Extension of Synchronous Data-flow with State Machines*, in: *EMSOFT*, 2005, pp. 173–182.

[9] Colaço, J.-L. and M. Pouzet, *Type-based Initialization Analysis of a Synchronous Data-flow Language*, STTT **6** (2004), pp. 245–255.

[10] Colaco, J.-L. and M. Pouzet, *Clocks as First Class Abstract Types*, in: *EMSOFT*, 2003, pp. 134–155.

[11] Collavizza, H., M. Rueher and P. van Hentenryck, *CPBPV: A Constraint-Programming Framework for Bounded Program Verification*, in: *CP*, 2008, pp. 327–341.

[12] Dormoy, F.-X., *SCADE 6: A Model Based Solution For Safety Critical Software Development*, in: *ERTS*, 2008.

[13] Gotlieb, A., B. Botella and M. Rueher, *Automatic Test Data Generation Using Constraint Solving Techniques*, in: *ISSTA*, 1998, pp. 53–62.

[14] Jaffar, J. and J.-L. Lassez, *Constraint logic programming*, in: *POPL*, 1987, pp. 111–119.

[15] Maraninchi, F. and Y. Rémond, *Mode-automata: About modes and states for reactive systems*, in: *ESOP*, 1998.

[16] Marre, B. and A. Arnould, *Test Sequences Generation from Lustre Descriptions: GATeL*, in: *ASE*, 2000, pp. 229–239.

[17] Offutt, J., S. Liu, A. Abdurazik and P. Ammann, *Generating test data from state-based specifications*, JSTVR **13** (2003), pp. 25–53.

[18] Pace, G., N. Halbwachs and P. Raymond, *Counter-example generation in symbolic abstract model-checking*, International Journal on Software Tools for Technology Transfer (STTT) **5** (2004), pp. 158–164.

[19] Sen, K., D. Marinov and G. Agha, *CUTE: a concolic unit testing engine for C*, in: *ESEC/SIGSOFT FSE*, 2005, pp. 263–272.

[20] Sheeran, M., S. Singh and G. Stålmarck, *Checking safety properties using induction and a SAT-solver*, Lecture Notes in Computer Science (2000), pp. 108–125.

[21] Williams, N., B. Marre, P. Mouy and M. Roger, *Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis*, in: *EDCC*, 2005, pp. 281–292.