Verification of embbeded systems with preemption: a negative result

Jérôme Ermont*, Frédéric Boniol*

*IRIT-ENSEEIHT, 2 rue C. Camichel. F31071 Toulouse, France
{frederic.boniol,jerome.ermont}@enseeiht.fr

Abstract. The aim of this article is to explore the problem of verification of preemptive communicating timed processes, i.e., timed processes which can be suspended and resumed by an on-line scheduler. The contribution of the article is to show that this problem is unfortunately undecidable. We discuss then an alternative verification method to overcome this negative result.

1 Introduction

Embedded systems often are characterized by the two following properties. Firstly they are an information processing sub system of their embedding systems. Secondly they are reactive, i.e. they interact with their physical environment at a speed imposed by the environment. Consequently, they have to meet both real time and safety constraints. These characteristics make writing embedded software a substantially different and more difficult task than classical software. To overcome this complexity, the avionics architectures traditionally being implemented are of federated, which means that each avionics system has its own independent and dedicated computing and communicating resources. Federated architectures have great advantage of inherent fault containment. Systems implemented by dedicated resources are loosely coupled allowing modular design and verification.

However, federated architectures are penalizing due to massive use of isolated resources, and results in increase in weight, maintenance costs, power consumption, etc. Due to these drawbacks, the aviation industry is gradually moving towards the use of Integrated Modular Architectures (IMA) for both civil and military aircraft programmes. Instead of using individual resources, IMA uses generic computing and communicating platforms. This allows multiple applications to share and reuse the same computing and communicating resources concurrently. This facilitates a reduction in the number of deployed subsystems which are not fully utilised and provides a more efficient use of system resources, leaving space for future expansion. This is the case of modern aircraft such as Airbus A380, Boeing B777, or Euro Fighter aircraft. Each function is allocated to a shared computer. Scheduling of functions is managed by an embedded real time operating system. Communications between computers are supported by multiplexed data buses (Boeing B777), or by switched communication networks (Airbus 1380), and are scheduled by real time communication protocols. The main advantage of such an organisation is to offer a modular view of the global system. However, the use of shared computing and communication resources introduces non-deterministic jitters and delays, which can affect the global behaviour of the system.

Due to safety constraints, embedded systems have to go through certification. It requires a rigorous design process based on tight rules. However, due to their increasing complexity, there is no guarantee that such a design process leads to error free systems. Another way for helping embedded system designers is *formal methods*, i.e., fundamental techniques for analysis, validation, or transformation of systems in a provably sound way. Real time systems correctness depends both, on qualitative (functional) aspects and, as said before, on quantitative (performance) ones (scheduling, delay, jitters...). Such a dependency becomes more important with IMA systems where concurrency between several functions on the same resources can lead to delays which can alterate the functional behaviour. Consequently, to verify an IMA system needs to to take into account the real time behaviour of its components, and then the implementation choices: the allocation of processes on computers, the messages allocation on communication buses, and more precisely the real time scheduling strategy of processes on computers and messages on buses.

The questions are then: (a) how to model such a system and (b) is it possible to formally verify timed properties over such a system. Unfortunately, the answer to the second question depends on the process scheduling strategy. As far as non preemptive strategy is concerned (processes can be aborted but not resumed), it is easy to show that the verification is still decidable. This result is obtained by translating such processes into timed automata. However, as far as preemptive strategy is concerned (processes can be suspended and resumed), the verification problem becomes undecidable. The aim of this article is to prove this last negative result. It can be obtained by reducing the verification problem of preemptive processes to the halting problem for two-counters machines (Henzinger et al., 1998; Cerans, 1992). For that purpose, we introduce in the next section a formal algebra of preemptive timed processes in order to model real time systems. We then give in section 3 a tiny logic $Sub\mathbb{SL}$ for modeling properties to be verified. And finally, section 4 proves that the satisfaction problem of a formula ϕ from $Sub\mathbb{SL}$ by a process p from $\mathcal{P}$ is undecidable.

2 Step one: a (small) calculus for modeling real time preemptive systems

Embedded systems are real-time systems composed of processes which evolve and communicate using events (or actions). In order to formalize such systems, we define in this section a timed process algebra, called $\mathcal{P}$. The time domain of $\mathcal{P}$ is $\mathcal{T} = \mathbb{R} \cup \{\infty\}$. Let us consider $\mathcal{A} = \{a, b, \ldots\}$ the set of events. The co-set $\overline{\mathcal{A}} = \{\overline{a} | a \in \mathcal{A}\}$ is the set of waiting actions. Let $\varepsilon \notin \mathcal{A} \cup \overline{\mathcal{A}}$ be an invisible action used to "break" delays. The following grammar defines the terms of $\mathcal{P}$, ranged over P, Q, ...

$$P ::= \delta | X | aP | [l,u]P | P + Q | P \| Q | (P \Downarrow \overline{a}) Q | (P \downarrow \overline{a} \uparrow \overline{b}) Q | recX.P$$

where:

- δ is an idle process which cannot execute any action but let the time pass.
- $\mathcal{V} = \{X, Y, ...\}$ is a set of process variables used to model recursive processes.
- aP is a non-blocking broadcasting process. It immediately executes action a and then behaves like P.

- $[l,u]P$ is a delayed process, with $l,u \in \mathcal{T}$ and $l \leq u$. When $u > 0$, this process may stay idle $t \leq u$ time units. When $l = 0$, it performs the ε action and behaves like P. If $l > 0$, it must stay idle at least l time units before performing the ε action.
- $P + Q$ is the non deterministic choice between P and Q.
- $P \| Q$ executes P and Q in parallel. It (1) can execute independently actions of P and Q, or (2) can synchronize whenever one of them sends a transmission event and the other waits for the complementary reception event, or (3) can let the time pass if P and Q are able to do so.
- $(P \Downarrow \overline{a})Q$ models the abortion mechanism. Before receiving event a, this process behaves like P. When receiving a, P is aborted and Q is started.
- $(P \downarrow \overline{a} \uparrow \overline{b})Q$ models the suspension mechanism. Before receiving event a, this process behaves like P. When a is received P is suspended and Q is resumed from its last suspension point. Afterwards, the global process behaves like $(Q \downarrow \overline{b} \uparrow \overline{a})P$.
- $recX.P$ is the classical recursive process modelling infinite behaviours.

The $\mathcal{P}$ semantics is given by the timed labeled transition system $(\mathcal{P}, P, \rightarrow)$ where $\rightarrow \subseteq \mathcal{P} \times \mathcal{A} \cup \mathcal{T} \cup \{\varepsilon\} \times \mathcal{P}$, defined by table 1. We write $P \xrightarrow{t} P'$ with $t \in \mathcal{T}$ to mean that P lets t time units pass and becomes P'. In the same way, we write $P \xrightarrow{a} P'$ to mean that P emits event a and instantaneously becomes P'. To specify non-blocking transmission event, reception event and broadcasting, we also add an auxiliary relation $\rightsquigarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ and we write $P \overset{a}{\rightsquigarrow} P'$ to mean that P is waiting for the event a and when receiving it, behaves like P'.

Let us note $\overline{a}P = (\delta \Downarrow \overline{a})P$. $\overline{a}P$ is a blocking process waiting for the broadcasting of action a (by another process), and when receiving it, behaves like P.

As we are only interested in observational behaviour, we introduce then a weak behavioural transition $\Rightarrow$ on $\mathcal{P}$ processes which abstracts invisible action ε.

Definition 1 *Abstraction of ε. Let $\Rightarrow \subseteq \mathcal{P} \times \mathcal{A} \cup \mathcal{T} \times \mathcal{P}$ defined by:*

- $P \overset{a}{\Rightarrow} P'$ *if* $P(\xrightarrow{\varepsilon})^* \xrightarrow{a} P'$
- $P \overset{t}{\Rightarrow} P'$ *if* $P(\xrightarrow{\varepsilon})^* \xrightarrow{t_1} (\xrightarrow{\varepsilon})^* \cdots (\xrightarrow{\varepsilon})^* \xrightarrow{t_n} (\xrightarrow{\varepsilon})^* P'$ *with* $t = \sum_{i \leq n} t_i$

3 Step two: a (tiny) logic

To prove that the verification problem is undecidable, a property language is required. The language we consider, called $Sub\mathbb{SL}$, is a fragment of a dense-time logic denoted $\mathbb{SL}$ presented in Laroussinie et al. (1995) (which in turn is a variation of the timed mu-calculus (Henzinger et al., 1992)). This logic (and then the fragment under consideration) allows model-checking by using accessibility analysis which is decidable for timed automata (Aceto et al., 2001).

Definition 2 *$Sub\mathbb{SL}$. The set of $Sub\mathbb{SL}$ formulae is generated by the following grammar:*

$$\varphi ::= tt \mid ff \mid \varphi_1 \wedge \varphi_2 \mid [a]\varphi \mid \mathbb{\forall}\varphi \mid X \mid \max(X, \varphi)$$

where $a \in \mathcal{A}$. X is a formula variable and $\max(X, \varphi)$ stands for the maximal solution of the recursion equation $X = \varphi$.

$$t \in \mathcal{T} \frac{}{\delta \xrightarrow{t} \delta} \qquad a \in \mathcal{A} \frac{}{aP \xrightarrow{a} P} \qquad t \in \mathcal{T} \frac{0<t\leq d_2}{[d_1,d_2]P \xrightarrow{t} [\max(0,d_1-t),d_2-t]P} \qquad \frac{}{[0,d_2]P \xrightarrow{\varepsilon} P}$$

$$\alpha \in \mathcal{A} \cup \{\varepsilon\} \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \alpha \in \mathcal{A} \cup \{\varepsilon\} \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'} \qquad a \in \mathcal{A} \frac{P \overset{a}{\rightsquigarrow} P'}{P+Q \overset{a}{\rightsquigarrow} P'} \qquad a \in \mathcal{A} \frac{Q \overset{a}{\rightsquigarrow} Q'}{P+Q \overset{a}{\rightsquigarrow} Q'}$$

$$t \in \mathcal{T} \frac{P \xrightarrow{t} P' \quad Q \overset{t}{\not\rightarrow}}{P+Q \xrightarrow{t} P'} \qquad t \in \mathcal{T} \frac{P \overset{t}{\not\rightarrow} \quad Q \xrightarrow{t} Q'}{P+Q \xrightarrow{t} Q'} \qquad t \in \mathcal{T} \frac{P \xrightarrow{t} P' \quad Q \xrightarrow{t} Q'}{P+Q \xrightarrow{t} P'+Q'}$$

$$a \in \mathcal{A} \frac{P \xrightarrow{a} P' \quad Q \overset{a}{\not\rightarrow}}{P \| Q \xrightarrow{a} P' \| Q} \qquad a \in \mathcal{A} \frac{P \overset{a}{\not\rightarrow} \quad Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P \| Q'}$$

$$a \in \mathcal{A} \frac{P \xrightarrow{a} P' \quad Q \overset{a}{\rightsquigarrow} Q'}{P \| Q \xrightarrow{a} P' \| Q'} \qquad a \in \mathcal{A} \frac{P \overset{a}{\rightsquigarrow} P' \quad Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P' \| Q'}$$

$$a \in \mathcal{A} \frac{P \overset{a}{\rightsquigarrow} P' \quad Q \overset{a}{\not\rightarrow} \quad Q \overset{a}{\not\rightarrow}}{P \| Q \overset{a}{\rightsquigarrow} P' \| Q} \qquad a \in \mathcal{A} \frac{P \overset{a}{\not\rightarrow} \quad P \overset{a}{\not\rightarrow} \quad Q \overset{a}{\rightsquigarrow} Q'}{P \| Q \overset{a}{\rightsquigarrow} P \| Q'}$$

$$a \in \mathcal{A} \frac{P \overset{a}{\rightsquigarrow} P' \quad Q \overset{a}{\rightsquigarrow} Q'}{P \| Q \overset{a}{\rightsquigarrow} P' \| Q'} \qquad t \in \mathcal{T} \frac{P \xrightarrow{t} P' \quad Q \xrightarrow{t} Q'}{P \| Q \xrightarrow{t} P' \| Q'}$$

$$a \in \mathcal{A} \frac{}{(P \Downarrow \overline{a}) Q \overset{a}{\rightsquigarrow} Q} \qquad b \in \mathcal{A} \frac{P \overset{b}{\rightsquigarrow} P' \quad b \neq a}{(P \Downarrow \overline{a}) Q \overset{b}{\rightsquigarrow} (P' \Downarrow \overline{a}) Q}$$

$$\alpha \in \mathcal{A} \cup \{\varepsilon\} \frac{P \xrightarrow{\alpha} P'}{(P \Downarrow \overline{a}) Q \xrightarrow{\alpha} (P' \Downarrow \overline{a}) Q} \qquad t \in \mathcal{T} \frac{P \xrightarrow{t} P'}{(P \Downarrow \overline{a}) Q \xrightarrow{t} (P' \Downarrow \overline{a}) Q}$$

$$c \in \mathcal{A} \frac{P \xrightarrow{c} P'}{(P \downarrow \overline{a} \uparrow \overline{b}) Q \xrightarrow{c} (P' \downarrow \overline{a} \uparrow \overline{b}) Q} \qquad c \in \mathcal{A} \frac{P \overset{c}{\rightsquigarrow} P' \quad c \neq a \quad c \neq b}{(P \downarrow \overline{a} \uparrow \overline{b}) Q \overset{c}{\rightsquigarrow} (P' \downarrow \overline{a} \uparrow \overline{b}) Q}$$

$$a \in \mathcal{A} \frac{}{(P \downarrow \overline{a} \uparrow \overline{b}) Q \overset{a}{\rightsquigarrow} (Q \downarrow \overline{b} \uparrow \overline{a}) P} \qquad t \in \mathcal{T} \frac{P \xrightarrow{t} P'}{(P \downarrow \overline{a} \uparrow \overline{b}) Q \xrightarrow{t} (P' \downarrow \overline{a} \uparrow \overline{b}) Q}$$

$$\alpha \in \mathcal{A} \cup \{\varepsilon\} \frac{P \xrightarrow{\alpha} P'}{recX.P \xrightarrow{\alpha} P'\{recX.P/X\}} \qquad a \in \mathcal{A} \frac{P \overset{a}{\rightsquigarrow} P'}{recX.P \overset{a}{\rightsquigarrow} P'\{recX.P/X\}}$$

$$t \in \mathcal{T} \frac{P \xrightarrow{t} P'}{recX.P \xrightarrow{t} P'\{recX.P/X\}}$$

TAB. 1 – *Semantics rules of* $\mathcal{P}$

The satisfaction relation of $Sub\mathbb{SL}$ formulae over $\mathcal{P}$ processes is defined by:

Definition 3 Satisfaction. *The satisfaction relation $\models$ between $\mathcal{P}$ processes, $Sub\mathbb{SL}$ formulae is the largest relation satisfying the following implications:*

$$
\begin{array}{lcl}
P \models tt & \Rightarrow & \textit{true} \\
P \models ff & \Rightarrow & \textit{false} \\
P \models \varphi_1 \wedge \varphi_2 & \Rightarrow & \forall P' : P(\xrightarrow{\varepsilon})^* P' \textit{ then } P' \models \varphi_1 \textit{ and } P' \models \varphi_2 \\
P \models [a]\varphi & \Rightarrow & \forall P' \textit{ and } \forall b \neq a : P(\stackrel{b}{\Rightarrow})^* \stackrel{a}{\Rightarrow} P' \textit{ then } P' \models \varphi \\
P \models \mathbb{W}\varphi & \Rightarrow & \forall t \in \mathcal{T}, \forall P' : P \stackrel{t}{\Rightarrow} P' \textit{ then } P' \models \varphi \\
P \models \max(X, \varphi) & \Rightarrow & \forall P' : P(\xrightarrow{\varepsilon})^* P' \textit{ then } P' \models \varphi\{\max(X, \varphi) \backslash X\}
\end{array}
$$

Any relation satisfying these implications is a satisfaction relation. It follows from standard fix-point theory that $\models$ is the union of all satisfaction relations and that these implications are in fact bi-implications.

A high-level operator has been defined in terms of the basic elements of $Sub\mathbb{SL}$, and is, for the designer, closer to intuition than the basic operators: always $\varphi = \max(X, \varphi \wedge \bigwedge_{a \in \mathcal{A}} [a]X \wedge \mathbb{W}X)$. Note that the formula $[a]$tt means that a must be broadcast instantaneously. Conversely, $[a]$ff is satisfied if and only if a cannot be broadcast instantaneously.

4 Step three: the undecidability result

After giving a timed preemptive process algebra and a (tiny) logic, we prove in this main section that the satisfaction problem of a $Sub\mathbb{SL}$ formula by a $\mathcal{P}$ process is undecidable. To do this, we use the halting problem for a two-counters machine.

Definition 4 Two-counters machine. *A two-counters machine, with non-negative integer variables called counters denoted by C_1 and C_2, is a finite sequence of instructions 1 : INS$_1, \cdots, k$: INS$_k$ where $k \in \mathbb{N}$, and all instructions INS$_i (0 \leq i < k)$ has one of the next forms (with $1 \leq l, l', l'' \leq k$ and $1 \leq j \leq 2$):*

- $C_j := C_j + 1$*; goto* l
- *if* $C_j = 0$ *then goto* l' *else* ($C_j := C_j - 1$*; goto* l'')

Minsky (1967) has shown that the halting problem of a two-counters machine is undecidable. The main proposition leading to our undecidability result is then:

Proposition 1 *The halting problem of a two-counters machine can be reduced to the satisfaction problem of a $Sub\mathbb{SL}$ property by a $\mathcal{P}$ process.*

To prove this proposition, we model the behaviour of a two-counters machine in the processes algebra $\mathcal{P}$.

4.1 Step 3.1: modelling counter values

Counters values can be encoded using implicit clock values (intervals) of the algebra. This encoding is defined by the equation: $c = 2^{1-C}$ where c is an implicit clock of the algebra and C is a counter value. In other words, the value C of a counter of the machine corresponds to the value 2^{1-C} of a clock of the algebra. In this way, counters values belonging to $\mathbb{R}$ are encoded into the clock values in [0,2]. The discrete evolutions of the counters are simulated by a set of continuous rounds taking W (or multiples of W) time units. This technique, named "wrapping", is inspired by K. Cerans (Cerans, 1992) and T. Henzinger *et al.* (Henzinger et al., 1998). Clocks which model counters have cyclic and continuous evolutions from 0 to W. To make understanding more easier, intervals, modelling time evolution in the algebra, are labelled by clock names. Thus, the interval labelled c_1 will encode the counter value C_1 and the interval labelled c_2 will encode the counter value C_2. When W time units have passed, the counters are reseted and an event reset_{c_j} ($j = 1, 2$) is sent to indicate this reset to other processes. The counters reset can also be "forced" when receiving an event $freset_{c_j}$ ($j = 1, 2$). The $\mathcal{P}$ models of the two counters C_1 and C_2 are :

$$Ct_1 = recX.((recY.[W, W]_{c_1}\text{reset}_{c_1} Y) \Downarrow \overline{freset_{c_1}})X$$
$$Ct_2 = recX.((recY.[W, W]_{c_2}\text{reset}_{c_2} Y) \Downarrow \overline{freset_{c_2}})X$$

A two-counters machine consists of three operations: (1) incrementing a counter, (2) decrementing a counter and (3) branching based upon whether a specific counter values are 0. We now model this three operations using $\mathcal{P}$ processes.

4.2 Step 3.2: modelling the test for zero

As we say before, the value C of a counter corresponds to an implicit clock value in the algebra. Testing if C is equal to 0 consists of determining if a clock value is equal to 2. Using the wrapping technique, the clock c_1 (or c_2) will evolve during a round (i.e. W time units). At the end of a round, the value of c_1 stays unchanged.

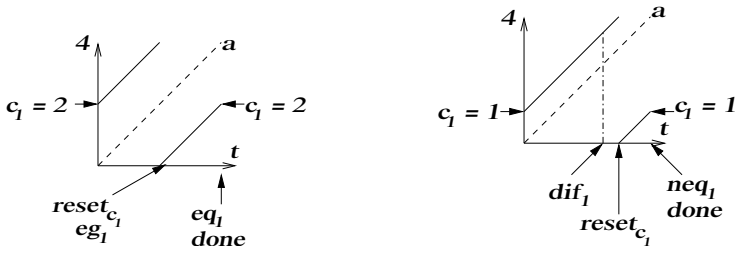
Suppose that the value of C_1 is x. The process Ct_1 will send the event $reset_{c_1}$ after waiting $W - x$ time units. The wait time is minimal when x is equal to 2 and will grow when x decrease (corresponding to incrementing a counter). When $C_1 = 1$, the value of clock c_1 is $x = 1$, according to the encoding equation. Then, to detect if specific counter values are 0, we wait during $W - 2 + \frac{1}{2}$ time units. In this way, if the process Ct_1 sends the event $reset_{c_1}$ after letting pass $W - 2 + \frac{1}{2}$ time units, then $x < 2$ and $C_1 \neq 0$. Conversely, if the process Ct_1 sends the event $reset_{c_1}$ before $W - 2 + \frac{1}{2}$ time units, then the only value for x is 2 and $C_1 = 0$. The test for zero of counter C_2 can be done in the same way.

To show if a counter is equal to zero or not, we use two events: eq_1 and neq_1. These two events will be sent at the end of the test, i.e. at the end of round. We use also two auxiliary events: $zero_1$ and dif_1. The process $ZERO_1$ models the previously described behaviour.

$$ZERO_1 = \overline{test_1}((\overline{reset_{c_1}}\ zero_1\ \delta \Downarrow \overline{dif_1})\delta \| ([W - 2 + \frac{1}{2}, W - 2 + \frac{1}{2}]dif_1\ \delta \Downarrow \overline{zero_1})\delta)$$

A process A_1 is used to count the number of rounds and sends an event $done$ at the end of the test.

$$A_1 = recX.\overline{test_1}[W, W]_a t_{11}\ \overline{t_{12}}\ done\ X$$



(a) $c_1 = 2$: counter C_1 is equal to 0 (b) $c_1 = 1$, counter C_1 is not equal to 0

FIG. 1 – *Examples of test for zero with* $W = 4$.

A third process T_1 receives $zero_1$ and dif_1 and emits eq_1 and neq_1 at the end of the test, i.e. after receiving event t_{11} from A_1.

$$T_1 = \overline{test_1}(\overline{zero_1}\ \overline{t_{11}}\ eq_1\ t_{12}\delta + \overline{dif_1}\ \overline{t_{11}}\ neq_1\ t_{12}\delta)$$

Finally, the $\mathcal{P}$ process testing for zero for the counter C_1 is done by composing the previous different processes.

$$TEST_1 = (recX.((ZERO_1 \| T_1) \Downarrow \overline{done})X) \| A_1$$

On figures 1(a) and 1(b), we show two executions of the process $TEST_1$ with $W = 4$. In the first case $c_1 = 2$ and in the second case $c_1 = 1$. In the same manner, we can model the test for zero for the counter C_2. Thus, we obtain the $TEST_2$ process.

4.3 Step 3.3: decrementing a counter

Decrementing a counter consists of synchronizing this counter with a process which executes a delay and which can be suspended. Suppose x the value of c_1. As $x = 2^{1-C_1}$, decrementing the counter C_1 consists of doubling the value of x. Let the process Z_{dec1} be synchronized with the process Ct_1 and which can be suspended. Whenever Z_{dec1} is suspended, time cannot evolve in this process. Let z be a clock modeling time evolution in Z_{dec_1}. The decrementing models consists of three step: (1) assigning the value of x to z; (2) doubling the value of z; then $z = 2 \times x$; and (3) assigning the value of z to c_1; finally $c_1 = 2 \times x$ and $C_1 = C_1 - 1$.

The first step, assigning the value of x to z, consists of letting a round pass (i.e. W times units) and resetting z when x is reseted, by synchronizing Ct_1 and Z_{dec_1} on event $reset_{c_1}$. Then, before the doubling procedure, the value of z is equal to x. Let a be a clock reseted at this time. Suppose that z is frozen, by suspending the process Z_{dec_1}, during $W - x$ time units, i.e. up to receiving the event $reset_{c_1}$. At the end of this duration, z is still equal to x, but $a = W - x$. Suppose that z is restarted by resuming process Z_{dec_1}. When $a = W$, i.e. x time units later, z is equal to $x + x$. z is then doubled. Finally (third step), z is assigned to c_1 by synchronizing Ct_1 and Z_{dec_1} on event $freset_{c_1}$ when z is reseted. c_1 is doubled and the counter has been decremented: $C_1 := C_1 - 1$.

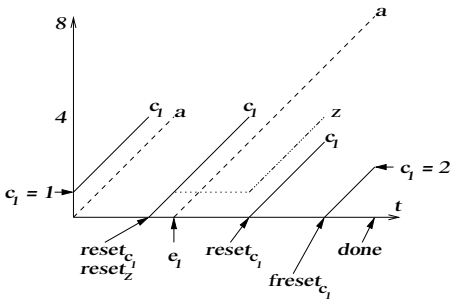


FIG. 2 – *Decrementing the counter C_1 when $W = 4$, initially $c_1 = 1$*

As for the test for zero, a process A_{dec_1} is used to control the evolution of the decrementing. At the end of the first round, the process Z_{dec_1} is suspended via the event e_1 transmitted by A_{dec_1}. The process A_{dec_1} lets the time to pass during the doubling and the second assignment procedures, i.e. during 2 rounds, and, afterwards, transmits the event $done$ to indicate the end of the decrementing.

$$A_{dec_1} = recX.\overline{dec_1}[W,W]_a e_1[2W,2W]_a doneX$$

The affectation $z := c_1$ is made by the process RZ_{dec_1}. To do so, this process synchronizes Z_{dec_1} and Ct_1 when $c_1 = 0$.

$$RZ_{dec_1} = \overline{dec_1}\ \overline{reset_{c_1}} reset_z \delta$$

A third process Z_{dec_1} controls the evolution of z. The execution of this process is suspended when receiving event e_1 and resumed when receiving event $reset_{c_1}$. Thus, evolution of z is suspended during $W - x$ time units, where x is initial value of c_1.

$$Z_{dec_1} = \overline{dec_1}((recX.(([W,W]_z freset_{c_1}\delta) \Downarrow \overline{reset_z})X) \downarrow \overline{e_1} \uparrow \overline{reset_{c_1}})\delta$$

Finally, the process DEC_1 controls the decrementing of the counter C_1.

$$DEC_1 = (recX.((RZ_{dec_1} \| Z_{dec_1}) \Downarrow \overline{done})X) \| A_{dec_1}$$

The event dec_1 is used to control the beginning of the decrementing and the event $done$ is used to indicate the end. An example is given by figure 2, with $W = 4$ and $c_1 = 1$, i.e. $C_1 = 1$. After transmission of event $done$, the value of c_1 is 2 and the counter C_1 is decremented: $C_1 = 0$. In the same way, decrementing the counter C_2 can be modelled by a similar process DEC_2.

4.4 Step 3.4: incrementing a counter

To model incrementing a counter is quite more difficult. If decrementing a counter consists of doubling a clock value, conversely incrementing a counter consists of halving this value. This halving procedure is done by using two auxiliary clocks c_{aux1} and c_{aux2}. We proceed in

four steps: (1) non-deterministic guessing a value to c_{aux1} and assigning c_{aux1} to c_{aux2}; (2) doubling the value of c_{aux1}: $c_{aux1} := 2 \times c_{aux1}$; (3) testing if $c_{aux1} = c_1$; in that case, we proceed to step 4, else we restart from step 1; and (4) assigning the value of c_{aux2} to c_1. This sequence is encoded by the process A_{inc_1}:

$$\begin{array}{rcl} A_{inc_1} & = & recX.\overline{inc_1}[W,W]_a \; doubl_1[W,W]_a e_1[2W,2W]_a \\ & & testeq_1[W,W]_a \; endtest_1[W,W]_a \; end_1 \; X \end{array}$$

Guessing a value to c_{aux1} consists of letting the time to pass during a non-deterministic delay between $[2, W]$, such that at the end of a round, the value of $c_{aux1} \in [0, 2]$. The process $Caux_1$ controls the evolution of the clock c_{aux1}. Initially, c_{aux1} evolves from 0 to a non-deterministic value in [2,W]. Then, the behaviour of $Caux_1$ is similar to the process Ct_1: When W time units have passed, c_{aux1} is reseted and an event $reset_{caux_1}$ is sent to indicate this reset. c_{aux1} can also be reseted when receiving an event $freset_{caux_1}$.

$$\begin{array}{rcl} Caux_1 & = & \overline{inc_1}[2,W]initc_2 \\ & & (recX.((recY.[W,W]_{c_{aux1}} reset_{caux_1} Y) \Downarrow \overline{freset_{caux_1}})X) \end{array}$$

The process $Caux_2$ is used to store the initial value of c_{aux1} in clock c_{aux2} to allow the final assignment. To do so, the clock c_{aux2} is initialized when receiving event $initc_2$ provided by $Caux_1$ and then evolves periodically from 0 to W.

$$Caux_2 = \overline{inc_1}\ \overline{initc_2} recX.[W,W]_{c_{aux2}} reset_{caux_2} X$$

As done for decrementing a counter, doubling consists of freezing a clock evolution during $W - c_{aux1}$ time units by synchronizing processes $Caux_1$ and Z_{inc_1}:

$$Z_{inc_1} = \overline{doubl_1}((recX.(([W,W]_z freset_{caux_1}\delta) \Downarrow \overline{reset_z})X) \downarrow \overline{e_1} \uparrow \overline{reset_{caux_1}})\delta$$

Before doubling, we have to synchronize clocks of processes $Caux_1$ and Z_{inc_1}. This is done using process RZ_{inc_1} and events $reset_{caux_1}$ and $reset_z$.

$$RZ_{inc_1} = \overline{inc_1}\ \overline{reset_{caux_1}} reset_z \delta$$

The third step (test) is modeled by the process EQ_{inc_1}. $caux_1$ and c_1 are equal if and only if the events $reset_{caux_1}$ and $reset_{c_1}$ are received by EQ_{inc_1} simultaneously.

$$\begin{array}{rcl} EQ_{inc_1} & = & (recX.\overline{testeq_1} \\ & & (\overline{reset_{caux_1}}\ \overline{reset_{c_1}} eq\ \delta + \overline{reset_{c_1}}\ \overline{reset_{caux_1}} eq\ \delta \Downarrow \overline{fail})neq\ X) \\ & & \|(recX.\overline{testeq_1} fail\ X) \end{array}$$

Remark 1 *Due to the non-determinism of the parallel operator, the event $fail$ can be produced before receiving $reset_{caux_1}$ and $reset_{c_1}$. In such a case, EQ_{inc_1} can emit event neq before eq although $c_{aux1} = c_1$. However, whenever $c_{aux1} = c_1$, it is also possible to receive $reset_{caux_1}$ and $reset_{c_1}$ before emitting $fail$, and then, to produce eq before neq. As the halting problem for a two-counters machine is encoded into a reachability problem (see section 4.6), the existence of such an execution is sufficient.*

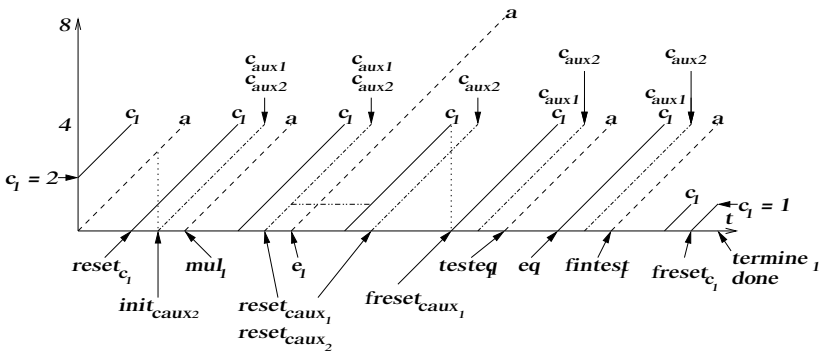


FIG. 3 – *Incrementing the counter C_1 when $W = 4$, initially $c_1 = 2$*

A process $Control_1$ determines if the halving procedure needs to be restarted (when the test failed) or, inversely, to assign c_{aux2} to c_1 by synchronizing C_{aux2} and Ct_1.

$$\begin{aligned} Control_1 \quad = \quad & recX.(\overline{eq}\ \overline{endtest_1}\ \overline{reset_{caux_2}} freset_{c_1} \overline{end_1}\ done\ X \\ & + \overline{neq}\ \overline{end_1}\ inc_1\ X) \end{aligned}$$

Finally, incrementing a counter C_1 can be model by the process INC_1.

$$\begin{aligned} INC_1 \quad = \quad & (recX.((Caux_1 \| Caux_2 \| Z_{inc_1} \| RZ_{inc_1}) \Downarrow \overline{done})X) \\ & \| A_{inc_1} \| EQ_{inc_1} \| Control_1 \end{aligned}$$

Remark 2 *As the choice of values c_{aux1} and c_{aux2} is non-deterministic, the execution time of process INC_1 can be unbounded. Nevertheless, the end of the process (transmission of the event done) stays inevitable at any time. In other words, incrementing a counter can terminate always despite his unbounded execution time.*

On figure 3 is depicted an example of incrementing the counter C_1 with $W = 4$ and $c_1 = 2$. In the same way, incrementing the counter C_2 is modelled by a similar process INC_2.

4.5 Global model of a two-counters machine

The two counters can only evolve if an operation is executed. We define two processes OP_1 and OP_2 which control processes Ct_1 and Ct_2 when an operation is requested.

$$\begin{aligned} OP_1 &= recX.(\overline{inc_1} op_1\ X + \overline{dec_1} op_1\ X + \overline{test_1} op_1\ X) \\ OP_2 &= recX.(\overline{inc_2} op_2\ X + \overline{dec_2} op_2\ X + \overline{test_2} op_2\ X) \end{aligned}$$

Whenever an operation is requested, process OP_i ($i = 1$ or 2) transmits event op_i. Ct_i must be resumed up to the end of the operation (reception of event $done$). Then evolution of counters and their associated operations can be model in $\mathcal{P}$ by the processes COUNTER$_1$ and COUNTER$_2$.

$$\begin{aligned} \text{COUNTER}_1 &= (\delta \downarrow \overline{op_1} \uparrow \overline{done}) Ct_1 \| INC_1 \| DEC_1 \| TEST_1 \| OP_1 \\ \text{COUNTER}_2 &= (\delta \downarrow \overline{op_2} \uparrow \overline{done}) Ct_2 \| INC_2 \| DEC_2 \| TEST_2 \| OP_2 \end{aligned}$$

A two-counters machine is a finite sequence of instructions. It remains to model in $\mathcal{P}$ such a sequence. Let us consider for example the following two-counters machine:

INS$_1$: $C_1 := C_1 + 1$; goto INS$_2$
INS$_2$: if $C_1 = 0$ then goto INS$_3$ else ($C_1 := C_1 - 1$; goto INS$_1$)
INS$_3$: if $C_2 = 0$ then goto INS$_4$ else ($C_2 := C_2 - 1$; goto INS$_2$)
INS$_4$: halt

This sequence can be encoded by a $\mathcal{P}$ process which transmits an event inc_1 when the counter C_1 must be incremented, an event dec_1 when the same counter must be decremented, or an event $test_1$ when the test $C_1 = 0$ must be performed. Executing the instruction *halt* corresponds to execute the process $halt\delta$ which transmits the event $halt$ and idles. In $\mathcal{P}$, the previous sequence is modelled by the process P_INS$_1$:

$$\begin{array}{lcl} \text{P\_INS}_1 & = & recX_1.inc_1\ \overline{done}\ \text{P\_INS}_2 \\ \text{P\_INS}_2 & = & recX_2.test_1(\overline{eq_1}\ \text{P\_INS}_3\ +\ \overline{neq_1}\ dec_1\ \overline{done}\ X_1) \\ \text{P\_INS}_3 & = & test_2(\overline{eq_2}\ \text{P\_INS}_4\ +\ \overline{neq_2}\ dec_2\ \overline{done}\ X_2) \\ \text{P\_INS}_4 & = & halt\ \delta \end{array}$$

Finally, the two-counters machine is modelled by the process $\mathcal{M}_\mathcal{P}$:

$$\mathcal{M}_\mathcal{P} = \text{P\_INS}_1 \| \text{COUNTER}_1 \| \text{COUNTER}_2$$

4.6 Last step: the halting problem of a two-counters machine

Let us consider the following proposition: $\mathcal{M}_\mathcal{P} \models$ always($[halt]$ff). This proposition means: "can the process $\mathcal{M}_\mathcal{P}$ never transmit the event $halt$?" This transmission models the execution of the instruction *halt* of the two-counters machine. So, the proposition can be rewritten in "can the two-counters machine never halt". Then, we have reduced the halting problem for a two-counters machine to the verification of the $Sub\mathbb{SL}$ formula always($[halt]$ff) by the process $\mathcal{M}_\mathcal{P}$. The proposition 1 is then proved.

Proposition 2 *The verification problem of an $Sub\mathbb{SL}$ formula by a $\mathcal{P}$ process is undecidable.*

Proof. From proposition 1 and from the undecidability result of the halting problem of a two-counters machine.

5 Discussion

The previous section show that the verification problem of real-time preemptive systems is undecidable. It is then theoretically impossible to verify functional properties over such systems. Two questions then arise: what are the causes of undecidability, and how to overcome this result. Firstly, let us remark that non preemptive processes, i.e., $\mathcal{P}$ processes without operator $(P \downarrow \overline{a} \uparrow \overline{b})Q$ can be translated into timed automata. The timed automata verification problem is decidable and is supported by numerous tools. The non preemptive processes verification problem then becomes decidable too. Secondly, in (Rusu, 1996), V. Rusu has shown

that the verification problem restricted to a time-bounded fragment of TCTL logic is also decidable, even in the case of preemptive processes. There are then two main undecidability sources: (1) the use of *time-unbound* operators, such as "always" or "never" is the future, for modeling properties to be verified, or (2) the use of the *suspension* and *resumption* operators for encoding preemptive scheduling strategy. These considerations show, from a theoretical point of view, where is the frontier between the decidable verification problems and the undecidable ones when considering real time systems.

However, as explained in introduction, industrial embedded systems such as integrated modular avionic (IMA) systems are based on on-line preemptive scheduling strategy and have to satisfy time-unbounded safety properties (e.g., engines are never reversed in flight). To verify such properties over IMA systems is then an undecidable problem. Abstract models or semi algorithms are then needed for verifying the behavior of scheduled preemptive systems.

In the specific case of critical embedded systems which have to go through certification, a specific functional property is often required: determinism, i.e., all computations must produce the same result (data and orders to actuators) when dealing with the same input environment, whatever the true execution time and the suspend and resume events interleaving. The benefit of this property, if ensured, is to allow abstraction of preemptive processes by non preemptive ones characterized by fixed beginning and end dates without modifying the whole behaviour of the system. To prove that a multi tasking system is deterministic has been shown decidable (Boniol et al., 2007). Then, to verify a functional property φ over a deterministic system S can be done following two decidable steps: firstly to prove determinism, and secondly to build to build an abstract non preemptive model S' of S and then to verify φ over S.

Unfortunately, in the other cases (i.e., non deterministic systems), the verification problem of functional properties on on-line scheduled preemptive multi tasking systems remains open.

References

Aceto, L., P. Bouyer, A. Burgueño, and K. G. Larsen (2001). The Power of Reachability Testing for Timed Automata. Technical Report LSV-01-6, ENS Cachan, France.

Boniol, F., C. Pagetti, and F. Revest (2007). Functionally deterministic scheduling. In *ISOLA07*.

Cerans, K. (1992). *Algorithmic Problems in Analysis of Real-Time Systems Specifications*. Ph. D. thesis, Institut of Mathematical and Computer Science, University of Latvia, Riga.

Henzinger, T. A., P. W. Kopke, A. Puri, and P. Varaiya (1998). What's Decidable About Hybrid Automata ? *Journal of Computer and Systems Sciences 57*, 94–124.

Henzinger, T. A., X. Nicollin, J. Sifakis, and S. Yovine (1992). Symbolic Model Checking for Real-Time Systems. In *7th. LiCS Symposium*, Santa-Cruz, California, pp. 394–406.

Laroussinie, F., K. G. Larsen, and C. Weise (1995). From Timed Automata to Logic - and Back. In *MFCS95*, Prague, Czech Republic.

Minsky, M. (1967). *Computation: Finite and Infinite Machines*. Prentice-Hall.

Rusu, V. (1996). *Vérification temporelle de programmes Electre*. Ph. D. thesis, Ecole Centrale de Nantes.