View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Correct Transformation from CCSL to Promela for verification

Frédéric Mallet, Ling Yin

To cite this version:

Frédéric Mallet, Ling Yin. Correct Transformation from CCSL to Promela for verification. [Research Report] RR-7491, INRIA. 2012, pp.33. hal-00667849

HAL Id: hal-00667849

https://hal.inria.fr/hal-00667849

Submitted on 8 Feb 2012

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

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

Correct Transformation from CCSL to Promela for verification

Ling Yin, Frédéric Mallet

RESEARCH REPORT
N° 7491
November 2011
Project-Teams Aoste

ISSN 0249-6399 ISRN INRIA/RR--7491--FR+ENG

Correct Transformation from CCSL to Promela for verification

Ling Yin, Frédéric Mallet

Project-Teams Aoste

Research Report n° 7491 — November 2011 — 33 pages

Abstract: Transforming a specification language into a language supported by a verification tool is a widely adopted way of doing formal verification. It enables the reuse of existing languages and tools. In this paper, we propose a correct transformation from CCSL to Promela to do formal verification by SPIN. To implement the transformation, we introduce "coincident instant" into Promela to deal with the discrete time in CCSL. Then we define property patterns to ensure that correctness properties are verified "coincident instant" by "coincident instant" during the verification. We define checkpoint transition systems (CTSs) to model source CCSL specifications and transformed Promel models. The proof of the correctness of our transformation relies on the checkpoint bisimulation defined over CTS. If a property is satisfied by a transformed Promela model, then it is satisfied by the source CCSL specification.

Key-words: MARTE; CCSL; verification; Promela

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Exemple de document utilisant le style rapport de recherche Inria

Résumé : Ce document montre comment utiliser le style RR.sty. Pour en savoir plus, consulter le fichier RR.dvi ou RR.pdf. Définissez toujours les commandes avant utilisation.

Mots-clés : calcul formel, base de formules, protocole, différentiation automatique, génération de code, modélisation, lien symbolique/numérique, matrice structurée, résolution de systèmes polynomiaux

1 Introduction

Real-time and embedded (RTE) systems have high requirements regarding safety aspects. Ensuring the correctness by applying formal methods during the development of such systems is a widely accepted approach. It is commonly agreed that using specification languages and their corresponding verification techniques can significantly reduce design errors and development cost. Among the approaches of doing formal verification, model transformation is widely used and has many advantages. Transforming a new specification into an existing one enables the reuse of existing tools. For example, it can avoid the extremely high cost of building a new efficient model checker. However, this approach provides a safe analysis method only if there is a guarantee that the process preserves the semantics of the original specification, that is to say that the transformation is correct. Depending on the source and target languages, this notion of correctness is not easy to achieve.

In this paper, we propose a correct transformation from CCSL to Promela for verification. CCSL [1] is a newly defined clock constraint specification language coming with MARTE. MARTE [2] is a UML Profile for Modeling and Analysis of Real-Time and Embedded systems, which has recently been adopted by the Object Management Group (OMG). It provides an explicit formal semantics to the UML elements and brings interoperability between the existing languages and formalisms of the RTE domain. A CCSL specification is attached to a MARTE model to specify its logical and chronometric time constraints. In this paper, we only focus on logical time constraints. CCSL supports both synchrony and asynchrony. Clock constraints in CCSL rely on three basic relations, coincidence, precedence and exclusion. Coincidence expresses synchronous dependency, that two clocks must tick simultaneously at that time. While precedence and exclusion express asynchronous dependency.

Promela is a verification modeling language supported by the SPIN model checker [3]. In Promela, system components are modeled as concurrent processes. The execution of processes are asynchronous and interleaved. In each step only one enabled action of a process is performed, without any assumptions of the relative speed of process executions. To model the logical discrete time in CCSL, we introduce a "coincident instant" concept into Promela. Then the three basic relations of CCSL constraints can be modeled. Tickings occur in one coincident instant are regarded as coincident. Correspondingly, properties of a transformed Promela model should be checked coincident instant by coincident instant. So we define some patterns for expressing correctness properties during the verification.

The correctness of our transformation is proved in three steps. First, we define checkpoint transition systems (CTSs) to model the behaviors of a CCSL specification and its transformed Promela model. Checkpoint parallel composition and checkpoint bisimulation over CTSs are also defined. Then we prove that our transformation preserves the checkpoint bisimulation. Based on that, we show that the logical truth is preserved during the transformation and verification. If a property (following the patterns) is satisfied by a transformed Promela model, then the corresponding property is satisfied by the source CCSL specification.

The remainder of the paper is structured as follows: Section 2 gives a short introduction to MARTE time structure and CCSL. Checkpoint transition systems and checkpoint bisimulation are defined in Section 3. Section 4 presents the transformation from CCSL to Promela and proves the correctness of the transformation. Related work is represented in Section 5. Then we conclude and discuss the future work in Section 6.

2 MARTE time structure and CCSL

2.1 Logical time

In RTE system design, timing requirements specify not only the physical time but also the causal functional intent. Logical multiform time is required. Modeling with logical time partial ordering was advocated in [4]. Logical clock, originally defined by Lamport [5], is a widely adopted way of representing logical time. A logical clock only relies on (partial or total) ordering of instants. The time duration between two instants is not necessarily relevant.

2.2 Clock

MARTE supports both discrete/dense and chronometirc/logical time. Clocks are defined to give access to time structures in MARTE. Here, we only exhibit logical time. A logical clock is a 5-tuple [6] $\langle \mathcal{I}, \prec, \mathcal{D}, \lambda, \mathcal{U} \rangle$, where $\mathcal{I}$ denotes a set of discrete instants, $\prec$ is a total, irreflexive and transitive binary relation on $\mathcal{I}$, named *strict precedence*, $\mathcal{D}$ is a set of labels, $\lambda : \mathcal{I} \rightarrow \mathcal{D}$ is a labeling function, $\mathcal{U}$ stands for a unit. An instant of a clock is when the clock ticks. Instants can be indexed by natural numbers in a way that respects the ordering on $\mathcal{I}$: let $\mathbb{N}^* = \mathbb{N} \backslash 0$, $idx : \mathcal{I} \rightarrow \mathbb{N}^*$, $\forall i \in \mathcal{I}, idx(i) = k$ if and only if i is the kth instant in $\mathcal{I}$. For any clock c $= \langle \mathcal{I}_c, \prec_c, \mathcal{D}_c, \lambda_c, \mathcal{U}_c \rangle$, c[k] denotes the kth instant in $\mathcal{I}_c$(i.e., k $= idx_c$(c[k])). To simplify computations, a virtual instant c[0] is used, c[0] $\prec$ c[1].

2.3 Time structure and CCSL

A time structure is a pair $< C, \preceq >$, where C is a set of clocks, $\preceq$ is a reflexive and transitive binary relation on $\bigcup_{c \in C} \mathcal{I}_c$, named *precedence*. From *precedence*, four instant relations are derived: *Coincidence* ($\equiv \triangleq \preccurlyeq \bigcap \succcurlyeq$), *Strict precedence*($\prec \triangleq \preccurlyeq \setminus \equiv$), *Independence*($\| \triangleq \overline{\preccurlyeq \bigcup \succcurlyeq}$), and *Exclusion*($\# \triangleq \prec \bigcup \succ$).

Usually, the number of instants of a set in a time structure is infinite. Specifying a full time structure using only instant relations is not realistic. So clock constraints, which are built on instants relation, are defined. The dedicated language CCSL for expressing clock constraints is introduced [1]. In CCSL, a clock constraint is a clock relation between two clocks or a clock expression which defines a new clock on existing clocks. A CCSL specification consists of several clocks and constraints among them. It is attached to a MARTE model to specify time requirements. A run of a CCSL specification is a sequence of coincident instants. A coincident instant has several valid configurations, each of which is a set of ticking decisions of the clocks without violation of any clock constraint. Which valid configuration is chosen in a run is non-deterministic.

2.4 Example

We use a common used example Digital Filter (DF) to show how to use CCSL and illustrate our transformation approach. DF is used in a video system. It reads image pixels from a memory, filters them and then sends output pixels out to a display device. One image is composed of LPI lines per image, each line consists of PPL pixels per line. The pixels are stored in words. A word contains PPW pixels per word, a line WPL words per line (WPL = [PPL/PPW]). The pixel transformation (digital filtering) is defined by a dot product: $y[k] = \Sigma_{j=-L}^{j=+L} c[j] * x[k-j]$, where k is a natural number, index of pixels in a line, y is an array of output pixels, x is an array of input pixels, c is an array of 2L + 1 constant coefficients. We can consider DF as a component with four signal ports. The input port InWord conveys WORD, the output port OutPixel conveys

PIXEL. The two other output ports (Ready and EndOfLine) are pure signals, that is, they do not carry values and are used for signaling some event occurrences.

The work flow of a DF can be specified as: It requests a new incoming word by asserting ready. In response, an external memory sends back the next word of the image (signal inWord). Signal outPixel is sequentially issued after receiving inWord and performing the filtering. Signal endOfLine is asserted each time the last pixel of a line is emitted(Cstr4,5). Each request is followed by a new word and no new request is sent before the preceding request has been acknowledged(Cstr1). Because of the chosen data structure, input pixels are packed within words of length PPW, which is 4 here, whereas output pixels are individually released(Cstr2,3). The output pixels should be delivered fast enough so the communication buffer contains at most two words(Cstr6-Cstr10). Its CCSL specification is represented below and the detailed analysis procedure can be found in [7]. We forbid nested clock expression, a clock defined by a clock expression is always named. This entails no loss of generality because nested clock expressions are replaced by clock definitions, one per clock expression [1]. E.g., $inWord \prec (outPixel\ filteredBy\ (1000)^{\omega})$ is split to Cstr2 and Cstr3.

$$
\begin{aligned}
&Cstr1.\ \ ready\ alternateWith\ inWord\\
&Cstr2.\ \ outPack \triangleq outPixel\ filteredBy\ (1000)^{\omega}\\
&Cstr3.\ \ inWord \prec_2 outPack\\
&Cstr4.\ \ endOfLine \triangleq outPixel\ filteredBy\ (0^7.1)^{\omega}\\
&Cstr5.\ \ twoWord \triangleq inWord\ filteredBy\ (0.1)^{\omega}\\
&Cstr6.\ \ outm1 \triangleq outPixel\ filteredBy\ 0.(1.0^7)^{\omega}\\
&Cstr7.\ \ outm2 \triangleq outPixel\ filteredBy\ 0^2.(1.0^7)^{\omega}\\
&Cstr8.\ \ twoWord \prec_2 outm2\\
&Cstr9.\ \ outm1 \prec_2 twoWord
\end{aligned}
$$

3 Checkpoint transition systems

Bisimulation is a usual way to compare the semantics of two systems. But for our transformation, the classical bisimulation equivalences, e.g., strong bisimualtion, weak bisimulation, are too strong and unnecessary. During the verification, properties are checked at special states. It is natural to introduce checkpoint as comparison unit. So we define checkpoint transition systems and checkpoint bisimulation over them.

Definition 1 Checkpoint Transition System(CTS)
A CTS is a tuple $T = \{S, \mathcal{A}, \rightarrow, I, clp\}$, where

- S is a finite set of states.
- $clp : S \rightarrow \{0, 1\}$ is a function defining checkpoints of the system. A state $s \in S$ with $clp(s) = 1$ is called a checkpoint.
- $\mathcal{A} = A \cup \tau$, A is a finite set of actions and τ is the invisible action.
- $I \subseteq S$ is the set of initial states. Each initial state $i \in I$ is a checkpoint, $clp(i) = 1$.
- $\rightarrow$ is the set of transitions: $\rightarrow \subseteq S \times L \times S$, $L = \mathcal{P}(\mathcal{A})$. Each label $l \in L$ is a set of actions, representing the simultaneously performed actions during that transition. A transition $(s, l, s^{'}) \in \rightarrow$ is often writhen as $s \xrightarrow{l} s^{'}$ for simplicity.

Checkpoint transition systems are defined to model the behaviors of a CCSL specification. So a visible action represents a ticking decision of a clock. It is denoted as a (clock a ticks), or $\neg a$ (clock a does not tick). Auto-concurrency of a visible action is forbidden since at one time, ticking decision of a clock is made only once. τ^+ and τ are not distinguished as usual. And we need to add some basic restrictions:

- Each transition is "conflict free", i.e., if a clock ticks, it can not "does not tick" at the same time. $a \in l \Rightarrow \neg a \notin l$, and vice versa.
- Not every state in S can be marked as a checkpoint. Since we relax the unit of comparison and operation to checkpoint, a path from a checkpoint to its intermediate post checkpoint should be conflict free and with no auto-concurrency.

  $\forall \rho = q_0, ..., q_n, q_i \xrightarrow{l_i} q_{i+1} (0 \leq i < n)$,
  if $clp(q_0) = 1, clp(q_j) = 0 (1 \leq j < n), clp(q_n) = 1$,
  then $a \in l_i \Rightarrow \neg a \notin \bigcup_0^{n-1} l_k, a \notin l_j (0 \leq j < n, j \neq i)$, so for $\neg a$

  q_n (resp. q_0) is called an immediate post (resp. pre) checkpoint of q_0 (resp. q_n). We denote the set of immediate pre and post checkpoints of s as $preclp(s)$ and $poclp(s)$ respectively.

We extend the transition relation to checkpoint transition relation $\Rightarrow$. $(s, \mu, s^{'}) \in \Rightarrow$ iff $\exists s, ..., s^{'}, s = q_0, s^{'} = q_n,\ q_i \xrightarrow{l_i} q_{i+1} (0 \leq i < n)$ and $s \in preclp(s\cdot), s^{'} \in poclp(s)$. $\mu = \bigcup_0^{n-1} l_i$. $(s, \mu, s^{'}) \in \Rightarrow$ is also denoted as $s \overset{\mu}{\Longrightarrow} s^{'}$.

Definition 2 Checkpoint Parallel composition

Given two CTSs $T_1 = \{S_1, \mathcal{A}_1, \Rightarrow_1, I_1, C_1\}$ and $T_2 = \{S_2, \mathcal{A}_2 \Rightarrow_2, I_2, C_2\}$, the checkpoint parallel composition of T_1 and T_2 is $T_1 || T_2 = \{S_1 \times S_2, \mathcal{A}_1 \cup \mathcal{A}_2, \Rightarrow, I_1 \times I_2\}$, where

$$\frac{s_1 \overset{\mu_1}{\Longrightarrow} s_1^{'} \in T_1, s_2 \overset{\mu_2}{\Longrightarrow} s_2^{'} \in T_2,\ \forall a \in \mathcal{A}_1 \cup \mathcal{A}_2,\ a \in \mu_1 \wedge\ \neg a \notin \mu_2}{(s_1, s_2) \overset{\mu_1 \cup \mu_2}{\Longrightarrow} (s_1^{'}, s_2^{'})}$$

Checkpoint bisimulation checks from checkpoint to checkpoint, requiring the compared systems have executed the same set of visible actions. Orders of the actions are irrelevant.

Definition 3 Checkpoint Bisimulation

Let $T_1 = \{S_1, \mathcal{A}, \Rightarrow_1, I_1\}$ and $T_2 = \{S_2, \mathcal{A}, \Rightarrow_2, I_2\}$ be two CTSs. A checkpoint bisimulation of T_1 and T_2 is a binary relation $\mathcal{R} \subseteq T_1 \times T_2$ such that

1. $\forall s_1 \in I_1, \exists s_2 \in I_2, (s_1, s_2) \in \mathcal{R}$ and $\forall s_2 \in I_2, \exists s_1 \in I_1, (s_2, s_1) \in \mathcal{R}$
2. for any $(s_1, s_2) \in \mathcal{R}$ it holds that
   (a) for all $s_1 \overset{\mu_1}{\Longrightarrow} s_1^{'}$, there exists $s_2 \overset{\mu_2}{\Longrightarrow} s_2^{'}$, where $\mu_1 - \{\tau\} = \mu_2 - \{\tau\}$.
   (b) $(s_1^{'}, s_2^{'}) \in \mathcal{R}$

   and vice versa.

T_1 and T_2 are checkpoint bisimulation equivalent if there exists a checkpoint bisimulation R between T_1 and T_2.

Theorem 1 Congruence w.r.t. parallel composition

For CTSs $T_1, T_1^{'}, T_2, T_2^{'}$, we assume that there exists some checkpoint bisimulation R_i between T_i and $T_i^{'}$, $(i \in \{1, 2\})$. Then, the relation:
$R = \{(\langle s_1, s_2 \rangle, \langle s_1^{'}, s_2^{'} \rangle) | (s_1, s_1^{'}) \in R_1\ \wedge (s_2, s_2^{'}) \in R_2\}$

is a checkpoint bisimulation for $T_1||T_2$ and $T_1^{'}||T_2^{'}$. That is to say, $T_1R_1T_1^{'}$ and $T_2R_2T_2^{'}$ implies $(T_1||T_2)R(T_1^{'}||T_2^{'})$.

Proof: The fulfillment of condition 1) is obvious. For condition 2), by $T_1R_1T_1^{'}$ $T_2R_2T_2^{'}$, we have $\mathcal{A}_1 = \mathcal{A}_1^{'}$ and $\mathcal{A}_2 = \mathcal{A}_2^{'}$. Consider a checkpoint transition $s_1 \overset{\mu_1}{\Longrightarrow} t_1$ in T_1, we have $(s_1, s_1^{'}) \in R_1$, $(t_1, t_1^{'}) \in R_1$, $clp(s_1^{'}) = 1$, $clp(t_1^{'}) = 1$, and $\exists\, s_1^{'} \overset{\mu_1^{'}}{\Longrightarrow} t_1^{'}$, where $\mu_1^{'} - \{\tau\} = \mu_1 - \{\tau\}$. Similarly, for and $s_2 \overset{\mu_2}{\Longrightarrow} t_2$ in T_2, we have $(s_2, s_2^{'}) \in R_2$, $clp(s_2^{'}) = 1$, $clp(t_1^{'}) = 1$, $(t_2, t_2^{'}) \in R_2$ and $\exists\, s_2^{'} \overset{\mu_2^{'}}{\Longrightarrow} t_2^{'}$, where $\mu_2^{'} - \{\tau\} = \mu_2 - \{\tau\}$. By the definition of checkpoint parallel composition, 1) if $\forall a \in \mathcal{A}_1 \cup \mathcal{A}_2, a \in \mu_1, \neg a \notin \mu_2$, we have $(s_1, s_2) \overset{\mu}{\Longrightarrow} (t_1, t_2)$ in $(T_1 \parallel T_2)$, $\mu = \mu_1 \cup \mu_2$. Then we get $\forall a \in \mathcal{A}_1^{'} \cup \mathcal{A}_2^{'}, a \in \mu_1^{'}, \neg a \notin \mu_2^{'}$. There is a checkpoint transition $(s_1^{'}, s_2^{'}) \overset{\mu^{'}}{\Longrightarrow} (t_1^{'}, t_2^{'})$ in $(T_1^{'} \parallel T_2^{'})$, $\mu^{'} = \mu_1^{'} \cup \mu_2^{'}$. And $\mu - \{\tau\} = (\mu_1 \cup \mu_2) - \{\tau\} = (\mu_1^{'} \cup \mu_2^{'}) - \{\tau\} = \mu^{'} - \{\tau\}$. 2) or $\neg(\forall a \in \mathcal{A}_1 \cup \mathcal{A}_2, a \in \mu_1, \neg a \notin \mu_2)$, then there is no corresponding checkpoint transition between (s_1, s_2) and (t_1, t_2) in $(T_1 \parallel T_2)$. Similar to case 1), we get $\neg(\forall a \in \mathcal{A}_1^{'} \cup \mathcal{A}_2^{'}, a \in \mu_1^{'}, \neg a \notin \mu_2^{'})$. There is no corresponding checkpoint transition between $(s_1^{'}, s_2^{'})$ and $(t_1^{'}, t_2^{'})$ in $(T_1^{'} \parallel T_2^{'})$ too. Starting from $(T_1^{'} \parallel T_2^{'})$ is dealt similarly. ■

To specify the properties of a checkpoint transition system, we define a checkpoint action linear temporal logic. It is an action based version of the linear temporal logic LTL, with checkpoint as checking unit.

Definition 4 Checkpoint Action LTL

Given an CTS $T = \{S, \mathcal{A}, \rightarrow, I, clp\}$, we consider linear temporal logic action formulas ranging over A:

$\phi ::= true \mid c \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathbf{X}\phi \mid \mathbf{F}\phi \mid \mathbf{G}\phi \mid \phi_1 \mathbf{U} \varphi_2$

Let $\pi_c =< s_1, \mu_1, s_2, \mu_2, \cdots >$ be a path of checkpoint transitions, s_i is a checkpoint, and μ_i is a checkpoint transition from s_i to s_{i+1}. π_c^i stands for the suffix of π_c starting from state s_i. The path satisfaction is defined as:

1. $\pi_c \models true$
2. $\pi_c \models c$ iff $c \in \mu_1$
3. $\pi_c \models \neg\phi$ iff $\pi_c \not\models \phi$
4. $\pi_c \models \phi_1 \wedge \phi_2$ iff $\pi_c \models \phi_1$ and $\pi_c \models \phi_2$
5. $\pi_c \models \mathbf{X}\phi$ iff $\pi_c^2 \models \phi$
6. $\pi_c \models \mathbf{F}\phi$ iff $\exists i \geq 1, \pi_c^i \models \phi$
7. $\pi_c \models \mathbf{G}\phi$ iff $\forall i \geq 1, \pi_c^i \models \phi$
8. $\pi_c \models \phi_1 \mathbf{U} \phi_2$ iff $\exists i \geq 1, \pi_c^i \models \phi_2$

and $\forall 1 \leq j \leq i-1, \pi_c^j \models \phi_1$

Then $T \models \phi$ iff for every path of checkpoint transitions starting from an inital state satisfies ϕ.

4 Correct transformation from CCSL to Promela for verification

4.1 Basic idea of the transformation

We transform a CCSL specification into a Promela model to do the formal verification by SPIN. *Clock* typed in Promela is defined to model clocks in CCSL. Clocks in a CCSL specification are declared as variables in type *Clock*

$$typedef\ Clock\{\ bool\ must_tick, cannot_tick, act_tick, dead;\}$$

Attribute *dead* is true when a clock cannot tick anymore. *must_tick* and *cannot_tick* express ticking conditions: When *must_tick* is true, the clock must tick in the current coincident instant.

When *cannot_tick* is true, the clock cannot tick in the coincident instant. If both are true in one coincident instant, the specification is not consistent. If neither of them are true, then a non-deterministic choice is made. The definitive choice on whether a clock actually ticks or not is modeled by *act_tick*.

Each kind of clock constraint is encoded as a Promela process, we call it an operator process. In the transformed Promela model of a CCSL specification, the *init* process creates an instantiation of corresponding operator process for each clock constraint in the specification. Each instantiation computes ticking conditions of related clocks. Then the *init* process makes ticking decision for each clock according to its condition.

Recalling that in Promela, each process defines an asynchronous thread of execution that interleaves its statement executions in arbitrary ways with other processes. In each step only one enabled action is performed. Processes can communicate via channels by either buffered message exchanges or rendezvous operations, and also through shared global variables. While the execution of a CCSL specification is based on simultaneous clock tickings in a coincident instant. To implement the simultaneity and avoid interleaving executions of simultaneous tickings belong to different coincident instants, we use rendezvous communications to control the slicing of coincident instants.

A coincident instant is implemented as three phases, $start \rightarrow firing \rightarrow end$. In the start phase, each operator process instantiation computes the ticking conditions of related clocks according to its state. Then in the firing phase, the *init* process makes the actual tickings according to the ticking conditions. Clocks are declared as global variables. So each ticking decision is a global one with respect to all the clock constraints. After that, the end phase comes. Each operator process instantiation updates its state according to the tickings made in the firing phase. All the attributes of clocks are reset before the execution of next coincident instant. The tickings between one cycle of $start \rightarrow end$ are regarded as simultaneous tickings in that coincident instant.

Synchronous-based constraint, e.g., *subClock*, *union*, cannot decide the ticking conditions in the start phase without knowing the ticking decisions of related clocks in the firing phase. E.g., without knowing ticking decision of the super clock b, $a\ subClock\ b$ may make $a.cannot_tick = false$ and $b.cannot_tick = false$ in a start phase of a coincident instant. While another constraint $b\ alternatesWith\ c$ may decide $b.cannot_tick = true$ in the start phase of the same coincident instant. Then in the firing phase, $a.cannot_tick == false$ and $b.cannot_tick == true$, the *init* process may make clock a tick and b not tick. But clock a should not tick since its super clock b does not tick in that coincident instant. To avoid this kind of problem and ensure the global feature of ticking decisions, we need some additional rules for synchronous based constraints. These operator processes do not decide ticking conditions right after entering a start phase. They update ticking conditions according to ticking decisions made in the firing phase of the same coincident instant. We force the *init* process to make the ticking decisions following a specific order of clocks in a firing phase. The ticking decision of a former clock is informed to related operator process instantiations to make them update affected ticking conditions. Then the *init* process makes the ticking decisions of latter clocks based on the updated ticking conditions. For detail implementation, please check the operator process for $subInstant$ bellow. The rules for ordering clocks are shown in Table 1.

A transformed Promela model consists three parts, declaration part, operator process part and the *init* process part.

In the declaration part, clocks are declared as global variables following the order. Rendezvous ports for slicing coincident instants are also declared. For the DF example, we get an order of the clocks as ready<outPixel<outPack<inWord<endOfLine<outm2<twoWord<outm1. So the declaration part of its transformed Promela model is:

Table 1: Clock ordering rules

Case	Order	Applied Constraints
a Cstr b	b < a	subClock, strictPrecedence
a Cstr b	b < a or a< b	alternatesWith, exclusion
b = a Cstr k	a < b	filteredBy, await, asFrom
c = a Cstr b	last(a,b) < c	union, intersec, preemption, concat, strictSampling, sup, inf, delayFor

```
#define NB_CLOCKS 8;
#define NB_CSTR 9;
#define ready 0
#define outPixel 1
#define outPack 2
#define inWord 3
#define endOfLine 4
#define outm2 5
#define twoWord 6
#define outm1 7
typedef Clock{bool must_tick,cannot_tick,act_tick,dead};
Clock clks[NB_CLOCKS];
chan start = [0] of{bool}; chan end = [0] of{bool};
chan subfir=[0] of{byte};chan suben=[0] of{bool};
chan gotosf=[0] of{bool}; byte subno,sub; bool inst;
```

In the operator process part, each kind of clock constraint used in the CCSL specification is encoded as an operator process. For the DF example, they are $alternatesWith$, $strictPrecedence$ and $filteredBy$. During execution of an operator process, states are maintained for computing ticking conditions. And ticking conditions are computed in a strictly monotonic way. *must_tick* and *cannot_tick* of related clocks are set to true according to the current state, but they will not be set back to false later. No backtracking guarantees the constructiveness and convergence of the model. For the detailed implementation for each kind of constraint, please check next subsection.

The last part of a transformed Promela model is the *init* process. It creates instantiates of operator processes and makes the actual tickings. Ticking decisions are global ones, with respect to the limitations made by all the operator process instantiations. Variable *inst* is used to mark the start and end of each coincident instant for property checking (see the property patterns in next subsection). E.g., the *init* process for the DF example is:

```
init  {
      inst=true;
      run alternatesWith(ready,inWord);
      run filteredBy(outPack,outPixel,w1);
      run strictPrecedence(inWord,outPack,2);
      run filteredBy(endOfLine,outPixel,w2);
      run filteredBy(twoWord,inWord,w3);
      run filteredBy(outm1,outPixel,w4);
      run filteredBy(outm2,outPixel,w5);
      run strictPrecedence(twoWord,outm2,2);
      run strictPrecedence(outm1,twoWord,2);
      int i;
loop: i=0;inst=false;do
      :: i<NB_CLOCKS->clks[i].must_tick = false; clks[i].cannot_tick = false; clks[i].
         act_tick=false;i++
      :: else -> break
      od;
      i=0; sub=0;
      do
      :: i< NB_CSTR -> start!true; i++
      :: else -> break
      od;
      i=0;do
```

```
      ::i<subno->suben!true;i++
      ::else->break
      od;
      byte k,nsubno;
      i=0; do
      ::i<NB_CLOCKS->
      if
      ::!clks[i].dead->
        if
        ::(clks[i].must_tick&&clks[i].cannot_tick)->assert(false)
        ::(clks[i].must_tick&&!clks[i].cannot_tick)->clks[i].act_tick=true
        ::(!clks[i].must_tick&&clks[i].cannot_tick)->clks[i].act_tick=false
        ::(!clks[i].must_tick&&!clks[i].cannot_tick)->clks[i].act_tick=true
        ::(!clks[i].must_tick&&!clks[i].cannot_tick)->clks[i].act_tick=false
        fi
      ::else->clks[i].act_tick=false
      fi;
      k=0; nsubno=subno-sub;do
      ::k<nsubno->subfir!i;k++
      ::else->break
      od;
      k=0;do
      ::k<subno-sub->gotosf!true;k++
      ::else->break
      od; i++;
    ::else->break
    od;
    i=NB_CSTR-1;do
    :: i>=0 -> end!true; i--
    :: else -> break
    od;
  inst=true; goto loop
  }
```

Then to do the verification by SPIN, correctness properties of a transformed Promela model are specified as linear temporal logic requirements (LTL). And the properties should be checked from coincident instant to coincident instant. As shown in last subsection, special variable *inst* is assigned to true to indicate that the next state is where properties should be checked. So we define the patterns for expressing correctness properties as φ:
$\varphi ::= true|inst \wedge \psi|\mathbf{X}\varphi|\mathbf{F}\varphi|\mathbf{G}\varphi|\varphi_1\mathbf{U}\varphi_2$
$\psi ::= c.act_tick|\neg\psi|\psi_1 \wedge \psi_2$
It can be regarded as a subset of LTL. There is no violation of the semantics of used operators. Expressing properties this way does not affect the verification algorithms used in SPIN. And it guarantees that properties are checked by the unit of coincident instant. For example, in the DF example, we expect that once the filter takes in one input, it must produce outputs. This is expressed as $p \rightarrow Fq$, where

$$p = clks[inWord].act_tick \wedge inst,$$

$$q = clks[outPixel].act_tick \wedge inst,$$

4.2 Transformation correctness

According to the semantics of CCSL, each clock constraint can be modeled as a CTS $T = \{S, \mathcal{A}, \rightarrow, I\}$. Visible action set of $A = \{a, \neg a|\ a$ is a clock involved in the constraint$\}$. A transition represents a valid ticking configuration of the clock constraint. Every state is marked as a checkpoint. A CCSL specification is a conjunction of its clock constraints. A run of it is a sequence of coincident instants. Each coincident instant has several valid configurations. In each valid configuration, a set of clocks tick simultaneously without violating any clock constraint. For a conjunction of two clock constraints, if a valid configuration of one clock constraint is not conflict with a valid configuration of the other clock constraint, then the union of the two configurations

is a valid configuration of the conjunction. So the conjunction of CCSL constraints corresponds to the checkpoint parallel composition defined above. The CTS of a CCSL specification is the checkpoint parallel result of CTSs of its clock constraints.

SPIN manual [3] gives an operational semantics of Promela. Every Promela process defines a finite state automata (S, s_0, L, T, F). The set of states S corresponds to the possible points of control location. Transition relation T defines the flow of control. The transition label set L links each transition with a specific basic statement that defines the executability and the effect of that transition. The set of final states F is defined with the help of Promela end-state, accept-state and progress-state labels. A global system state is a tuple ($gvars$, $procs$, $chans$, $exclusive$, $handshake$, $timeout$, $else$, $stutter$), $gvars$ is a finite set of global variables, $procs$ is a finite set of processes, $chans$ is a finite set of message channels, $exclusion$ is an integer to enforce $atomic$ and d_step sequences, $handshake$ is an integer to enforce the rendezvous operations, $timeout$ and $else$ enforce the semantics of the matching Promela statements, and $stutter$ enforces the stutter extension rule. The Promela semantics engine executes step by step, selecting one executable statement and applying the effect to the global system state. A state transition between two system states is labeled by the selected basic statement. We can get the CTS of a Promela model during the execution. If a basic statement can be executed, a transition labeled by the statement is added. During the construction of CTSs, we observe that a CTS of a Promela model PS, which is transformed from a CCSL specification $S = C_1 \wedge C_2 \cdots \wedge C_2$, is a checkpoint parallel result of the CTSs of Promela models for constraints in S: $CTS_PS = CTS_PC_1 || CTS_PC_2 \cdots || CTS_PC_n$, where CTS_PC_i is the CTS of Promela model for constraint C_i.

Since checkpoint bisimulation is a congruence for checkpoint parallel composition. To prove the transformation is checkpoint bisimulation preserving, we just need to prove that the transformation for each kind of CCSL constraint preserves checkpoint bisimulation. Clock constraints can be classified in three categories: 1)synchronous-based constraints, 2)asynchronous-based constraints and 3)mixed constraints. We present the transformation for and its correctness for each kind of constraint bellow.

1. **Synchronous-based clock constraints**
   Synchronous-based clock constraints are inspired from synchronous languages and define infinitely many coincidence instant relations.

   - **subClock**: a clock relation, a subClock b means that each instant in clock a is coincident with one instant in b:

     $$(\forall k \in \mathbb{N}^\star)(\exists j \in \mathbb{N}^\star)(a[k] \equiv b[j])$$

     And there is a special case $tightsubClock$, the sub clock ticks coincidentally with its super clock at every instant. A constraint is not violated if no clock ticks, so there is a self-loop checkpoint transition labeled as $\{\neg a | a \in A\}$ for each state, as Figure 1 shows.

     There is no limitation on clock b. And clock a can tick only if b ticks. The ticking result of one *instant* does not affect that of the next *instant*. As Figure **??** shows, the transitions are all self-transitions. The ticking condition of clock b is not limited by *subClock* and the ticking condition of clock a can not be fixed without knowing the ticking result of clock b in the same *step*. So the Promela process for *subClock* does no computation in the start phase of an *isntant*. As discussed above, rendezvous ports $subfireable$ and $gotosf$ are added between the init process and the instantiation of

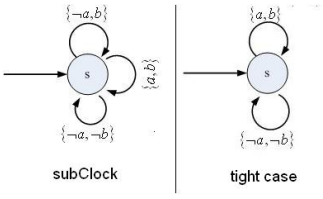

Figure 1: CTS of *a subclock b*

process *subClock* to ensure the global decision feature. In firing phase, the *init* process makes the ticking decision of each clock follow the order. For each clock, its decision is broadcast to "open" each subClock instantiation. It checks if the clock is related: 1) if match, decides the ticking condition of the other clock, and closes itself in that instant (sub++,the rest clocks will not send it ticking decisions since subno-sub rendezvous ports will be executed for next clock) and go to wait for the execution of the end phase (started by rendezvous port end). 2) if not match, it is blocked until the rendezvous port gotosf executes. Execution of gotosf will make the operator process go back to the subfireable?index to wait for the broadcast of next clock. If not match, it does not go back to the subfireable?index directly. Because in that case, it can be called again by the same clock(due to the do loop of calling subfireable), one operator may be called twice while the one really needs it may not be called. The ticking decision of later clocks is made according to its updated ticking conditions. And at the end of the firing phase, all the synchronous-based clocks are "closed" and waiting for the execution of end, since all clocks are traversed.

```
proctype subclock(int cLeft,cRight; bool istight){
    byte index;
    do
    ::enable?true;
    atomic{subenable?true;
    if
    ::istight->
        if
        ::clocks[cLeft].cannot_tick=true->clocks[cRight].cannot_tick=true
        ::else->skip
        fi
    ::else->skip
    fi;}
loopsub:    atomic{subfireable?index;
    if
    ::index==cRight->sub++;
        if
        ::clocks[cRight].act_tick==true->
            if
            ::istight->clocks[cLeft].must_tick=true
            ::else->skip
            fi
        ::else->clocks[cLeft].cannot_tick=true
        fi
    ::gotosf?true->goto loopsub
    fi;}
    fireable?true;
    od
}
```

During the execution of the Promela model, an instantiation of the *subClock* is created by the *init* process. And the *init* process makes ticking decisions of clock a and clock b according to their ticking conditions made by the instantiation. We can get the CTS for the transformation for *subClock*, as Figure 2. (The blue transition

can not be enabled in tight case). For simplicity $clocks[c].act_tick = true$ (resp. $clocks[c].act_tick$) which means clock c ticks (resp. does not tick) is denoted as c (resp. $\neg c$). We are only interested in clock tickings, so we replace other labels with τ (event hiding) and do not show them explicitly in the figure for simplicity. A sequence $s(\xrightarrow{\tau})^* p \xrightarrow{a} q(\xrightarrow{\tau})^* s'$ is shown as $s \xrightarrow{a} s'$. . States after $inst = true$ are marked as checkpoints (colored in grey).

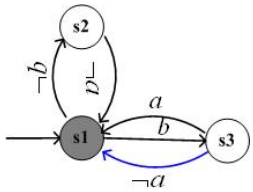

Figure 2: $CTS_P(a\ subClock\ b)$

- **exclusion**, says that none of the instants of the two clocks are coincident.

$$((\forall j \in \mathbb{N}^*, a[j] \in \mathcal{I}_a)(\forall k \in \mathbb{N}^*, b[k] \in \mathcal{I}_b)(\neg a[j] \equiv b[k]))$$

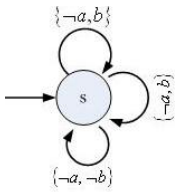

Figure 3: CTS of $a\ exclusion\ b$

```
proctype exclusion(int cLeft,cRight){
    byte index;
    do
     ::enable?true;subenable?true;
irrxe:atomic{subfireable?index;
    if
    ::index==cRight||index==cLeft->sub++;
        if
        ::clocks[cRight].act_tick->clocks[cLeft].cannot_tick=true
        ::clocks[cLeft].act_tick->clocks[cRight].cannot_tick=true
        ::else->skip;
        fi;
    ::gotosf?true->goto irrxe
    fi;}
    fireable?true;
    od
}
```

As Table 1 shows, for $a\ exclusion\ b$, the clock order could either be b<a or a<b. In the declaration part of the transformed Promela model, either one of them may be chosen. Since the checkpoint bisimulation only requires the set of visible actions, ignoring their order. So no matter which order is chosen, the transformation still preserves the checkpoint bisimulation. We show the CTS for order a<b in Figure 4.

It is easy to see that the two CTSs, Figure **??** for CCSL constraint and Figure 4 for transformed Promela model, are checkpoint bisimilar.

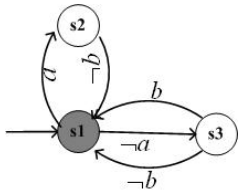

Figure 4: $CTS_P(a\ exclusion\ b)$

- **union**, a clock expression creates a new clock c which ticks whenever a or b ticks. $c = a$ union b is defined as

$$(\forall k \in \mathbb{N}^\star)(\exists j \in \mathbb{N}^\star)(c[k] \equiv a[j]) \vee (c[k] \equiv b[j])$$

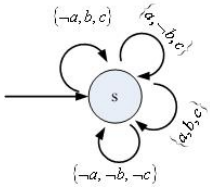

Figure 5: CTS of $c = a\ union\ b$

```
proctype union(int cLeft,cRight,new){
    byte index,count1,count2;
    do
    ::enable?true;
     atomic{subenable?true;
    if
    ::clocks[new].cannot_tick->clocks[cRight].cannot_tick=true;clocks[cLeft].
        cannot_tick=true
    ::else->skip
    fi;}
irru: atomic{subfireable?index;
    if
    ::index==cRight||index==cLeft->count1++;
        if
        ::clocks[cLeft].act_tick||clocks[cRight].act_tick->count2++
        ::else->skip
        fi;
        if
        ::count1<2->gotosf?true->goto irru
        ::count1==2->sub++;
            if
            ::count2==0->clocks[new].cannot_tick=true
            ::else->clocks[new].must_tick=true
            fi
        fi;
    ::gotosf?true->goto irru
    fi;}
    fireable?true;
    od
}
```

From Figure 6 and Figure 5, we can see that our transformation for union preserves checkpoint bisimulation.

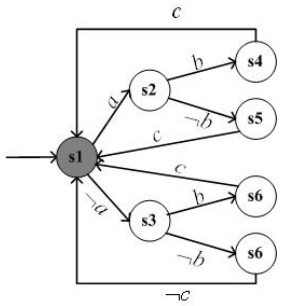

Figure 6: $CTS_P(a\ union\ b)$

- **intersection**, $c = a \wedge b$ defines a new clock c which ticks whenever both a and b tick, as Figure 7 shows.

$$(c\ subClock\ a) \bigwedge (c\ subClock\ b) \bigwedge ((\forall i \in I_a)(\forall j \in \mathcal{I}_b)(\exists k \in \mathcal{I}_c)(i \equiv j) \Rightarrow (i \equiv k))$$

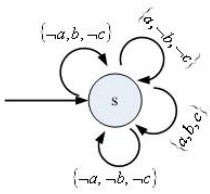

Figure 7: CTS of $c = a\ intersection\ b$

```
proctype intersection(int cLeft,cRight,new){
    byte index,count1,count2;
    do
    ::atomic{enable?true;count1=0;count2=0;}
    subenable?true;
irri: atomic{subfireable?index;
    if
    ::index==cRight||index==cLeft->count1++;
        if
        ::index==cLeft&&clocks[cLeft].act_tick->count2++
        ::index==cRight&&clocks[cRight].act_tick->count2++
        ::else->skip;
        fi;
        if
        ::count1<2->gotosf?true->goto irri
        ::count1==2->sub++;
            if
            ::count2==2->clocks[new].must_tick=true
            ::else->skip
            fi
        fi;
    ::gotosf?true->goto irri
    fi;
    }
    fireable?true;
```

```
    od
}
```

Only when both clock a and clock b tick in a coincident instant, clock c is forced to tick in the same instant, otherwise clock c cannot tick in that instant. $CTS_P(C = a\ intersection\ b)$, is checkpoint bisimilar with CTS of $c = a\ union\ b$ in Figure 7, is shown in Figure 8.

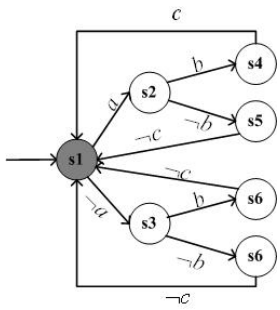

Figure 8: $CTS_P(a\ intersection\ b)$

- **filteredBy**, a clock expression, $c = a \blacktriangledown w$ defines a new clock c. $w \uparrow k$ denotes the index of the kth '1' in Binary word w. Each instant of c is coincident with one instant of a (the mth instant of a), if the corresponding (m^{th}) bit of w is '1'.

$$(\forall k \in \mathbb{N}^*)(c[k] \equiv a[w \uparrow k])$$

When the bit of binary word is '1': If the right clock ticks in that coincident instant,

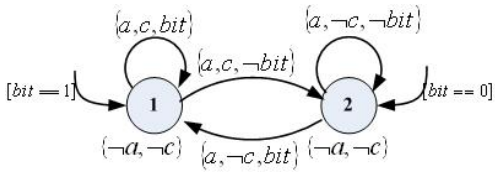

Figure 9: CTS of $c = a\ filteredBy\ w$

the new defined clock must tick. Otherwise, the new defined clock cannot tick. An instantiation of *subInstant* whose parameter *istight* is true should be called.

```
typedef Binaryword{bool b[10];byte peristart,periend,len;}

proctype filteredBy(int cL,cR; Binaryword w){
    bool state=w.b[0]; bool endf=false;
    do
    ::atomic{state?true;
    if
    ::(state&&!endf)-> run subInstant(cL,cR,true);subno++
    ::(!state&&!endf)->clocks[cL].cannot_tick=true
    ::endf->skip
    fi;}
    atomic{end?true;
    if
    ::!endf->
      if
      ::clocks[cR].act_tick->atomic{id2++;
        if
        ::w.peristart==-1 ->
```

```
            if
            ::id<w.n->
              if
              ::w.b[id2]==0 ->state=false
              ::w.b[id2]==1 ->state=true
              fi
            ::else->endf=true
            fi
          ::else->
            if
            ::id2<=w.periend->skip
            ::else->id2=((id-w.peristart)%(w.periend-w.peristart+1))+w.
                peristart
            fi;
            if
            ::w.b[id2]==0->state=false
            ::w.b[id2]==1->state=true
            fi
          fi}
        ::else->skip
        fi
      ::else->skip
      fi}
      od
} proctype subInstant(int cL,cR;bool istight){
        byte index;
        atomic{subenable?true;
        if
        ::istight->
          if
          ::clocks[cL].cannot_tick->clocks[cR].cannot_tick=true
          ::else->skip
          fi
        ::else->skip
        fi;}
lins: atomic{subfireable?index;
        if
        ::index==cR->subno--;
          if
          ::clocks[cR].act_tick->
            if
            ::istight->clocks[cL].must_tick=true
            ::else->skip
            fi
        ::else->clocks[cL].cannot_tick=true
          fi
        ::gotosf?true->goto lins
        fi;}
        od}
```

For $filteredBy$, bit value of the binary word also affects the behaviors. Statements for getting the bit value from the binary word are not replaced by τ, as Figure 10 shows.

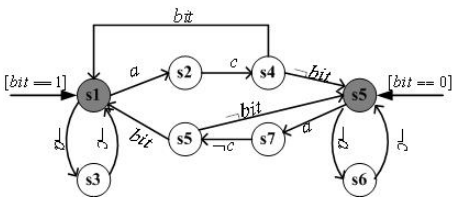

Figure 10: $CTS_P(c = \ a\ filteredBy\ w)$

- **sup**, $c = a\ sup\ b$ defines a clock c that is slower than both a and b. The kth tick of c is coincident with the later of the kth tick of a and b.

$$(\forall k \in \mathbb{N}^*, a[k] \in \mathcal{I}_a, b[k] \in \mathcal{I}_b, c[k] \in \mathcal{I}_c)$$

$$\begin{cases} c[k] = a[k], \ if \ b[k] \preceq a[k] \\ c[k] = b[k], \ if \ b[k] \succeq a[k] \end{cases} \tag{1}$$

If both clock a and b have ticked the same times (diff==0), then 1) clocks a, b

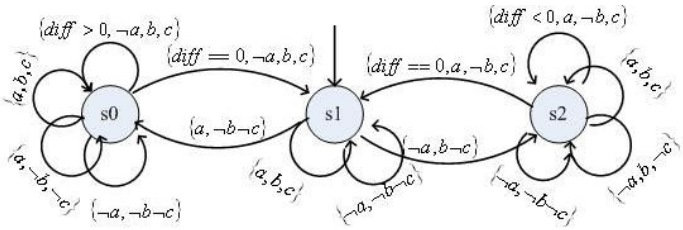

Figure 11: CTS of $c = a \ sup \ b$

and c tick coincidentally in next *instant*. 2) clock a(or b) ticks ahead, $diff > 0$(or $diff < 0$) is the successor. When $diff > 0$: Clock a is faster than b, then 1) clock a can still tick ahead. 2) clock c ticks coincidentally with b. 3) all of clocks a, b and c tick in the next step(the mth instant for b and c, n^{th} for a, and $n > m$). If the difference of the indexes a-b is 0, state $diff == 0$ is the successor. State $diff < 0$: the opposite of $diff > 0$, clock b is faster than a. Clock c ticks coincidentally with the slower one or the faster clock tick alone in next coincident instant.

```
proctype sup(int cLeft,cRight,new){
    byte diff,count1,count2,index;
    do
    ::atomic{enable?true;count1=0;count2=0;
    if
    ::diff>0->run subInstant(new,cRight,true);subno++
    ::diff<0->run subInstant(new,cLeft,true);subno++
    ::diff==0->subno++;subenable?true;
    irr2:atomic{subfireable?index;
        if
        ::index==cRight||index==cLeft->count1++;
            if
            ::index==cLeft&&clocks[cLeft].act_tick->count2++
            ::index==cRight&&clocks[cRight].act_tick->count2++
            ::else->skip;
            fi;
            if
            ::count1<2->gotosf?true->goto irr2
            ::count1==2->sub++;
                if
                ::count2==2->clocks[new].must_tick=true
                ::else->clocks[new].cannot_tick=true
                fi
            fi
        ::gotosf?true->goto irr2
        fi}
    fi;}
    atomic{fireable?true;
        if
        ::diff==0->subno--
        ::else->skip
        fi;
        if
```

```
        ::clocks[cLeft].act_tick&&!clocks[cRight].act_tick->diff++
        ::clocks[cLeft].act_tick&&clocks[cRight].act_tick->skip
        ::!clocks[cLeft].act_tick&&clocks[cRight].act_tick->diff--
        ::!clocks[cLeft].act_tick&&!clocks[cRight].act_tick->skip
        fi}
    od
}
```

When current state is $diff > 0$, if clock a ticks while clock b does not tick in one coincident instant, their difference increases (diff++). State remains in $diff > 0$. So for simplicity, we do not show this kind of transition in the graph, see Figure 12. We only show diff related transitions which will change the state and denote them as $diff == 0$, $diff < 0$ and $diff > 0$.

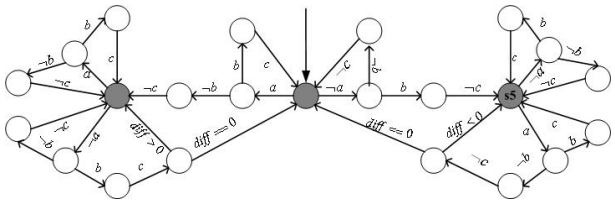

Figure 12: $CTS_P(c = \ a \ sup \ b)$

- **inf**. This expression is the dual of *sup*. The kth tick is coincident with the earlier of the kth tick of a and b.

$$(\forall k \in \mathbb{N}^*, a[k] \in \mathcal{I}_a, b[k] \in \mathcal{I}_b, c[k] \in \mathcal{I}_c)$$

$$\begin{cases} c[k] = b[k], \ if \ b[k] \preceq a[k] \\ c[k] = a[k], \ if \ b[k] \succeq a[k] \end{cases} \tag{2}$$

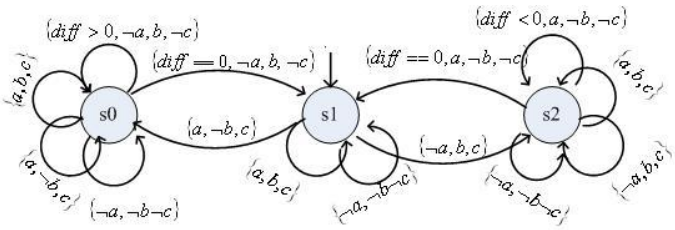

Figure 13: CTS of $c = a \ inf \ b$

```
proctype inf(int cLeft,cRight,new){
    byte diff,count1,count2,index;
    do
    ::atomic{enable?true;count1=0;count2=0;
    if
    ::diff>0->run subInstant(new,cLeft,true);subno++
    ::diff<0->run subInstant(new,cRight,true);subno++
    ::diff==0->subno++;
    irri2:atomic{subfireable?index;
        if
```

```
            ::index==cRight||index==cLeft->count1++;
                if
                ::index==cLeft&&clocks[cLeft].act_tick->count2++
                ::index==cRight&&clocks[cRight].act_tick->count2++
                ::else->skip;
                fi;
                if
                ::count1<2->gotosf?true->goto irri2
                ::count1==2->sub++;
                    if
                    ::count2==0->clocks[new].cannot_tick=true
                    ::else->clocks[new].must_tick=true
                    fi
                fi
            ::gotosf?true->goto irri2
            fi}
        fi;}
        atomic{fireable?true;
            if
            ::diff==0->subno--
            ::else->skip
            fi;
            if
            ::(clocks[cLeft].act_tick&&clocks[cRight].act_tick)->skip
            ::(clocks[cLeft].act_tick&&!clocks[cRight].act_tick)->diff++
            ::(!clocks[cLeft].act_tick&&clocks[cRight].act_tick)->diff--
            ::(!clocks[cLeft].act_tick&&!clocks[cRight].act_tick)->skip
            fi}
        od
}
```

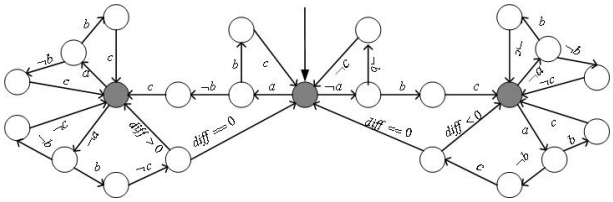

Figure 14: $CTS_P(c = \ a \ inf \ b)$

- **asFrom**, $c = a \ asFrom \ n$ defines a clock c that is a tight sub-clock of clock a starting after index n.

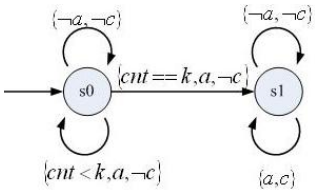

Figure 15: CTS of $c = a \ asFrom \ n$

```
proctype asFrom(int cLeft,cRight,n){
    bool state;
    int count;
    do
    ::atomic{enable?true;
    if
    ::state->run subInstant(cLeft,cRight,true);subno++
    ::!state->clocks[cLeft].cannot_tick=true;
    fi;}
```

```
    atomic{fireable?true;
    if
    ::!state->
        if
        ::clocks[cRight].act_tick->count++;
        ::else->skip;
        fi;
        if
        ::count==n-1->state=true
        ::else->skip
        fi
    ::else->skip
    fi}
    od
}
```

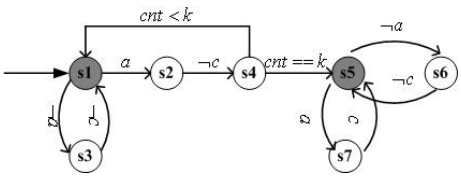

Figure 16: $CTS_P(c = \ a\ asFrom\ n)$

- **preemption**,defines a clock c which behaves like a while b does not tick. When b ticks, clock c dies.

$$((\forall k \in \mathbb{N}^*, a[k] \in \mathcal{I}_a)(a[k] \prec b[k]) \Rightarrow (c[k] \equiv a[k])) \bigwedge c\mathbf{subClock}a$$

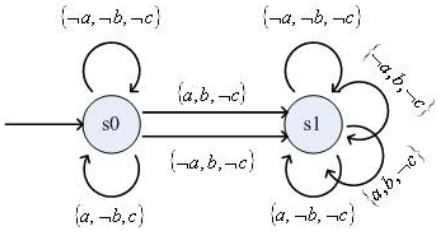

Figure 17: CTS of $c = a$ *preemption* b

```
proctype preemption(int cLeft,cRight,new){
    bool end,flag;
    byte index;
    do
    ::atomic{enable?true;
    if
    ::!end->run subInstant(new,cLeft,true);subno++
    ::else->subenable?true;
irrp: atomic{subfireable?index;
        if
        ::index==cRight->sub++;
          if
          ::clocks[cRight].act_tick->clocks[new].dead=true;end=true;flag=true
          ::else->skip
          fi
        ::gotosf?true->goto irrp;
```

```
            fi;}
        fi;}
        atomic(fireable?true;
        if
        ::flag->subno--;flag=false
        ::else->skip
        fi}
        od
}
```

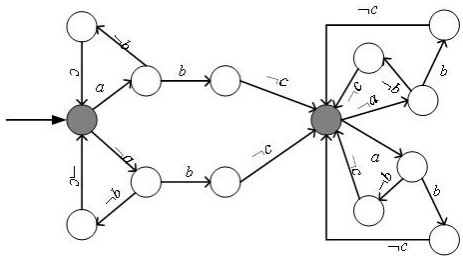

Figure 18: $CTS_P(c = \ a\ preemption\ b)$

- **await**, $c = a\ await\ n$ defines a clock c. It ticks in coincidence with the next n^{th} strictly future tick of a, and then dies.

$$((\exists a[n] \in \mathcal{I}_a)(c[1] \equiv a[n])) \bigwedge |\mathcal{I}_c| = 1$$

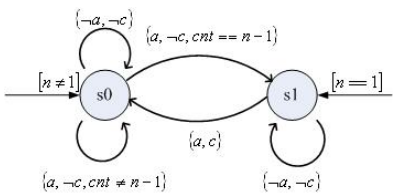

Figure 19: CTS of $c = a\ await\ n$

```
proctype await(int cLeft,cRight,n){
    bool state;
    int count;
    if
    ::n==1->state=true;
    ::else->state=false;
    fi;
    do
    ::atomic{enable?true;
    if
    ::state->run subInstant(cLeft,cRight,true);subno++
    ::!state->clocks[cLeft].cannot_tick=true;
    fi;}
    atomic{fireable?true;
    if
    ::state->
      if
      ::clocks[cLeft].act_tick->state=false;clocks[cLeft].dead=true;
      ::else->skip
      fi
```

```
      ::!state->
        if
        ::clocks[cRight].act_tick->count++;
        ::else->skip
        fi;
        if
        ::count==n-1->state=true
        ::else->skip
        fi
      fi}
     od
}
```

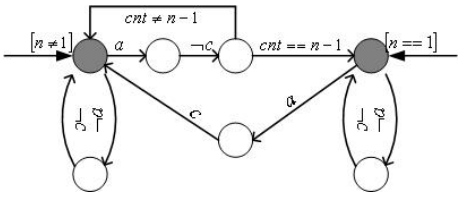

Figure 20: $CTS_P(c = \ a\ await\ n)$

- **concat**,$c = a * b$ defines a new clock c behaves like a up to the death of a. When a dies, the expression behaves like b.

let l=$|\mathcal{I}_a|$,$(\forall k \in \mathbb{N}^*, c[k] \in \mathcal{I}_c)$
$((k <= l) \bigwedge (c[k] \equiv a[k])) \bigvee$
$((k > l) \bigwedge (\exists m \in \mathbb{N}^*, b[m] \preceq a[l] \prec b[m+1]) \bigwedge (b[k+m-l] \in \mathcal{I}_b) \bigwedge (c[k] \equiv b[m+k-l]))$

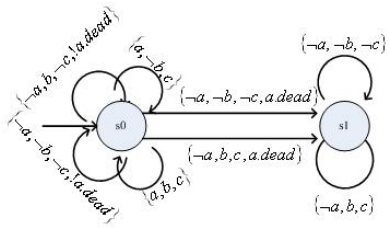

Figure 21: CTS of $c = a\ concat\ b$

```
proctype concat(int cLeft,cRight,new){
   byte index,count; bool state,flag;
    do
    ::atomic{enable?true;count=0;
        if
        ::state->subInstant(new,cRight,true);subno++
        ::else->subenable?true;
loopcon:     atomic{subfireable?index;
             if
             ::index==cLeft||index==cRight->count++;
                 if
                 ::count<2->gotosf?true->goto loopcon;
                 ::count==2->sub++;
                     if
                     ::!clocks[cLeft].dead->
                       if
                       ::clocks[cLeft].act_tick->clocks[new].must_tick=true
```

```
                        ::else->clocks[new].cannot_tick=true
                        fi
                ::else->state=true;flag=true
                    if
                    ::clocks[cRight].act_tick->clocks[new].must_tick=true
                    ::else->clocks[new].cannot_tick=true
                    fi
                fi
            fi
        ::gotosf?true-> goto loopcon
        fi}
    fi;}
    atomic{fireable?true;
    if
    ::flag->subno--;flag=false
    ::else->skip
    fi}
    od
}
```

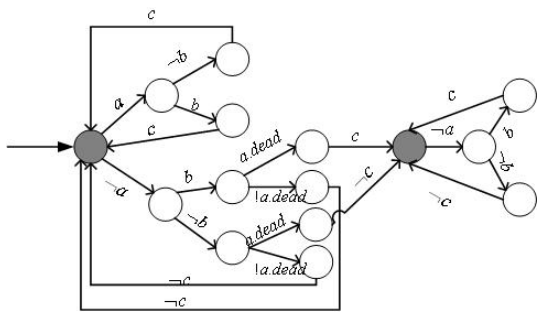

Figure 22: $CTS_P(c = \ a \ concat \ b)$

Asynchronous-based clock constraints

Asynchronous-based clock constraints define infinitely many precedence instant relations. They characterize the asynchronous features.

- **strictPrecedence**, a precedence-based relation, says that the left clock is strictly faster than the right one:

$$(\forall k \in \mathbb{N}^*)(a[k] \prec b[k])$$

For model checking, we bound the number of precedence operator with an integer k. I.e., $a \ strictPrecedence_k \ b$ means clock a is faster than b by k indexes. When the bound is reached, the left clock can not tick if the right one does not tick. There are three valid configurations for next instant: 1) none of them ticks, 2)both of them tick, 3) right clock ticks while left does not. It is like that the left clock is a sub clock of the right one only in that coincident instant. And for the sake of global decision feature, the nondeterministic choice can not be made by *strictPrecedence*. So *subInstant* is called when the bound is reached. That is why we define the order of clocks in *strictPrecedence* as $right < left$. Rendezvous ports *suben*, *subfir* and *gotosf* are used to notify the ticking decision of the right clock. Related *subInstant* instantiations are forced to update ticking conditions of their left clocks.

```
proctype strictPrecedence(int cL,cR,k){
    bool s1,s2,s3; s3=true; byte l=0;
    do
     ::atomic{start?true;
     if
     ::s1->skip
```

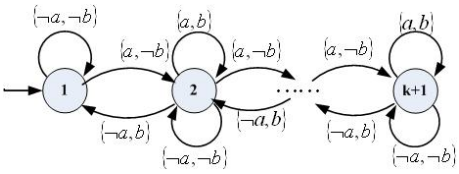

Figure 23: CTS of a $strictPredence_k$ b

```
  ::s2->run subInstant(cL,cR,false);subno++
  ::s3->clks[cR].cannot_tick=true
  fi;}
  atomic{end?true;
  if
  ::clks[cL].act_tick&&!clks[cR].act_tick->l++
  ::!clks[cL].act_tick&&clks[cR].act_tick->l--
  fi;
  if
  ::s1->
    if
    ::l==0 ->
      if
      ::!clks[cL].act_tick&&clks[cR].act_tick->s3=true;s1=false;s2=false
      ::else->skip
      fi
   ::l==k ->
     if
      ::clks[cL].act_tick&&!clks[cR].act_tick->s2=true;s1=false;s3=false
      ::else->skip
      fi
   ::else->skip
   fi
  ::s2->
    if
    ::!clks[cL].act_tick&&clks[cR].act_tick->s1=true;s2=false;s3=false
    ::else->skip
    fi
  ::s3->
    if
    ::clks[cL].act_tick&&!clks[cR].act_tick->s1=true;s2=false;s3=false
    ::else->skip
    fi
  fi}
 od}
```

When the bound is reached, the left clock can not tick if the right one does not tick. There are three valid configurations for next instant: 1) none of them ticks, 2)both of them tick, 3) right clock ticks while left does not. It is like that the left clock is a sub clock of the right one only in that coincident instant. And for the sake of global decision feature, the nondeterministic choice can not be made by *strictPrecedence*. So *subInstant* is called when the bound is reached. That is why we define the order of clocks in *strictPrecedence* as $right < left$.

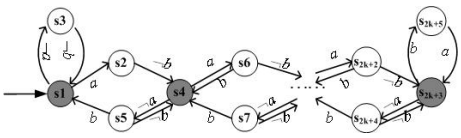

Figure 24: $CTS_P(a\ strictPrecedence_k\ b)$

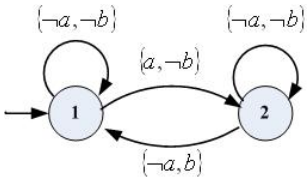

Figure 25: CTS of $a\ alternatesWith\ b$

- **alternatesWith** is a kind of mutual precedence.

$$(\forall k \in \mathbb{N}^{\star})\,((a[k] \prec b[k]) \wedge (b[k] \prec a[k+1]))$$

```
proctype alternatesWith(int cL,cR){
    bool state=true;
    do
     ::atomic{start?true;
     if
     ::state->clks[cR].cannot_tick=true
     ::!state->clks[cL].cannot_tick=true
     fi;}
     atomic{end?true;
     if
     ::state->
       if
       ::clks[cL].act_tick->state=false
       ::else->skip
       fi
     ::!state->
       if
       ::clks[cR].act_tick->state=true
       ::else-> skip
       fi
     fi}
    od}
```

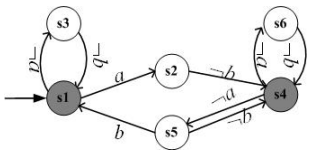

Figure 26: $CTS_P(a\ alternatesWith\ b)$

synchronization. Unlike the alternation, it does not impose a strict ordering on instants of a and b. Instead, the kth instants of a and b are not ordered, but they both precede the $(k+1)^{th}$ instants of a and b.

$$(\forall k \in \mathbb{N}^{*}, a[k] \in \mathcal{I}_a, b[k] \in \mathcal{I}_b, a[k+1] \in \mathcal{I}_a, b[k+1] \in \mathcal{I}_b)((a[k] \prec b[k+1]) \bigwedge (b[k] \prec a[k+1]))$$

```
proctype Synchronization(int cLeft,cRight){
    byte diff,count;
    do
    ::atomic{enable?true;
    if
    ::diff==1->clocks[cLeft].cannot_tick=true;
```

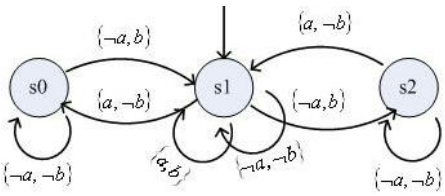

Figure 27: CTS of *a synchronization b*

```
        ::diff==-1->clocks[cRight].cannot_tick=true;
        fi;}
        atomic{fireable?true;
        if
        ::(clocks[cLeft].act_tick&&clocks[cRight].act_tick)->skip
        ::(clocks[cLeft].act_tick&&!clocks[cRight].act_tick)->diff++
        ::(!clocks[cLeft].act_tick&&clocks[cRight].act_tick)->diff--
        ::(!clocks[cLeft].act_tick&&!clocks[cRight].act_tick)->skip
        fi}
        od
}
```

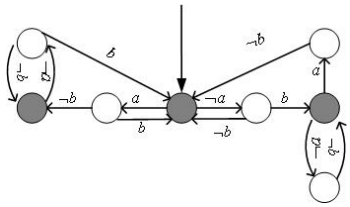

Figure 28: CTS_P(*a synchronization b*)

2. **Mixed constraints**
   Mixed constraints combine coincidences and precedences. They are used to synchronize clock domains in globally asynchronous and locally synchronous models.

   - **strictSampling**: $c = a \searrow b$, a clock expression, defines a new clock c which is a sub clock of b that ticks only after a tick of clock a.

$$(\forall k \in \mathbb{N}^\star)(\exists i, j \in \mathbb{N}^\star)$$
$$(c[k] \equiv b[i]) \wedge (a[j] \prec b[i]) \wedge (b[i-1] \preceq a[j])$$

```
proctype strictSampling(int cLeft,cRight,new){
    bool state1,state2,state3;
    state1=true;
    do
    ::atomic{enable?true;
    if
    ::state1->clocks[new].cannot_tick=true;
    ::state2->clocks[new].cannot_tick=true;
    ::state3->run subInstant(new,cRight,true);subno++
    fi;}
    atomic{fireable?true;
        if
        ::state1->
            if
            ::clocks[cLeft].act_tick&&clocks[cRight].act_tick->state3=true;
                state1=false;state2=false
```

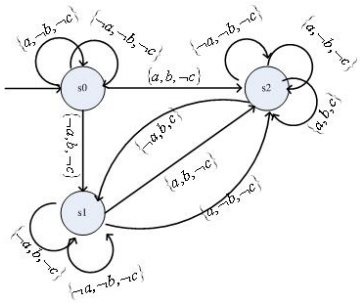

Figure 29: CTS of $c = a$ *strictSampling* b

```
            ::!clocks[cLeft].act_tick&&clocks[cRight].act_tick->state2=true;
                state1=false;state3=false
            ::else->skip
            fi
        ::state2->
            if
            ::clocks[cLeft].act_tick->state3=true;state1=false;state2=false
            ::else->skip
            fi
        ::state3->
            if
            ::!clocks[cLeft].act_tick&&clocks[cRight].act_tick&&clocks[new].
                act_tick->state2=true;state1=false;state3=false
            ::else->skip
            fi
        fi}
    od
}
```

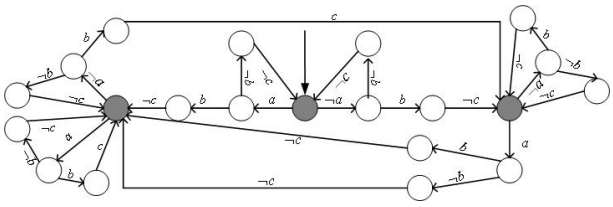

Figure 30: $CTS_P(a$ *strictSampling* $b)$

3. **delayFor**, defines a clock c = a **delayedFor** ns **on** b. Clock a is a trigger, b is a base clock. At each tick of a the head of ns is dequeued and encoded in a binary word associated with the operator. This binary word is a kind of diary that contains the future rendezvous with b ticks.

$$
\begin{aligned}
&(c \textbf{ subClock } b) \bigwedge \\
&((\forall i \in \mathbb{N}^*, a[i] \in \mathcal{I}_a)(\exists n_i \in \mathbb{N}^*), \\
&(\exists j \in \mathbb{N}^*, j \geq n_i,\ b[j], b[j-n_i] \in \mathcal{I}_b, b[j-n_i] \preceq a[i] \prec b[j-n_i+1]), \\
&(\exists k \in \mathbb{N}^*, c[k] \equiv b[j]))
\end{aligned}
$$

The encoding of $delayfor$ is a generalization of the encoding for *strictSampling*. When the trigger a ticks, an instantiation of subdelayInstant is created. Process subdelayInstant

is similar to subInstant, except that it counts the tickings of the base clock b and makes the c tick coincidentally with the ns^{th} tick of b. And it does not decide the c.cannot_tick to true when the ns is not reached. Because clock c may be made tick in that coincident instant together with the ns^{th} ticking of b after different ticking of the trigger a. The subfireable?index matched by clock c is used to force c not to tick in that instant if none instantiations of subdelayInstant restrict c must tick coincidentally with b in that instant.

```
proctype delayfor(int cLeft,cRight,new,n){
bool state1,state2,state3,state4,flag;
    state1=true;
    byte index;
    do
    ::atomic{enable?true;
    if
    ::state1->clocks[new].cannot_tick=true
    ::state2->clocks[new].cannot_tick=true
    ::state3&&flag->run subdelayInstant(new,cRight,true,n);subno++;
    ::else->skip
    fi;}
    subenable?true;
loopdelay:atomic{subfireable?index;
    if
    ::index==new->sub++;
        if
        ::clocks[new].must_tick==true->skip
        ::else->clocks[new].act_tick=true
        fi
    ::gotosf?true->goto loopdelay
    fi;}

    atomic{fireable?true;
        if
        ::state1->
          if
          ::clocks[cLeft].act_tick&&clocks[cRight].act_tick->state3=true;state1=
              false;state2=false;state4=false
          ::!clocks[cLeft].act_tick&&clocks[cRight].act_tick->state2=true;state1=
              false;state3=false;state4=false
          ::else->skip
          fi
        ::state2->
          if
          ::clocks[cLeft].act_tick->state3=true;state1=false;state2=false;state4=
              false
          ::else->skip
          fi
          ::state3->flag=false
          if
          ::clocks[cLeft].act_tick->state4=true;state1=false,state2=false;state3=
              false
          ::else->skip
          fi
        ::state4->
          if
          ::clocks[cLeft].act_tick->state3=true;state1=false;state2=false;state4=
              false
          ::else->skip
          fi
        fi
    }
    od
}
proctype subdelayInstant(int cLeft,cRight; bool istight; int delay){
    byte index,count;
    count=0;
    subenable?true;
  lpdi: atomic{subfireable?index;
    if
    ::index==cRight->
        if
        ::clocks[cRight].act_tick==true->count++;
```

```
            ::else->skip
            fi;
            if
            ::count==delay->subno--;
                if
                ::istight->clocks[cLeft].must_tick=true
                ::else->skip
                fi
            ::else->gotosf?true->goto lpdi
            fi
        ::gotosf?true->goto lpdi
        fi;}
}
```

In practice, a special case that the trigger clock is also the base clock $c = a\ delayForn$ is often used to model offset. Its transformation, simpler than the general $delayFor$, is shown bellow.

```
proctype delayfor(int dClock,new,n){

    bool state; int count;
    if
    ::n==0->state=true
    ::else->state=false
    fi;
    do
    ::atomic{enable?true;
    if
    ::state->run subInstant(new,dClock,true);subno++
    ::else->clocks[new].cannot_tick=true
    fi;}
    atomic{fireable?true;
        if
        ::!state->
          if
          ::clocks[dClock].act_tick->count++
          ::else->skip
          fi;
          if
          ::count==n->state=true
          ::else->skip
          fi
        ::else->skip
        fi
    }
    od
}
```

4.3 Logical truth preservation

SPIN checks a property by constructing the synchronous product of the transition system produced by the semantics engine and the automata transformed from the LTL formula. We denote the CTS of a CCSL specification as CTS_C and the CTS of its transformed Promela model as CTS_P. The transition system of the Promela model generated by the SPIN semantics engine is denoted as TS_P. Since LTL formulas are expressed following the φ pattern (section **??**), properties are checked upon checkpoints. $CTS_P \models \varphi \Rightarrow CTS_C \models \phi$, ϕ is the corresponding checkpoint action LTL formula of φ. For a path $\pi = s_1, s_2, \cdots$ in TS_P,

- $\pi \models inst \wedge\ c.act_tick$ iff $s_1 \models inst \wedge\ c.act_tick$.
  $s_1 \models inst \wedge\ c.act_tick$ indicates that s_1 is a checkpoint, and statement $c.act_act = true$ is executed in a path from an immediate previous checkpoint of s_1, we denote it as s_0. Then in CTS_P, there exists a checkpoint transition $s_0 \overset{\mu}{\Longrightarrow} s_1$, and $c \in \mu$. Since CTS_C and CTS_P are checkpoint bisimilar, we get $s_0^{'} \overset{\mu^{'}}{\Longrightarrow} s_1^{'}$ in CTS_C, $s_0^{'}\ R\ s_0$, $s_1^{'}\ R\ s_1$,

$\mu - \{\tau\} = \mu - \{\tau\}$ and $c \in \mu^{'}$. That is to say that there is a checkpoint path π_c in CTS_C that $\pi_c \models c$.

- $\pi \models inst \wedge \neg \psi$ iff $s_1 \models inst \wedge \neg \psi$.
  It means that statements representing the ticking decisions of $\neg\psi$ are executed from s_0 to s_1. Then in CTS_P, there is a path from s_0 to s_1 satisfying $\neg\phi$. By $CTS_C\ R\ CTS_P$, we have $\pi_c \models \neg\phi$ in CTS_C. ϕ is $f(\psi)$, abandoning the *inst* and denoting each *c.act_tick* as c.

- $\pi \models inst \wedge \psi_1 \wedge \psi_2$ iff $s_1 \models inst \wedge \psi_1 \wedge \psi_2$
  It means that the statements representing the ticking decisions of ψ_1 and ψ_2 are executed from s_0 to s_1. Then there is a path in CTS_P from s_0 to s_1 holds both ϕ_1 and ϕ_2, $\phi_1 = f(\psi_1)$ and $\phi_2 = f(\psi_2)$. By checkpoint bisimulation, we get the corresponding path in CTS_C, $\pi_c \models \phi_1 \wedge \phi_2$.

- $\pi \models \mathbf{X}\varphi$ iff $\pi^2 \models \varphi$.
  If $\varphi = inst \wedge \psi$, $\pi^2 \models \varphi$ iff $s_2 \models \varphi$, corresponding to $s_1 \overset{\mu}{\Longrightarrow} s_2$, $\mu \models \phi$ in CTS_P, $\phi = f(\psi)$. Then $s_1^{'} \overset{\mu^{'}}{\Longrightarrow} s_2^{'}$ and $\mu^{'} \models \phi$ in CTS_C. Since there is a self loop checkpoint transition for each checkpoint, we can get $\pi_c \models \mathbf{X}\phi$.
  If φ has temporal operator, satisfiability is still preserved. Because neither the φ pattern nor the Checkpoint Action LTL changes the semantics of temporal operators.

- Similar to $\mathbf{X}$, we can get $\pi \models \mathbf{F}\varphi \Rightarrow \pi_c \models \mathbf{F}\phi$, $\pi \models \mathbf{G}\varphi \Rightarrow \pi_c \models \mathbf{G}\phi$ and $\pi \models \mathbf{U}\varphi \Rightarrow \pi_c \models \mathbf{U}\phi$.

For the other direction $TS_P \models \varphi \Leftarrow CTS_C \models \phi$, the proof is just the reverse procedure. As a conclusion, verifying a property of a transformed Promela model corresponds to checking the satisfiability of the action based property of the source CCSL specification.

5 Related work

There are a lot of works on transforming a specification language into the input language of a prominent model checking tool [8, 9, 10]. The transformation for CCSL has also been studied, e.g., CCSL to Signal and Timed Petri Net in [11], and CCSL to PSL in [12]. But their transformations are not well suited for our objective. The synchronous languages, like Esterel and Signal, are not convenient to express the asynchronous constraints in CCSL. E.g., for "a alternatesWith b", an additional super clock has to be built explicitly, clock a takes the true value and clock b takes the false value. The lack of non-deterministic choice in synchronous languages is also a major difficulty in the transformation. In this paper, we only concern on logical time, so we do not need the full expressive power of Timed Petri Net or Timed Automata, which would increase the cost of the verification dramatically. PSL is compatible with CCSL but it is mostly used for assertions in hardware electronic systems. Since our goal is to to verify a CCSL specification, it is more natural to build a CCSL specification as a model supported by a verification tool rather than as a temporal logic based specification. VHDL is also considered. But it is difficult to express coincident relations in VHDL and the delta-cycle simulation semantics of VHDL may cause fake errors in transformation. Finally, we decide to use Promela as the target language. Promela supports nondeterminism well. And its synchronous and asynchronous communication mechanisms can be used to model the coincidence, precedence and exclusion relations of CCSL (the "coincident instant"). The "coincident instant" way of implementing time slices in Promela uses native functionality provided by Promela rather than by additional extensions. Similar

works can be found in [13], which models time constructs from the process algebra ACP by macro definitions entirely on the level of Promela, and in [14] which defines a new variable type timer corresponding to the discrete time countdown timer. But it is difficult to express the simultaneously clock tickings required by our transformation in their extensions.

Bisimulation relations are widely used to prove the semantical equivalence of two systems. Our checkpoint bisimulation is a weak version of bisimulation relation, because the classical strong bisimulation is too strong and unnecessary in our transformation. The checkpoint bisimulation is inspired from the communicative bisimulation proposed by [15]. We reuse the checkpoint concept, but adjust the bisimulation relation according to our formalism and goal. We prove the logical truth preservation of our transformation and verification based on the checkpoint bisimulation. The logical truth preservation here is a integration of two aspects: 1)validity equivalence between state-based logics on state-labeled structures and action-based logics on transition-labeled structures. A similar work can be found in [16] and [17], which discusses the translation between CTL over Kripke Structures structure and ACTL over labeled transitions systems. 2) logical truth preservation of bisimulations, which is discussed by [18] and [19]. We adopt their ideas but describe the preservation in the context of our formalisms and property patterns.

6 Conclusion and future work

In this paper, we have presented a method of transforming a CCSL specification into a Promela model to do the formal verification by SPIN. The transformation is not just a syntax mapping but semantic compatibility. And we prove the correctness of the method. First, we prove that the transformation preserves checkpoint bisimulation. Then we show that verifying a property of a transformed Promela model corresponds to checking the satisfiability of the action based property of the source CCSL specification.

In our transformation, clocks in a CCSL specification are implemented as global variables in the Promela model. This affects the partial order reduction in SPIN. However cluster-based partial order reduction can solve this problem. In [20], they group the global variables and encapsulate the dependencies between processes. It enhances the partial order reduction scheme and reduces the checking time. We are developing a tool to implement our transformation and formal verification on top of SPIN. And we will adapt the cluster-based partial order reduction into our tool to improve the time complexity. After that, we will extend our work to dense time.

References

[1] Charles André. Syntax and semantics of the clock constraint specification language(CCSL). Research Report RR-6925,INRIA, 2009.

[2] Object Management Group. UML Profile for MARTE, v1.0. formal/2009-11-02, 2009.

[3] Gerard J. Holzmann, SPIN Model Checker, Addison Wesley, 2003.

[4] Edward A. Lee, Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. IEEE Transaction on Computer-Aided Design, vol. 17, December 1998.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM,21(7):558-565, July 1978.

[6] Charles André,Frédéric Mallet,Robert de Simone. Modeling time(s). Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, Eds., Springer. 4735:559-573, 2007.

[7] Charles André,Frédéric Mallet. Specification and Verification of Time Requirements with CCSL and Esterel. Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compliers, and tools for embedded systems, 2009.

[8] Leila Ribeiro, Osmar Marchi dos Santos, Fernando Luis Dotti and Luciana Foss, Correct transformation: From object-based graph grammars to PROMELA, Science of Computer Programming, 2011.

[9] R. K. Poddar and P. Bhaduri. Verification of giotto based embedded control systems. Nordic Journal of Computing, 2006.

[10] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, S. Yovine. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. Proceedings of the 40th IEEE Conference on Decision and Control, 2001.

[11] Frédéric Mallet, Charles André. UML/MARTE CCSL, Signal and Petri nets. Research Report RR-6545, INRIA, 2008.

[12] Régis Gascon, Frédéric Mallet and Julien DeAntoni. Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. Research Report RR-7459, INRIA, 2011.

[13] Bošnački, Implementing Discrete Time in Promela and Spin, International Con- ference on Logic in Computer Science, LIRA '97, University of Novi Sad, Yu- goslavia, 1997.

[14] Dragan Bošnački, Dennis Dams, Integrating Real Time into Spin: A Prototype Implementation, Theoretical Computer Science 309: 313-355, 2003.

[15] David de Frutos-Escrig, Fernando Rosa-Velardo, and Carlos Gregorio-Rodríguez. New Bisimulation Semantics for Distributed Systems. In FORTE 2007, Proceedings, LNCS 4547: 143¨C159. Springer, 2007.

[16] Rocco De Nicola, Frits Vaandrager, Stefania Gnesi and Gioia Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. Compututer Network and ISDN Systems. 25: 761-778, 1993.

[17] Bengt Jonsson, Ahmed Hussain Khan and Joachim Parrow. Implementing a model checking algorithm by adapting existing automated tools. Automatic verification methods for finite state systems, 407: 179-188, 1990.

[18] Martin R. Neuhäuβer, Joost-Pieter Katoen. Bisimulation and logical preservation for continuous-time markov decision processes. Concur 2007, 4703: 412-427, 2007.

[19] Johan Van Benthem, Jan Bergstra. Logic of Transition systems. Journal of Logic, Language and Information, 3(4): 247-283, 1994.

[20] Twan Basten, Dragan Bošnački, Enhancing Partial-Order Reduction via Process Clustering. 16th IEEE Conference on Automated Software Engineering ASE 2001, IEEE Computer Society Press, 2001.

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399