Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 312 (2015) 161–177

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Symbolic Model for Timed Concurrent Constraint Programming

Jaime Arias[1]

Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
CNRS, LaBRI, UMR 5800, F-33400 Talence, France

Michell Guzmán[2]

Comète, LIX, Laboratoire de l'Ecole Polytechnique associé à l'INRIA/CORDI-S. France
DECC, Universidad del Valle. Colombia

Carlos Olarte[3]

ECT, Universidade Federal do Rio Grande do Norte. Natal, Brazil
DECC, Pontificia Universidad Javeriana Cali. Colombia

Abstract

Concurrent Constraint Programming (`ccp`) is a model for concurrency where agents interact with each other by telling and asking constraints (i.e., formulas in logic) into a shared store of partial information. The `ntcc` calculus extends `ccp` with the notion of discrete time-units for the specification of reactive systems. Moreover, `ntcc` features constructors for non-deterministic choices and asynchronous behavior, thus allowing for (1) synchronization of processes via constraint entailment during a time-unit and (2) synchronization of processes along time-intervals. In this paper we develop the techniques needed for the automatic verification of `ntcc` programs based on symbolic model checking. We show that the internal transition relation, modeling the behavior of processes during a time-unit (1 above), can be symbolically represented by formulas in a suitable fragment of linear time temporal logic. Moreover, by using standard techniques as difference decision diagrams, we provide a compact representation of these constraints. Then, relying on a fixpoint characterization of the timed constructs, we obtain a symbolic model of the observable transition (2 above). We prove that our construction is correct with respect to the operational semantics. Finally, we introduce a prototypical tool implementing our method.

Keywords: Concurrent constraint programming, temporal logic, model checking

[1] Email: ariasalmeida@gmail.com
[2] Email: michellrad@gmail.com
[3] Email: carlos.olarte@gmail.com

http://dx.doi.org/10.1016/j.entcs.2015.04.010

1 Introduction

In the last years we have seen how Concurrent Constraint Programming [20,21] (ccp) has been extensively used to specify and program concurrent systems. The increasing interest in the community for this powerful model of concurrency is perhaps due to its simplicity and tight connection to logic: processes *tell* and *ask* information (formulas in logic) in a store of partial information; moreover, processes can be seen as both computing agents and as logic formulas. The use of ccp models has pervaded different areas in science (e.g., biochemical systems), engineering (e.g., security protocols, mobile and service oriented computing and social networks) and even the arts (e.g., tools for multimedia interaction)–see a survey in [17]. Nevertheless, in spite of the many semantic and logical frameworks designed to reason about ccp processes, the automatic verification of ccp programs has received little attention so far.

This paper aims at providing the theoretical and practical tools to carry out the verification of systems specified in the ntcc [16] calculus, a timed extension of ccp to model reactive systems. For that, we propose a symbolic representation of the behavior of processes and we prove that such symbolic model is suitable to be used as a basis for standard techniques in model checking. One of the challenges to define such symbolic representation is that the operational semantics of ntcc is given by two different transition relations: the internal transition representing the steps of the processes during a time-interval and the observable transition describing how processes evolve along time-units. Moreover, the proposed model has to deal with two non-elementary temporal constructs in ntcc: $!P$ that executes infinite copies of P (one per time-unit) and $\star P$ that describes asynchronous behavior by delaying P an unbounded (but finite) number of time-units. As we shall see, we can neatly characterize the behavior of these constructs by means of a fixpoint computation.

Organization and contributions. We start recalling the ntcc calculus in Section 2. Section 3 describes our approach to represent symbolically the behavior of ntcc processes. We prove that the symbolic model can be obtained for any process in a finite number of steps (Theorem 3.6) and also that our construction is correct with respect to the operational semantics (Theorem 3.9). In Section 3 we also present some examples to show how to compute the symbolic model of a process. Section 4 describes the logic that we shall use to specify properties and Section 5 shows how the symbolic model can be used in standard (symbolic) model checking algorithms. We conclude in Section 6 by pointing out to related work and briefly describing a prototypical tool implementing our methodology.

2 The ntcc Calculus

Concurrent Constraint Programming (ccp) [20,21] (see a survey in [17]) is a model for concurrency that combines the traditional operational view of process calculi with a *declarative* view based on logic. This allows ccp to benefit from the large set of reasoning techniques of both process calculi and logic.

Processes in ccp *interact* with each other by *telling* and *asking* constraints (pieces of information) in a common store of partial information. The type of constraints processes may act on is not fixed but parametric in a constraint system (CS). Intuitively, a CS provides a signature from which constraints can be built from basic tokens (e.g., predicate symbols) and variables, and two basic operations: conjunction ($\sqcup$) and variable hiding ($\exists$). The CS defines also an *entailment* relation ($\vdash$) specifying inter-dependencies between constraints: $c \vdash d$ means that the information d can be deduced from the information represented by c. Such systems can be formalized as a Scott information system as in [21], or they can be built upon a suitable fragment of logic e.g., as in [10,16]. Here we shall follow the second approach and constraints are seen as formulas in intuitionistic logic.

Definition 2.1 (Constraint System) *A constraint system is a tuple $(\mathcal{C}, \vdash)$ where $\mathcal{C}$ is a set of formulas (constraints) built from a first-order signature and the following grammar*

$$F := \mathtt{true} \mid A \mid F \wedge F \mid \exists \overline{x}.F$$

where A is an atomic formula. We shall use c, c', d, d', etc, to denote elements of $\mathcal{C}$. Moreover, let Δ be a set of non-logical axioms of the form $\forall \overline{x}.[c \supset c']$ where all free variables in c and c' are in $\overline{x}$. We say that d entails *d', written as $d \vdash d'$, iff the sequent $\Delta, d \longrightarrow d'$ is probable in LJ [13].*

2.1 Process Syntax

The ntcc calculus [16] extends ccp with time-units for the specification of reactive systems, i.e., systems that continuously interact with the surrounding environment. In this language, time is conceptually divided into *discrete intervals* (or *time-units*). Intuitively, in a particular time interval, a process P receives a stimulus (i.e., a constraint) from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the resulting store. The resting point also determines a residual process Q, which is then executed in the next time interval.

Definition 2.2 (Syntax) *Processes $P, Q, \ldots \in Proc$ are built from constraints in the underlying constraint system as follows:*

$$P, Q, \ldots ::= \mathbf{tell}(c) \mid \textstyle\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \mid P \parallel Q \mid \mathbf{local}\ x(P)$$
$$\mid \mathbf{next}\ P \mid \mathbf{unless}\ c\ \mathbf{next}\ P \mid !\,P \mid \star P$$

Untimed processes. The process $\mathbf{tell}(c)$ adds the constraint c to the current store, thus making c available to other processes in the current time interval. Let I be a finite set of indexes. The *ask* process $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ non-deterministically chooses a process P_i s.t. c_i is entailed by the current store. The chosen alternative, if any, precludes the others. If no choice is possible in the current time-unit, all the alternatives are precluded from execution. Ask processes thus define a powerful synchronization mechanism based on entailment of constraints. For the sake of readability, we shall omit "$\sum_{i \in I}$" when I is a singleton and we simply write $\mathbf{when}\ c\ \mathbf{do}\ P$.

$$\frac{}{(X;\mathbf{tell}(c),\Gamma;d)\longrightarrow(X;\Gamma;c\wedge d)}\ \mathrm{R_{TELL}} \qquad \frac{d\vdash c_i,\quad i\in J}{(X;\sum_{j\in J}\mathbf{when}\ c_j\ \mathbf{do}\ P_j,\Gamma;d)\longrightarrow(X;P_i,\Gamma;d)}\ \mathrm{R_{ASK}}$$

$$\frac{x\notin X\cup fv(d)\cup fv(\Gamma)}{(X;\mathbf{local}\ x(P),\Gamma;d)\longrightarrow(X\cup\{x\};P,\Gamma;d)}\ \mathrm{R_{LOC}} \qquad \frac{d\vdash c}{(X;\mathbf{unless}\ c\ \mathbf{next}\ P,\Gamma;d)\longrightarrow(X;\Gamma;d)}\ \mathrm{R_{UNL}}$$

$$\frac{}{(X;!\,P;\Gamma;d)\longrightarrow(X;P,\mathbf{next}\,!\,P;\Gamma;d)}\ \mathrm{R_{REP}} \qquad \frac{n\geq 0}{(X;\star P,\Gamma;d)\longrightarrow(X;\mathbf{next}^{\,n}P,\Gamma;d)}\ \mathrm{R_{STAR}}$$

$$\frac{(X;\Gamma;c)\equiv(X';\Gamma';c')\longrightarrow(Y';\Delta';d')\equiv(Y;\Delta;d)}{(X;\Gamma;c)\to(Y;\Delta;d)}\ \mathrm{R_{STR}} \qquad \frac{(\emptyset;\Gamma;c)\longrightarrow^{*}(X;\Gamma';d)\not\longrightarrow}{\Gamma\overset{(c,\exists X.d)}{\Longrightarrow}\mathbf{local}\ X(F(\Gamma'))}\ \mathrm{R_{OBS}}$$

Fig. 1. Internal ($\longrightarrow$) and Observable ($\Longrightarrow$) transitions. Let $\Gamma=\{P_1,\ldots,P_n\}$. The future of Γ, $F(\Gamma)$, is $\{F_P(P_1),\ldots,F_P(P_n)\}$ where $F_P(\sum_{j\in J}\mathbf{when}\ c_j\ \mathbf{do}\ P_j)=\emptyset$ and $F_P(\mathbf{next}\,Q)=F_P(\mathbf{unless}\,c\,\mathbf{next}\,Q)=Q$. The set of free variables in d (resp. P) is denoted as $fv(d)$ (resp. $fv(P)$).

The process $P \parallel Q$ represents the parallel composition of P and Q. The process **local** $x(P)$ behaves like P, except that all the information on x produced by P can only be seen by P, i.e., x is a local variable of P.

Timed processes. The process **next** P delays the execution of P for one time-unit. We shall use $\mathbf{next}^n\,P$ as an abbreviation for $\mathbf{next}\,\mathbf{next}\cdots\mathbf{next}\,P$ where **next** is repeated n times. The process **unless** c **next** P is also a unit delay but the process P is executed in the next time-unit only if the guard c cannot be entailed from the current store. This is known as a *negative ask* or the *preemption* of P.

The process $!\,P$ represents unboundedly many copies of P but one per time-unit. This is, $!\,P$ can be seen as $P \parallel \mathbf{next}\,P \parallel \mathbf{next}^{\,2}P\cdots$. This construct is powerful enough to encode some forms of recursive definitions in `ntcc` as shown in [16]. Finally, the process $\star P$ represents an arbitrary long but finite delay for the activation of P. This process can be viewed as $P+\mathbf{next}\,P+\mathbf{next}^{\,2}P+\ldots$.

2.2 Structural Operational Semantics (SOS)

The SOS of `ntcc` consists of two kind of reductions: the internal transition ($\longrightarrow$) representing the evolution of processes during a time-unit and the observable transition ($\Longrightarrow$) representing the evolution of processes between time-units.

The internal transition relation $\gamma\longrightarrow\gamma'$ satisfies the rules in Figure 1. Here we follow the formulation in [10] where the local variables created by the program appear explicitly in the transition system and parallel composition of agents is identified as a multiset of agents. More precisely, a *configuration* γ is a triple of the form $(X;\Gamma;c)$, where c is a constraint representing the store, Γ is a multiset of processes, and X is a set of hidden (local) variables of c and Γ. The multiset $\Gamma=\{P_1,P_2,\ldots,P_n\}$ represents the process $P_1 \parallel P_2 \parallel \ldots \parallel P_n$. We shall indistinguishably use both notations to denote parallel composition of processes. Moreover, processes are quotiented by a structural congruence relation $\cong$ satisfying: (STR1) **local** $x(P)\cong$ **local** $y(P[y/x])$ if $y\notin fv(P)$ (alpha conversion); (STR2) $P \parallel Q\cong Q \parallel P$; (STR3) $P \parallel (Q \parallel R)\cong(P \parallel Q) \parallel R$. We shall write

$(X;\Gamma;c) \equiv (X';\Gamma';c')$ whenever $X = X'$, $\Gamma \cong \Gamma'$ and $c \equiv d$ (i.e., $c \vdash d$ and $d \vdash c$).

The rules in Figure 1 are straightforward realizing the operational intuitions given above: a tell agent $\mathbf{tell}(c)$ adds c to the current store d (Rule $\mathrm{R_{TELL}}$); the process $\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ executes P_i if its corresponding guard c_i can be entailed from the store (Rule $\mathrm{R_{ASK}}$); a local process $\mathbf{local}\ x(P)$ adds x to the set of hidden variables X when no clashes of variables occur (Rule $\mathrm{R_{LOC}}$). Observe that rule $\mathrm{R_{STR}}$ can be used to do alpha conversion if the premise $x \notin X \cup fv(d) \cup fv(\Gamma)$ does not hold; if the current store entails c, then the process P in $\mathbf{unless}\ c\ \mathbf{next}\ P$ is not executed (Rule $\mathrm{R_{UNL}}$); the seemingly missing rule for the process $\mathbf{next}\ P$ is given by the Rule $\mathrm{R_{OBS}}$ as explained below; the process $!\,P$ generates a copy of P and then, it is executed again in the next time-unit (Rule $\mathrm{R_{REP}}$); for a given $n \geq 0$, the process $\star\, P$ executes $\mathbf{next}^n\ P$ (Rule $\mathrm{R_{STAR}}$).

Let us now describe the rule for the observable transitions. Rule $\mathrm{R_{OBS}}$ says that an observable transition from P labeled with $(c, \exists X.d)$ is obtained by performing a terminating sequence of internal transitions from $(\emptyset;\Gamma;c)$ to $(X;\Gamma';d)$, for some Γ'. The process to be executed in the next time interval corresponds to the future of Γ' (i.e., $F(\Gamma')$) as shown in Figure 1. Note that the future function F is not defined for the processes $\mathbf{tell}(c)$, $\mathbf{local}\ x(P)$, $!\,P$ and $\star\, P$ since all these processes have an internal transition. Moreover, the variables in X are hidden by using existential quantification, i.e., the information about X is not visible from the final store d.

Definition 2.3 (Observable Behavior) *Let P be a process and $s = c_1.c_2.c_3\cdots$ be an infinite sequence of constraints. We say that P outputs $s' = c'_1.c'_2.c'_3\cdots$ under input s if*

$$P \equiv P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} P_3 \xrightarrow{(c_3,c'_3)} \cdots$$

and we write $P \xrightarrow{(s,s')}$. We define the input-output behavior of P as the set $io(P) = \{(s,s') \mid P \xrightarrow{(s,s')}\}$.

3 Symbolic Model of `ntcc` Processes

Model Checking [6] (see a survey in [19]) is a well established technique for the automatic verification of systems. In this section, we show how to construct a symbolic, and then compact, model of the behavior of a `ntcc` process. Later, in Section 5, we shall use this model as input to a symbolic model checking algorithm.

One of the main difficulties to develop automatic verification techniques for `ntcc` programs is the fact that the semantics of processes is given by two different transition systems, namely, the internal ($\longrightarrow$) and the observable ($\Longrightarrow$) transitions. On one hand, building a model for the internal transition seems to be unnecessary since the internal movements of a process during a time-unit are unobservable from the external environment. Moreover, abstracting away from the internal transition should lead to a more compact representation of the system, thus reducing the search space. On the other hand, the internal transition dictates much of the observable behavior when non-deterministic processes are considered (see e.g., Rules

$\mathrm{R_{ASK}}$ and $\mathrm{R_{STAR}}$). Our approach is then to use (temporal) formulas as a compact representation of the reachable states (i.e., stores) of a process. As we shall see, the proposed formulas capture the *observable* contributions (i.e., constraints) that processes can make to the final store; additionally, the internal (*unobservable*) transitions are symbolically captured by logical connectives. More precisely, we shall follow the steps below:

- **Step 1:** Give a logical interpretation of P (Definition 3.2). The interesting cases will be the temporal operators ! and $\star$ that require a fixpoint characterization.
- **Step 2:** Perform a fixpoint computation to find a formula that models, symbolically, the reachable states of P.
- **Step 3:** Deal with dead-ends, i.e., states without any transition.

3.1 Step 1: Logical Interpretation of Processes

We start by introducing some needed notation. The behavior of a process will be specified as a disjunction of formulas of the shape

$$\circ^0(c_0) \wedge \circ^1(c_1) \wedge \cdots \wedge \circ^n(c_n) \tag{1}$$

where each c_i is a constraint (Definition 2.1). Intuitively, the above formula reads as "c_0 *is valid in the current state and, after* i *observable transitions,* c_i *holds*". The "$\circ$" symbol corresponds to the *next* modality in Linear Time Temporal Logic (LTL) [14] as described below. For the sake of readability, we write c instead of $\circ^0(c)$ and $\circ(c)$ instead of $\circ^1(c)$. Moreover, we write

$$\{\{F_1^1, F_2^1, \cdots, F_{n_1}^1\}, \{F_1^2, F_2^2, \cdots, F_{n_2}^2\}, \cdots, \{F_1^m, F_2^m, \cdots, F_{n_m}^m\}\}$$

instead of the following formula in disjunctive normal form

$$(F_1^1 \wedge F_2^1 \wedge \cdots \wedge F_{n_1}^1) \vee (F_1^2 \wedge F_2^2 \wedge \cdots \wedge F_{n_2}^2) \vee \cdots \vee (F_1^m \wedge F_2^m \wedge \cdots \wedge F_{n_m}^m) \tag{2}$$

Definition 3.1 (States) *We shall use $\mathcal{C}^\circ$ to denote the set of formulas built from the set of constraints $\mathcal{C}$ and the LTL operator $\circ$ (next). A state is a conjunction of $\mathcal{C}$ formulas of the shape $c_1 \wedge \cdots \wedge c_m$. Two states A and B are equivalent if $A \vdash B$ and $B \vdash A$. A $\mathcal{C}^\circ$ formula of the shape $F = A_0 \wedge \circ(A_1) \wedge \cdots \wedge \circ^n(A_n)$ represents a label transition system (LTS) where there is a transition from state s_x to state s_y, notation $s_x \rightsquigarrow s_y$, if A_i (resp. A_{i+1}) holds in s_x (resp. s_y). We shall use $\mathcal{L}(F)$ to denote such an LTS. Given an LTS L, we shall use $state(L)$ (resp. $trans(L)$) to denote the set of states (resp. transitions) of L.*

Now we are ready to give logical meaning to processes by using $\mathcal{C}^\circ$ formulas.

Definition 3.2 (Symbolic Representation) *Given a* `ntcc` *process P, let $\mathcal{S}(P)$ be inductively defined as in Figure 2 where μ (resp. ν) represents the least (resp. greatest) fixpoint operator in the complete lattice $\langle \mathcal{L}(\mathcal{C}^\circ), \leq \rangle$ where $L_1 \leq L_2$ iff $state(L_1) \subseteq state(L_2)$ and $trans(L_1) \subseteq trans(L_2)$.*

$$\mathcal{S}(\mathbf{tell}(c)) = c \qquad \mathcal{S}(\textstyle\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i) = \bigwedge_{i\in I}(\neg c_i) \vee \bigvee_{i\in I}(c_i \wedge \mathcal{S}(P_i))$$

$$\mathcal{S}(P \parallel Q) = \mathcal{S}(P) \wedge \mathcal{S}(Q) \qquad \mathcal{S}(\mathbf{local}\ x(P)) = \exists x.(\mathcal{S}(P))$$

$$\mathcal{S}(\mathbf{next}\ P) = \circ(\mathcal{S}(P)) \qquad \mathcal{S}(\mathbf{unless}\ c\ \mathbf{next}\ P) = (\neg c \wedge \circ(\mathcal{S}(P))) \vee c$$

$$\mathcal{S}(\star P) = \mu Y.(\mathcal{S}(P) \vee \circ(Y)) \qquad \mathcal{S}(!\,P) = \nu Y.(\mathcal{S}(P) \wedge \circ(Y))$$

Fig. 2. Symbolic representation of `ntcc` processes (Definition 3.2). $\neg c$ denotes the *absence* of c.

Let us give some intuitions about the previous definition. A process $\mathbf{tell}(c)$ defines a state where c holds. A process $\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ generates a state where none of the guards hold ($\bigwedge_{i\in I}(\neg c_i)$) and states where c_i and $\mathcal{S}(P_i)$ hold. A process $P \parallel Q$ defines states where both $\mathcal{S}(P)$ and $\mathcal{S}(Q)$ hold. A local process $\mathbf{local}\ x(P)$ generates a state where P holds but the information about x is irrelevant. A process $\mathbf{unless}\ c\ \mathbf{next}\ P$ defines a state where c holds (and then P is not executed) and a state where c is absent (i.e., $\neg c$) and the states generated from P hold.

As shown in [16], the process $\star P$ resembles the eventually modality ($\Diamond$) in LTL. Then, the states generated by this process can be characterized as the least fixpoint of the disjunction of a state where P holds and a state where P holds in the next time-unit. Similarly, the process $!\,P$ resembles the always ($\Box$) modality in LTL. Then, the generated states correspond to the greatest fixpoint of the conjunction of a state satisfying P and a future state where P also holds.

3.2 Step 2: Fixpoint Computation

Once we have the logical reading $\mathcal{S}(P)$ of a given process P, we need to perform a fixpoint computation in order to obtain a $\mathcal{C}^\circ$ formula representing symbolically the states of the system. Before giving some examples of this step, we need to define the degree of a formula (Notation 3.3 below) and a simple program transformation in order to capture correctly the state transitions.

Consider the process $P = \mathbf{next}\,\mathbf{tell}(c)$. We know that $\mathcal{S}(P) = \circ(c)$. Then, what should be the observable behavior of P during the first time-unit? We know that P does not add any information to the first time-unit. Then, we need to "complete" the formula $\circ(c)$ to model the fact that P generates two states: s_1 where no information can be deduced and s_2 where c holds such that $s_1 \rightsquigarrow s_2$. We shall represent symbolically this situation as the formula $\mathtt{true} \wedge \circ(c)$.

Notation 3.3 (Empty States and Degrees) *Let $F = \circ^0(c_0) \wedge \circ^1(c_1) \wedge \cdots \wedge \circ^n(c_n)$, we shall say that the degree of F, notation degree(F), is n. We shall assume that for every $i \in 0..n$ in F, there exists c_i such that $\circ^i(c_i)$ occurs in F; in other case we assume that $c_i = \mathtt{true}$. For the sake of readability, we shall omit the $\mathtt{true}$ constraint and we shall write, for instance, $\circ^2(c)$ instead of $\mathtt{true} \wedge \circ(\mathtt{true}) \wedge \circ^2(c)$.*

Now we introduce a simple program transformation needed to correctly capture the state transitions during the fixpoint computation. Let us explain the need of

$$\begin{array}{lll}
\mathcal{P}(\mathbf{tell}(c), n) & = \mathbf{tell}(c \wedge \mathtt{st}_n) & \mathcal{P}(\sum_{j\in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j, n) = \sum_{j\in J} \mathbf{when}\ c_j \wedge \mathtt{st}_n\ \mathbf{do}\ \mathcal{P}(P_j, n) \\
\mathcal{P}(P \parallel Q, n) & = \mathcal{P}(P, n) \parallel \mathcal{P}(Q, n) & \mathcal{P}(\mathbf{local}\ x(P), n) = \mathbf{local}\ x(\mathcal{P}(P, n)) \\
\mathcal{P}(\mathbf{next}\, P, n) & = \mathbf{next}\, \mathcal{P}(P, n+1) & \mathcal{P}(\mathbf{unless}\ c\ \mathbf{next}\, P, n) = \mathbf{unless}\ c \wedge \mathtt{st}_n\ \mathbf{next}\, \mathcal{P}(P, n+1) \\
\mathcal{P}(\star P, n) & = \star \mathcal{P}(P, n) & \mathcal{P}(!\, P, n) = !\, \mathcal{P}(P, n)
\end{array}$$

Fig. 3. Labeling (see Definition 3.4).

such transformation with a simple example. Assume the processes P and Q below:

$$P = \mathbf{tell}(c) \parallel \mathbf{next}\, \mathbf{tell}(c) \qquad\qquad Q = !\, \mathbf{tell}(c)$$

We know that $\mathcal{S}(P) = c \wedge \circ(c)$. Moreover, $c \wedge \circ(c)$ is also a solution for the equation $\mathcal{S}(Q)$. In the first case, we want to represent the LTS where there are two different states s_1 and s_2, s_1 goes to s_2 ($s_1 \rightsquigarrow s_2$) and c holds in both states. In the second case, we want to represent an LTS with a single state s_3 where c holds and, there is a loop in s_3 ($s_3 \rightsquigarrow s_3$). Hence, how can we distinguish between the formula $\mathcal{S}(P)$ and the solution of $\mathcal{S}(Q)$? The idea is to *label* the constraints in order to specify that s_1 and s_2 above are different and there is a temporal dependency (transition) between them. For instance, in the case of $\mathcal{S}(P)$ we shall produce a formula of the shape $c_1 \wedge \circ(c_2)$ to distinguish the two occurrences of c in P. The labeling process is a simple program transformation as shown in the next definition.

Definition 3.4 (Labeling) *Without loss of generality, we assume that for each $i \in \mathbb{N}$, $\mathtt{st}_i$ is a distinguished atomic constraint in the constraint system. Given a process P, we define inductively $\mathcal{P}(P, n)$ as in Figure 3. To simplify the notation, we shall write c_n instead of $c \wedge \mathtt{st}_n$. Moreover, instead of c_0 we shall write c.*

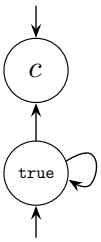
The labeling process is also needed to produce a formula of the shape $\mathtt{true}_0 \wedge \circ(\mathtt{true}_1) \wedge \circ^2(c_2)$ instead of $\mathtt{true} \wedge \circ(\mathtt{true}) \wedge \circ^2(c)$ when the formula $\mathtt{true} \wedge \circ(\mathtt{true})$ is added to $\circ^2(c)$ as explained in Notation 3.3. This avoids, for instance, the unwanted loop $\mathtt{true} \rightsquigarrow \mathtt{true}$ when computing the model of a process of the shape $\mathbf{next}\, \mathbf{next}\, \mathbf{tell}(c)$.

Now we are ready to show the fixpoint procedure. The idea is to compute the LTS that satisfies the equation $\mathcal{S}(\mathcal{P}(P, 0))$. Recall that every state satisfies $\mathtt{true}$ and the constraint $\mathtt{false}$ only holds in an inconsistent store. Therefore, as standardly done, the computation for a solution of the equation $\mu Y.(F \vee \circ(Y))$ (resp. $\nu Y.(F \wedge \circ(Y))$) starts with $Y_0 = \mathtt{false}$ (resp. $Y_0 = \mathtt{true}$). The following example finds the symbolic model for the process $!\, \star\, \mathbf{tell}(c)$ that requires both, a least and a greatest fixpoint computation.

Example 3.5 *Let $P = !\, \star\, \mathbf{tell}(c)$. We start by computing $\mathcal{S}(\star \mathbf{tell}(c)) = \mu X.(\mathcal{S}(\mathbf{tell}(c)) \vee \circ(X))$ as depicted in Figure 4a. Note that both X_3 and X_4 represent the transition system in the same figure. Then X_3 is the fixpoint and we can use it to compute the meaning of "!", i.e., $\nu Y.(X_3 \wedge \circ(Y))$ as shown in Figure 4b. Y_2 is a solution for the equation $\mathcal{S}(P)$ and it represents the LTS in Figure 4b.*

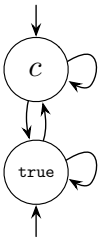
The reader may wonder whether the fixpoint computation stops in a finite number of steps in the presence of replicated ($!\, P$) processes. The following theorem

$$
\begin{aligned}
X_0 &= \texttt{false}\\
X_1 &= c \vee \circ(\texttt{false}) \equiv c\\
X_2 &= \{\{c\}, \{\circ(c)\}\}\\
X_3 &= \{\{c\}, \{\circ(c)\}, \{\circ^2(c)\}\}\\
X_4 &= \{\{c\}, \{\circ(c)\}, \{\circ^2(c)\}, \{\circ^3(c)\}\}
\end{aligned}
$$



(a) Symbolic model for $\star\textbf{tell}(c)$. $X_0 = \texttt{false}$ since we are computing a solution for $\mu X.(\mathcal{S}(\textbf{tell}(c)) \vee \circ(X))$ (a least fixpoint).

$$
\begin{aligned}
Y_0 &= \texttt{true}\\
Y_1 &= \{\{c\}, \{\circ(c)\}, \{\circ^2(c)\}, \{\circ(\texttt{true})\}\}\\
Y_2 &= \{\{c, \circ(c)\}, \{c, \circ^2(c)\}, \{c, \circ^3(c)\}, \{\circ(c)\}, \{\circ(c), \circ^2(c)\},\\
&\quad \{\circ(c), \circ^3(c)\}, \{\circ^2(c)\}, \{\circ^2(c), \circ^3(c)\}\}
\end{aligned}
$$



(b) Symbolic model for $!\star\textbf{tell}(c)$.

Fig. 4. Label transition systems for the Example 3.5.

answers positively that question.

Theorem 3.6 *Let P be a process and $\mathcal{S}(P) = F(X_1, \ldots, X_n)$ be a formula where the variables $X_1, \ldots, X_n$ occur in F preceded by either μ or ν. The fixpoint of F can be reached in a finite number of steps.*

Proof. The proof is a direct corollary from: (1) [22, Theorem 4.12] that shows that the output of P can be characterized by a finite subset of $\mathcal{C}$ (which is not necessarily finite); and (2) [22, Lemma 4.13] that shows that the number of different states P may generate is also finite. Hence, since the number of possible reachable states is the LTS $\mathcal{L}(\mathcal{S}(P))$ is finite, the fixpoint computation must terminate. □

3.3 Dead-ends

After the fixpoint computation in the previous step, it may be the case that the resulting LTS has *dead-ends*, i.e., states without outgoing transitions. This happens when the process P is not a replicated ("!") process. As a matter of example, consider the process $P = \textbf{tell}(c)$ whose resulting LTS has a unique state c without transitions. We recall that processes in `ntcc` are supposed to react continuously with the environment. Then, in the case of $\textbf{tell}(c)$, the process outputs c in the first time-unit and `true` in the subsequent time-units. Note that this behavior is in accordance with the operational semantics in Definition 2.3: the outputs of a process are always infinite sequences of constraints.

Hence, given a $\mathcal{C}^\circ$ formula F of degree n representing an LTS with dead-end

$$
\begin{aligned}
Y_0 &= \mathtt{true}\\
Y_1 &= \{\{\mathtt{signal}, \circ(\mathtt{on}_1)\}, \{\neg(\mathtt{signal}), \circ(\mathtt{off}_1)\}\}\\
Y_2 &= \{\{\mathtt{signal}, \circ(\mathtt{on}_1), \circ(\mathtt{signal}), \circ^2(\mathtt{on}_1)\}, \{\mathtt{signal}, \circ(\mathtt{on}_1), \circ(\neg\mathtt{signal}), \circ^2(\mathtt{off}_1)\},\\
&\quad \{\neg\mathtt{signal}, \circ(\mathtt{off}_1), \circ(\mathtt{signal}_1), \circ^2(\mathtt{on}_1)\}, \{\neg\mathtt{signal}, \circ(\mathtt{off}_1), \circ(\neg\mathtt{signal}), \circ^2(\mathtt{off}_1)\}\}\\
Y_3 &= \{\{\mathtt{signal}, \circ(\mathtt{signal}), \circ(\mathtt{on}_1), \circ^2(\mathtt{on}_1), \circ^2(\mathtt{signal}), \circ^3(\mathtt{on}_1)\},\\
&\quad \{\mathtt{signal}, \circ(\mathtt{signal}), \circ(\mathtt{on}_1), \circ^2(\mathtt{on}_1), \circ^2(\neg\mathtt{signal}), \circ^3(\mathtt{off}_1)\},\\
&\quad \{\mathtt{signal}, \circ(\neg\mathtt{signal}), \circ(\mathtt{on}_1), \circ^2(\mathtt{off}_1), \circ^2(\mathtt{signal}), \circ^3(\mathtt{on}_1)\},\\
&\quad \{\mathtt{signal}, \circ(\neg\mathtt{signal}), \circ(\mathtt{on}_1), \circ^2(\mathtt{off}_1), \circ^2(\neg\mathtt{signal}), \circ^3(\mathtt{off}_1)\}\\
&\quad \{\neg\mathtt{signal}, \circ(\mathtt{signal}), \circ(\mathtt{off}_1), \circ^2(\mathtt{on}_1), \circ^2(\mathtt{signal}), \circ^3(\mathtt{on}_1)\},\\
&\quad \{\neg\mathtt{signal}, \circ(\mathtt{signal}), \circ(\mathtt{off}_1), \circ^2(\mathtt{on}_1), \circ^2(\neg\mathtt{signal}), \circ^3(\mathtt{off}_1)\},\\
&\quad \{\neg\mathtt{signal}, \circ(\neg\mathtt{signal}), \circ(\mathtt{off}_1), \circ^2(\mathtt{off}_1), \circ^2(\mathtt{signal}), \circ^3(\mathtt{on}_1)\},\\
&\quad \{\neg\mathtt{signal}, \circ(\neg\mathtt{signal}), \circ(\mathtt{off}_1), \circ^2(\mathtt{off}_1), \circ^2(\neg\mathtt{signal}), \circ^3(\mathtt{off}_1)\}\}
\end{aligned}
$$

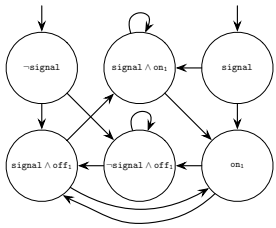


Fig. 5. Transition system from Example 3.7.

states, we shall add to F (in conjunction) the states $\circ^{n+1}(\mathtt{true}_{n+1}) \wedge \circ^{n+2}(\mathtt{true}_{n+1})$. Therefore, the dead-ends of F have a transition to a looping state where only `true` can be deduced.

Example 3.7 (Control System) *Assume a simple control system that must emit the signal* ***on*** *in the next time-unit when the environment reports a given signal* ***signal*** *in the current time-unit. Otherwise, it must emit the signal* ***off*** *in the next time-unit. This can be modeled as the process*

$$P = !\,(\textbf{when } \mathit{signal} \textbf{ do next tell}(\mathit{on}) \parallel \textbf{unless } \mathit{signal} \textbf{ next tell}(\mathit{off}))$$

The symbolic model of P results from the formula $\mathcal{S}(\mathcal{P}(P, 0)) =$

$$\nu Y.(((\mathit{signal} \wedge \circ(\mathit{on}_1) \vee \neg(\mathit{signal})) \wedge (\neg(\mathit{signal}) \wedge \circ(\mathit{off}_1) \vee \mathit{signal})) \wedge \circ(Y))$$

and the fixpoint computation leads to the results in Figure 5.

Example 3.8 (Asynchronous Behavior) *Consider now a control system that must emit the signal* ***stop*** *once an error is detected. Moreover, we know that the system is doomed to fail (due to the process $\star$***tell**(error) *below):*

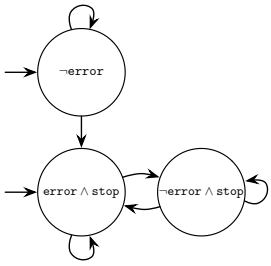
$$P = \star\textbf{tell}(\mathit{error}) \parallel !\,\textbf{when } \mathit{error} \textbf{ do } !\,\textbf{tell}(\mathit{stop})$$

Note that as soon as the ***error*** *signal is detected, the ask process executes the process* **!tell**(stop) *and then, the constraint* ***stop*** *can be deduced from that time interval on. The symbolic model of $\star$***tell**(error) *is given by the formula:*

$$F_1 = \mathit{error} \vee \circ(\mathit{error}) \vee \circ^2(\mathit{error})$$

that determines an LTS similar to that of Figure 4a.

The symbolic model of the process **!tell**(stop) *is* ***stop*** $\wedge \circ($***stop***$)$ *which determines an LTS such that a state where* ***stop*** *holds is always followed by another state where* ***stop*** *also holds. The symbolic model of P and its corresponding LTS is shown in Figure 6.*



$Y_2 = \{\{\texttt{error}, \texttt{stop}, \circ(\texttt{stop}), \circ(\texttt{error}), \circ^2(\texttt{stop}), \circ^2(\texttt{error})\},$
$\{\texttt{error}, \texttt{stop}, \circ(\texttt{stop}), \circ(\texttt{error}), \circ^2(\texttt{stop}), \circ^2(\neg(\texttt{error}))\},$
$\{\texttt{error}, \texttt{stop}, \circ(\texttt{stop}), \circ(\neg(\texttt{error})), \circ^2(\texttt{stop}), \circ^2(\texttt{error})\},$
$\{\texttt{error}, \texttt{stop}, \circ(\texttt{stop}), \circ(\neg(\texttt{error})), \circ^2(\texttt{stop}), \circ^2(\neg(\texttt{error}))\},$
$\{\neg(\texttt{error}), \circ(\texttt{stop}), \circ(\texttt{error}), \circ^2(\texttt{stop}), \circ^2(\texttt{error})\},$
$\{\neg(\texttt{error}), \circ(\texttt{stop}), \circ(\texttt{error}), \circ^2(\texttt{stop}), \circ^2(\neg(\texttt{error}))\},$
$\{\neg(\texttt{error}), \circ(\neg(\texttt{error})), \circ^2(\texttt{stop}), \circ^2(\texttt{error})\},$
$\{\neg(\texttt{error}), \circ(\neg(\texttt{error})), \circ^2(\neg(\texttt{error}))\}\}$

Fig. 6. Symbolic model and LTS from Example 3.8.

We conclude this section by showing that our symbolic construction is correct. Recall that c_i means $c \wedge \texttt{st}_i$ and $\neg c$ means that c is absent. Since those constraints were introduced during the model construction procedure (and they do not make part of the original process), the correctness result can safely ignore those constraints. Recall also that the resulting LTS does not have dead-ends, i.e., paths in it are infinite sequences of states.

Theorem 3.9 (Correctness) *Let P be a process, F a solution for the equation $\mathcal{S}(\mathcal{P}(P, 0))$ and L be the LTS $\mathcal{L}(F)$ as in Definition 3.1. Consider an infinite sequence of constraints π. Then, π is a path in L iff there exists a sequence $\pi^i = c_1.c_2.c_3.\cdots$ such that $(\pi^i, \pi^o) \in io(P)$ where π^o is like π but without any occurrence of constraints of the shape $\texttt{st}_i$ and $\neg c$.*

Proof. The proof proceeds by induction on the structure of P. For the $\Rightarrow$ part, assume that π is a path in the LTS $\mathcal{L}(F)$. We shall show that the corresponding π^o is indeed an output of P (for a given input π^i). The interesting cases are those of the temporal constructs:
- $P = \mathbf{next}\, Q$. It is easy to see that π', defined as π without the first element, is a path for the LTS $\mathcal{L}(\mathcal{S}(Q))$. By induction, there exists a π'^o which is an output of Q. Hence, it is easy to see that π^o is indeed an output of P.
- $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$. If $\pi(1)$ (the first element of π) is a state where c holds, the proof is trivial. If c does not hold in $\pi(1)$, then we proceed as in the **next** case.
- $P = !\, Q$. We know that F is a solution for the equation $G(X) = \mathcal{S}(Q) \wedge \circ X$. Also, by induction, we know that any path π_q in the LTS $L_q = \mathcal{L}(\mathcal{S}(Q))$ corresponds to an output π_q^o of Q. Hence, any path starting in one of the initial states in L_q (and also in $\mathcal{L}(F)$) corresponds to an output of Q. Furthermore, since F is a solution for $G(X)$, any suffix of π corresponds also to an output of Q. Since all the suffixes of π (including π itself) are in the output of Q, we conclude that π^o is an output of P.
- $P = \star Q$. Note that F is a solution for $G(X) = \mathcal{S}(Q) \vee \circ X$. If we consider only *fair* paths in the LTS (i.e., π is not an infinite sequence where a Q-state is never visited) then there exists a suffix π' of π such that π'^o corresponds to an output of Q. By induction we can conclude that π^o corresponds to an output of P.

The $\Leftarrow$ side of the proof is analogous. □

As we shall see in Section 5, the *fairness* condition needed to prove the case $\star Q$

$\langle \beta, i\rangle \models \dot{\texttt{true}}$

$\langle \beta, i\rangle \models c$ iff $\beta(i) \models c$

$\langle \beta, i\rangle \models \circ F$ iff $\langle \beta, i+1\rangle \models F$

$\langle \beta, i\rangle \models \Diamond F$ iff $\exists_{j \geq i}\ s.t.\ \langle \beta, j\rangle \models F$

$\langle \beta, i\rangle \models F_1 \dot{\wedge} F_2$ iff $\langle \beta, i\rangle \models F_1$ *and* $\langle \beta, i\rangle \models F_2$

$\langle \beta, i\rangle \models F_1 \dot{\vee} F_2$ iff $\langle \beta, i\rangle \models F_1$ *or* $\langle \beta, i\rangle \models F_2$

$\langle \beta, i\rangle \not\models \dot{\texttt{false}}$

$\langle \beta, i\rangle \models \dot{\neg} F$ iff $\langle \beta, i\rangle \not\models F$

$\langle \beta, i\rangle \models \Box F$ iff $\forall_{j \geq i}\, \langle \beta, j\rangle \models F$

Fig. 7. Semantics of CLTL formulas.

is guaranteed by the model checking algorithm that considers *fairness* constraints.

4 The Language of Properties

Constraint Temporal Logic (CLTL). Since `ntcc` processes manipulate constraints, it is reasonable to think that system properties must be stated in a logic able to deal with constraints. Hence, we shall use CLTL [16], a Linear Time Temporal Logic [14] where atomic formulas are constraints. Formulas in propositional CLTL are built from the grammar below:

$$F ::= \dot{\texttt{true}} \mid \dot{\texttt{false}} \mid c \mid F \dot{\wedge} F \mid F \dot{\vee} F \mid \dot{\neg} F \mid \circ F \mid \Diamond F \mid \Box F$$

where c is a constraint. $\dot{\texttt{true}}$, $\dot{\texttt{false}}$, $\dot{\wedge}$, $\dot{\vee}$ and $\dot{\neg}$ represent the linear-temporal logic true, false, conjunction, disjunction and negation respectively. These symbols should not be confused with their counterpart in the constraint system (i.e., `true`, `false` and $\wedge$). Symbols $\circ$, $\Box$ and $\Diamond$ denote the LTL temporal operators *next*, *always* and *eventually*.

The interpretation structures of formulas in CLTL are infinite sequences of constraints (as the observable behavior in Definition 2.3). We say that the infinite sequence of constraints β is a model of (or that it satisfies) a formula F, notation $\beta \models F$, if $\langle \beta, 1\rangle \models F$. The meaning of $\langle \beta, i\rangle \models F$ is given in Figure 7.

While the semantics of CLTL is given by sequences of constraints, models of LTL formulas are sequences of states (maps assigning values to variables). The relation between satisfiability in CLTL and standard LTL [14] was established in [22, Lemma 5.4]: A formula F is LTL satisfiable iff $F \dot{\wedge} \Box\dot{\neg}\dot{\texttt{false}}$ is CLTL satisfiable. Intuitively, the formula `false` (the constraint representing the inconsistent store) has at least one model (e.g., the sequence of constraints `false.false`...) while $\dot{\texttt{false}}$ does not have any model ($\langle \beta, i\rangle \not\models \dot{\texttt{false}}$). Then, the "$\Box\dot{\neg}\dot{\texttt{false}}$" part gets rid of the CLTL models containing (the constraint) `false`. This result holds when negation has atomic scope, i.e., G must be an atom (i.e., a constraint) in all formula of the shape $\dot{\neg} G$. This is known in [22] as the *restricted-negation formula* condition.

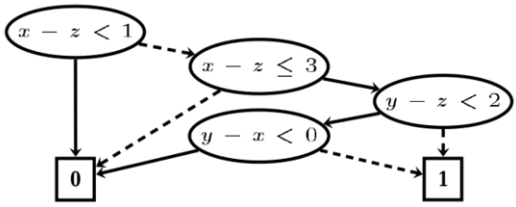


Fig. 8. *Example of a DDD structure. Image and example extracted from [15].*

4.1 Representation of Constraints

Many of the systems modeled in the `ntcc` calculus make use of numerical constraints (see e.g., [17]). Hence, we use Difference Decision Diagrams (DDD) [15], a suitable extension of Binary Decision Diagrams (BDD) [4] to represent difference constraint expressions built from the syntax below:

$$\phi ::= false \mid true \mid x - y < c \mid x - y \leq c \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \\ \mid \phi_1 \longrightarrow \phi_2 \mid \phi_1 \longleftrightarrow \phi_2 \mid \exists_x.\phi \mid \forall_x.\phi$$

A DDD can be seen as a directed acyclic graph where the set of vertex contains the terminals 0 and 1 and the non-terminals are ϕ formulas. The set of edges is given by the so-called high and low edges, $(v, high(v)), (v, low(v))$, for each non-terminal vertex. These edges represent the path taken by the DDD in case that the constraint in the vertex v holds or not. As an example, the Figure 8 shows the DDD corresponding to the expression $\phi = 1 \leq x - z \leq 3 \wedge (y - z \geq 2 \vee y - x \geq 0)$.

DDDs share a large number of features with BDDs. As BDDs, DDDs need to be ordered and reduced, but in the case of DDDs, it is more difficult to obtain a canonical representation of the boolean formula. To deal with this problem, in [15] the authors propose the use of *path reduced DDDs*, with the aim to obtain a semi-canonical data structure, thus reducing the complexity of handling constraints. In fact, most of the operations on DDDs run in constant time.

5 Symbolic Model Checking

In Section 3 we saw how to build a symbolic model which is a compact representation of the behavior of a `ntcc` process. In Section 4 we recalled the CLTL logic able to express temporal properties of `ntcc` processes. The last step is to use standard techniques from symbolic model checking to verify if a process satisfies a given property. This is done by combining the model of the system and the formula to be proved. In the following we give the relevant details to perform this step.

Recall from Section 4 that the satisfiability problem of CLTL can be reduced to the same problem in LTL. Moreover, as it was shown in [5], the model checking problem for LTL can be solved by reducing it to the symbolic model checking problem for Computation Tree Logic (CTL) [9] with *fairness constraints*. Then, we can use all the machinery and tools developed for CTL model checking to verify programs written in `ntcc`.

In CTL, unlike LTL, the temporal operator must be preceded by a path

quantifier. Such quantifiers define where the temporal formulas must hold in the computation tree: the quantifier **A**, stands for "every path", and **E** stands for "there exists a path". The temporal operators to build CTL formulas are: $\circ G$, which means that G holds at the next time; and $G\mathbf{U}H$, which means that G holds until H holds. We recall that $\Diamond F$ can be defined as $\texttt{true}\mathbf{U}F$ and $\Box F$ as $\neg\Diamond\neg F$. Hence, in the following, we shall only use the temporal constructs "$\circ$" and "**U**".

The algorithm. Given a process P and a CLTL property ϕ, we proceed as follows:

1. Obtain the DDD representation $\mathcal{M}$ for the model of the process P.
2. Compute the DDD representation $\mathcal{T}$ of the tableau for the (negated) formula $\psi = \neg(\phi \wedge \Box\dot{\neg}\texttt{false})$.
3. Build the set $\mathcal{F}$ with all the fairness constraints, i.e., all the subformulas in ψ containing the **U** operator.
4. Obtain the product $\mathcal{P}$ (through DDD operations) between the model $\mathcal{M}$ and the tableau of $\mathcal{T}$.
5. Apply the CTL symbolic model checking algorithm with fairness constraints $\mathcal{F}$ over the symbolic product $\mathcal{P}$ and the property $\mathbf{E}\texttt{true}$.
6. If the algorithm returns an empty set of states (satisfying the negated property), then P satisfies ϕ; otherwise, the algorithm returns the set of states satisfying the formula ψ as counterexamples.

Let us elaborate on the above steps. First we build the symbolic model of P as explained in Section 3. Then, as shown in [5], the states $\sigma, \sigma' \ldots$ in the tableau $\mathcal{T}$ correspond to $pow(el(\psi))$ where $el(\psi)$ (the set of elementary formulas of ψ) is:

- $el(F) = F$ if F is an atomic formula.
- $el(\neg F) = el(F)$.
- $el(F \vee G) = el(F) \cup el(G)$.
- $el(\circ F) = \circ F \cup el(F)$.
- $el(F\mathbf{U}G) = \circ(F\mathbf{U}G) \cup el(F) \cup el(G)$.

The transition relation $\mathcal{T}$ relies on the definition of the function $sat(\cdot)$ that maps subformulas into sets of states:

- $sat(G) = \{\sigma \mid G \in \sigma\}$ if $G \in el(\psi)$.
- $sat(\neg F) = \{\sigma \mid \sigma \notin sat(F)\}$.
- $sat(F \vee G) = sat(F) \cup sat(G)$.
- $sat(F\mathbf{U}G) = sat(G) \cup sat(F) \cap sat(\circ(F\mathbf{U}G))$.

Since we are dealing with formulas in LTL, the transition relation of $\mathcal{T}$ satisfies the following condition for any state σ: $\sigma \in sat(\circ F)$ iff $\sigma' \in sat(F)$ for all successor σ' of σ.

The set $\mathcal{F}$ of fairness constraints corresponds simply to the set of subformulas of the shape $F\mathbf{U}G$ in ψ. The set of fairness constraints is needed for the model checking algorithm to guarantee that in any path π where the formula $F\mathbf{U}G$ holds, it must be the case that G eventually holds. This situation can be also explained

from the point of view of processes. Consider the process $P = \star\mathbf{tell}(\texttt{fail})$ and the formula $\phi = \Box\dot{\neg}\texttt{fail}$. We know that the model of P contains a loop where `fail` does not hold (due to the loop $\texttt{true} \rightsquigarrow \texttt{true}$ in the LTS). Then, without fairness constraints, we would be able to prove that P satisfies ϕ, which is wrong.

The product $\mathcal{P}$ is simply obtained by operations on DDD (see, e.g., [3]). This product corresponds to the model where the negation of ϕ (i.e., ψ) holds and also the model of the program. Then, by running the symbolic model checking algorithm with fairness constraints [4,19] on the formula $\mathbf{E}\texttt{true}$, if the resulting set of paths is empty, P satisfies the property ϕ, otherwise, we can exhibit a counterexample.

6 Concluding Remarks

In this paper we introduced a symbolic model to capture the behavior of `ntcc` processes. We showed that the internal and the observable transition relations in `ntcc` can be neatly captured as temporal formulas. Such a compact representation was shown to be adequate to use standard techniques in model checking to automatically verify concurrent systems programmed in `ntcc`.

We implemented a tool in OCaml (`http://ocaml.org`) to execute our verification process automatically. The power of functional programming, the compilation process and the type system of OCaml, made possible to quickly develop such prototype. Moreover, we provide to users a more friendly way for writing programs in `ntcc` by parsing their syntax with `ocamllex` and `menhir`.

The tool receives as input the `ntcc` program and recursively computes its symbolic representation. In order to carry out the verification, the tool compiles the symbolic model into the format of the model checker NuSMV (`http://nusmv.fbk.eu/NuSMV/`). Then, system properties can be verified on NuSMV. Moreover, the tool generates a PDF file with the LTS of the system as those shown in Section 3' figures.

We do not describe the implementation of our tool in depth here in order to give a higher priority to the technical aspects of our approach. The reader can find the details of the implementation as well as the execution of the examples described in this paper at `http://www.labri.fr/perso/jarias/symbolicMC`.

Related work. The ideas of this paper stem mainly from the works in [16] and [22]. In [16] it was shown that the `ntcc` constructs ! and $\star$ have a strong relation with the LTL temporal operators $\Box$ and $\Diamond$. Moreover, the duality of these operators as greatest and least fixpoints was studied to give a denotational semantics for `ntcc` processes based on closure operators. In [22] the author showed that the strongest postcondition of a process, notation ${}^{(}P)$, can be characterized as a Büchi automata. Roughly the set ${}^{(}P)$ contains all the possible outputs of P regardless the input. This fact was used in [22] to show that the verification problem for `ntcc` is decidable. We used this fact here to prove the Theorem 3.6. The construction of the Büchi automata in [22], however, has a non-elementary space complexity (i.e., the space complexity is worse than exponential). The construction we propose here is symbolic

thus ameliorating this situation: the states do not need to be explicitly enumerated and, using logical rules, some states can be reduced in early stages of the model construction.

Automatic verification techniques for languages based on the ccp model have been also studied in [1,2,12]. In those works, the target language is tccp [8]. Unlike ntcc, tccp does not consider constructs for asynchronous behavior ($\star P$ in ntcc). Moreover, the notion of time is identified with the time needed to ask and tell information to the store and the information in the store is carried through the time-units. Note that in ntcc the output in a time-unit is not related to the output in the previous time-unit. For this reason, the SOS of ntcc requires both an internal and an observable transition relation and thus, the above techniques for tccp cannot be used for the verification of ntcc programs.

Finally, we refer to the work in [18] where the authors use a symbolic representation based on LTL formulas to give meaning to temporal ccp processes. Such representation was proved useful to give meaning to processes engaging in divergent computations due to universally quantified asks (not present in ntcc).

Future work. Symbolic techniques in model checking aim at reducing the space and time needed to verify a given property, thus allowing for dealing with more complex systems [4,19]. However, the state explosion problem is unavoidable. In fact, the model checking algorithm for LTL is linear in the size of the model but exponential in the size of the formula to be verified. To mitigate this situation, we plan to provide tools for abstract debugging that allow the programmer to quickly find problems in her design before attempting the verification of more precise desirable properties. For that, we may rely on the abstract interpretation frameworks for the analysis of ccp programs that have been proposed in the literature (see e.g., [1,7,11,23]). Our idea is to use an abstraction of the constraint system in the lines of [11] in order to reduce the number of states generated by our technique.

Acknowledgment

We thank the anonymous reviewers for their detailed comments. The work of Arias has been supported by the ANR project OSSIA (ANR-12-CORD-0024).

References

[1] María Alpuente, María del Mar Gallardo, Ernesto Pimentel, and Alicia Villanueva. An abstract analysis framework for synchronous concurrent languages based on source-to-source transformation. *Electr. Notes Theor. Comput. Sci.*, 206:3–21, 2008.

[2] María Alpuente, Moreno Falaschi, and Alicia Villanueva. A symbolic model checker for tccp programs. In Nicolas Guelfi, editor, *RISE*, volume 3475 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2004.

[3] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[4] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[5] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at ltl model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[6] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[7] Marco Comini, Laura Titolo, and Alicia Villanueva. Abstract diagnosis for timed concurrent constraint programs. *TPLP*, 11(4-5):487–502, 2011.

[8] Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000.

[9] E Allen Emerson and Joseph Y Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of computer and system sciences*, 30(1):1–24, 1985.

[10] François Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Inf. Comput.*, 165(1):14–41, 2001.

[11] Moreno Falaschi, Carlos Olarte, and Catuscia Palamidessi. Abstract interpretation of temporal concurrent constraint programs. *TPLP*, Published online on 10 February 2014.

[12] Moreno Falaschi and Alicia Villanueva. Automatic verification of timed concurrent constraint programs. *TPLP*, 6(3):265–300, 2006.

[13] Gerhard Gentzen. Investigations into logical deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1969.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[15] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 1999.

[16] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.

[17] Carlos Olarte, Camilo Rueda, and Frank D. Valencia. Models and emerging trends of concurrent constraint programming. *Constraints*, 18(4):535–578, 2013.

[18] Carlos Olarte and Frank D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In Roger L. Wainwright and Hisham Haddad, editors, *SAC*, pages 145–150. ACM, 2008.

[19] Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.

[20] Vijay A. Saraswat and Martin C. Rinard. Concurrent constraint programming. In Frances E. Allen, editor, *POPL*, pages 232–245. ACM Press, 1990.

[21] Vijay A. Saraswat, Martin C. Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In David S. Wise, editor, *POPL*, pages 333–352. ACM Press, 1991.

[22] Frank D. Valencia. Decidability of infinite-state timed ccp processes and first-order ltl. *Theor. Comput. Sci.*, 330(3):577–607, 2005.

[23] Enea Zaffanella, Roberto Giacobazzi, and Giorgio Levi. Abstracting synchronization in concurrent constraint programming. *J. of Functional and Logic Programming*, 1997(6), 1997.