A Semantic Framework for the Abstract Model Checking of tccp programs [1]

María Alpuente [a] María del Mar Gallardo [b] Ernesto Pimentel [b] Alicia Villanueva [a]

[a] *DSIC, Technical University of Valencia Camino de Vera s/n, E-46022, Spain*

[b] *Dept. LCC, University of Málaga Campus de Teatinos s/n, E-29071, Spain*

Abstract

The *Timed Concurrent Constraint* programming language (tccp) introduces time aspects into the Concurrent Constraint paradigm. This makes tccp especially appropriate for analyzing timing properties of concurrent systems by model checking. However, even if very compact state representations are obtained thanks to the use of constraints in tccp, large state spaces can still be generated, which may prevent model-checking tools from verifying tccp programs completely. Model checking tccp programs is a difficult task due to the subtleties of the underlying operational semantics, which combines constraints, concurrency, non-determinism and time. Currently, there is no practical model-checking tool that is applicable to tccp. In this work, we introduce an abstract methodology which is based on over- and under-approximating tccp models and which mitigates the state explosion problem that is common to traditional model-checking algorithms. We ascertain the conditions for the correctness of the abstract technique and show that this preliminary abstract semantics does not correctly simulate the suspension behavior, which is a key feature of tccp. Then, we present a refined abstract semantics which correctly models suspension. Finally, we complete our methodology by approximating the temporal properties that must be verified.

1 Introduction

In the past few years, some extensions of the concurrent constraint paradigm [3,30] have been defined in order to model reactive systems. All these extensions introduce a quantitative notion of time that makes it possible to model the typical ingredients of these systems, such as timeouts, preemptions, etc. The automatic verification of systems specified in the timed concurrent constraint language tccp of [3] was first studied in [14]. Then, an exhaustive method for applying the classical model-checking technique to tccp was proposed in [15], which uses the temporal logic for reasoning about tccp programs of [4]. The

Email addresses: `alpuente@dsic.upv.es` (María Alpuente), `gallardo@lcc.uma.es` (María del Mar Gallardo), `ernesto@lcc.uma.es` (Ernesto Pimentel), `villanue@dsic.upv.es` (Alicia Villanueva).

[1] Work partially supported by MCyT under grant TIC2001-2705-C03.

main idea behind these methods is to take advantage of the constraint dimension of tccp in order to obtain a compact representation of the system, which is then used as an input for the model-checking algorithms. Unfortunately, both [14] and [15] develop exhaustive model-checking algorithms. This causes the traditional state explosion problem and makes them inapplicable to large size systems. In this work, we develop a suitable approximation methodology that is based on abstract interpretation [11] in order to drastically reduce the state space of model checking tccp, thus providing a framework where exhaustive analysis of more complex systems can be achieved.

Abstract model checking [10,13,27] combines abstract interpretation [11] and model checking [7] to improve the automatic verification of large systems. Applying abstract model checking involves the abstraction of both the model to be analyzed (M) and the properties to be checked within the model. In the classic abstract model-checking literature, the abstract model M^+ is an over-approximation of the concrete model M, meaning that each possible concrete execution trace is mimicked in the abstract model. This approach allows the verification of properties which regard all the possible behavior paths. Two techniques have been successfully developed to construct M^+. The *predicate abstraction* approach consists of substituting some selected model expressions with boolean variables, which leads to important simplifications (e.g., this is used in the tool SLAM [2,1]). In contrast, the *data abstraction* method reduces the type of certain data by transforming its original concrete domain into an approximate and simpler domain. This second approach has been applied for abstracting models in the Bandera [23] and αSPIN [16] tools.

In this paper, we follow the *data abstraction* method to approximate tccp computations. The common way of formalizing this technique is to introduce *abstract operations* that over-approximate the original ones (see, for instance, [17] where a data-based abstraction for the modeling language Promela is developed). However, due to the double, logical as well as temporal dimension of tccp, inaccurate abstract models would be obtained in our context by simple over-approximation. In order to achieve fine accuracy, we combine over- and under-approximation in the abstraction of tccp operators. This approach is novel and allows us to build abstract models which are satisfactorily precise. The inspiration to combine over- and under-approximation in our context comes from [16].

Applying abstract interpretation in presence of quantifiable information such as time, raises other specific problems which are related to the process synchronization. In tccp, processes are totally synchronized meaning that, at each time instant, all enabled agents (i.e., actions) are simultaneously carried out. Unfortunately, the loss of information caused by the abstraction affects the suspension behavior of processes: the suspension of a process in the original model does not generally imply that the process abstractly suspends; hence synchronization in the abstract model might be damaged. To overcome this problem, we slightly modify the abstract semantics to preserve the suspension behavior mentioned above. To the best of our knowledge, this is the first total correctness result for abstract model checking of tccp programs in the literature.

In the context of model checking, a well-known and practical approach for implementing abstraction is the automatic source-to-source transformation of the original specification into its abstract version. Thus, any existing model checker for the original modelling language may be used as an abstract model checker and, in addition, the abstraction approach and the rest of optimization techniques implemented in the tool may be combined to improve the analysis. Following these ideas, we develop a source-to-source transformation methodology for implementing abstraction of tccp programs.

The paper is organized as follows. Section 2 recalls the main features of the tccp language. In Section 3, we introduce our data abstraction methodology for tccp, which is based on two entailment relations $\vdash^+$ and $\vdash^-$. The combination of the two abstract relations allows us to contain the potential addition of non-determinism caused by the abstraction, thus achieving very accurate approximations. However, this preliminary abstract semantics does not take into account the suspension behavior of processes. Section 4 discusses the correctness of this semantics and proves that the abstract semantics is correct w.r.t. the original one, provided the suspension behavior is correctly simulated. Then, we formalize a refined abstract semantics that correctly models process suspension. Section 5 develops an implementation of the abstract semantics which is defined as a source-to-source transformation that compiles the abstract program back into tccp code. This transformation is non-trivial as it requires introducing delays for the synchronization of agents inside instantaneous, non-deterministic choices. Section 5.4 discusses the incompleteness (lack of optimality of this semantics) showing that the approximated model contains abstract traces which not correspond to any concrete computation. To improve the accuracy of the abstract model, two abstraction refinements are proposed which we sketch and illustrate by means of an example. In Section 6, we provide an abstract methodology for approximating the satisfiability of the temporal logic properties being checked. Usually, in the classic papers about abstract model checking [10,13,27], properties are under-approximated which, in some way, compensates the over-approximation of the model and is correct for analyzing universal properties (those that refer to all execution paths). In our methodology, we need to combine over- and under-approximation again in order to achieve accurate approximations. Finally, Section 8 concludes and points out several directions for further research. Proofs of all technical results of the paper are given in Appendix B.

2 The tccp language

In [3], the *Timed Concurrent Constraint* language (tccp in short) was defined as an extension of the Concurrent Constraint programming language ccp [29]. In the cc paradigm, the notion of *store as valuation* is replaced by the notion of *store as constraint*. The computational model is based on a global store where constraints are accumulated and on a set of agents that interact with the store. The model is parametric w.r.t. a particular class of constraint system $\mathcal{C}$ [30,3]. The basis of ccp languages is the ask-tell paradigm [28], which can be understood as an extension of Constraint Logic Programming [25]: in addition to satisfiability (tell), entailment (ask) is introduced. Synchronization is achieved through blocking ask: the process is suspended when the store doesn't entail the ask constraint and it remains suspended until the store entails it. In tccp, a new (w.r.t. ccp) conditional agent $\mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B$ is introduced which makes it possible to model situations where the absence of information can cause the execution of a specific action. Intuitively, the execution of a tccp program evolves by asking and telling information to the store.

Let us briefly recall the tccp syntax for agents:

$$A ::= \mathsf{stop} \mid \mathsf{tell}(c) \mid \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A \mid A||A \mid \exists x\, A \mid \mathsf{p}(x)$$

where c, c_i are *finite constraints* (i.e., atomic propositions) of $\mathcal{C}$. A tccp *process* P is an object of the form $D.A$, where D is a set of procedure declarations of the form $\mathsf{p}(x)$:-B, and B is an agent.[2]

Intuitively, the stop agent finishes the execution of the program, $\mathsf{tell}(c)$ adds the constraint c to the store, whereas the choice agent $(\sum_{i=0}^{n}\mathsf{ask}(c_i) \rightarrow A_i)$ consults the store and non-deterministically executes the agent A_i in the following time instant, provided the store satisfies the condition c_i; otherwise the agent suspends. The conditional agent $\mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B$ can process *negative information* in the sense that, if the store satisfies c, then the agent A is executed; otherwise (even if $\neg c$ does not hold), B is executed. $A||B$ executes the two agents A and B in parallel. The $\exists x\, A$ agent is used to hide the information regarding x, i.e., it makes x local to the agent A.

The notion of time is introduced by defining a global clock that synchronizes all agents. In the semantics, the only agents that consume time are the *tell, choice* and *procedure call* agents. In order to simulate the values of the system variables throughout time, we use streams that are encoded by means of lists. The head of the list represents, at each time instant, the current value of the variable.

```
user(C,A):- ask(A=[free|_]) → tell(C=[on|_]) +
    ask(A=[free|_]) → tell(C=[off|_]) +
    ask(A=[free|_]) → tell(C=[c|_]) +
    ask(A=[free|_]) → tell(true).
photocopier(C,A,MIdle,E,T):- ∃ Aux,Aux',T' (tell(T=[Aux|T']) ||
   ask(true) → now (Aux>0) then
                   now (C=[on|_]) then
                       tell(E=[going|_] ∧ T'=[MIdle|_] ∧ A=[free|_])
                   else now (C=[off|_]) then
                           tell(E=[stop|_] ∧ T'=[MIdle|_] ∧ A=[free|_])
                       else now (C=[c|_]) then
                               tell(E=[going|_] ∧ T'=[MIdle|_] ∧ A=[free|_])
                           else tell(Aux'=Aux-1) || tell(T'=[Aux'|_] ∧ A=[free|_])
               else tell(E=[stop|_]) || tell(A=[free|_])).
system(MIdle,E,C,A,T):- ∃ E',C',A',T'(tell(E=[_|E']) || tell(C=[_|C']) ||
   tell(A=[_|A']) || tell(T=[_|T']) || user(C,A) ||
   ask(true)→photocopier(C,A',MIdle,T,E') ||
   ask(A'=[free|_])→(system(MIdle,E',C',A',T'))|| tell(s(E',C',A',T'))).
initialize(MIdle):- ∃ E,C,A,T(tell(A=[free|_]) || tell(T=[MIdle|_]) ||
                        tell(E=[off|_]) || system(MIdle,E,C,A,T) ||
                        tell(s(E,C,A,T))).
```

Figure 1. A tccp program modeling a photocopier

[2] We assume that all programs considered in this work are well-typed.

We show an example of a `tccp` program in Figure 1. This program models a photocopier by means of four procedure declarations which represent the two main processes (`user(C,A)` and `photocopier(C,A,MIdle,E,T)`) and the synchronization of such processes (`system(MIdle,E,C,A,T)` and `initialize(MIdle)`).

Agent `user(C,A)` can execute four different actions: turn on the photocopier (`on`), turn it off (`off`), do a copy request (`c`), or do nothing. The system is assumed to be synchronous, in the sense that the user cannot execute (through stream `C`) any action before the photocopier satisfies the previous request. This behavior is modeled by instantiating the (head of the) system variable `A` to `free`. The stream variable `T` is used as a counter to verify that no request has been received after `MIdle` time units. When this occurs, the photocopier is automatically turned-off.

In order to start the execution, the system is initialized by running the process `initialize(MIdle)`, which fixes the value of variable `MIdle`, and then the photocopier and the user processes are executed in parallel by means of the synchronization process `system(MIdle,E,C,A,T)`. The convenience of storing constraint `s(E,C,A,T)` will be clear in Section 6 when we approximate the properties to be checked in the abstract program.

3 Abstract tccp programs

Recently, some model-checking algorithms have been developed for the concurrent constraint paradigm [14,15]. The common idea behind them is to exploit the constraint nature of the language to represent a model of the system in a compact way. However, the state explosion problem of classical model-checking techniques also occurs in these algorithms. In this section, we develop an abstract model-checking technique as a solution to this problem.

3.1 Abstracting constraint systems

Definition 1 *A* simple constraint system *is a structure* $\langle \mathcal{C}, \vdash \rangle$ *where* $\mathcal{C}$ *is the set of atomic constraints and relation* $\vdash \subseteq \wp(\mathcal{C}) \times \mathcal{C}$ *satisfies*

C1. $u \vdash C$*, for all* $C \in u$. *C2.* $u \vdash C$*, if* $u \vdash C', \forall C' \in v$, and $v \vdash C$

Relation $\vdash$ can be extended to a relation $\vdash \subseteq \wp(\mathcal{C}) \times \wp(\mathcal{C})$ as follows:

$$u \vdash v \iff \forall C \in v, u \vdash C$$

During `tccp` computations, stores are represented by elements of $\wp(\mathcal{C})$. In other words, if $u \subseteq \mathcal{C}$ is the current store, the information accumulated in u is the *conjunction* of all constraints $C \in u$. In addition, $\vdash$ is the entailment relation used to deduce information from stores. We will denote by Θ the set $\wp(\mathcal{C})$.

Proposition 2 *Relation* $\vdash$ *has the following properties:*

(1) (Reflexivity) $\forall u \in \Theta . u \vdash u$.
(2) (Transitivity) $\forall u, v, w \in \Theta . u \vdash v, v \vdash w$ *implies that* $u \vdash w$.

An *abstract interpretation* (an *abstraction*) of the simple constraint system $\langle \mathcal{C}, \vdash \rangle$ is given by an *upper closure operator* (uco) $\rho : \wp(\Theta) \rightarrow \wp(\Theta)$, that is, a *monotonic* ($sst_1 \subseteq sst_2$ then $\rho(sst_1) \subseteq \rho(sst_2)$), *idempotent* ($\rho(sst) = \rho(\rho(sst))$) and *extensive* ($sst \subseteq \rho(sst)$) operator. The intuition of this definition is that each store $st \in \Theta$ is abstracted by its closure $\rho(\{st\})$. Closure operators have many interesting properties. For instance, when the considered domain is a complete lattice, e.g. $\langle \wp(\Theta), \subseteq \rangle$, each closure operator is uniquely determined by the set of its fixed points. In the context of abstract interpretation, closure operators are important because abstract domains can be equivalently defined by using them or by Galois insertions, as introduced in [12]. Let $\iota : \rho(\wp(\Theta)) \rightarrow E$ be an isomorphism. Then, given an uco $\rho : \wp(\Theta) \rightarrow \wp(\Theta)$, structure $(\wp(\Theta), \iota\circ\rho, \iota^{-1}, E)$ is a Galois insertion, where $\iota \circ \rho$ and ι^{-1} are the abstraction and concretization functions, respectively.

Using abstract interpretation terminology, $\rho(\{st\})$ is the most precise abstraction of the store $st \in \Theta$ and, if $\rho(\{st\}) \subseteq sst$, then sst is also an abstraction of st.

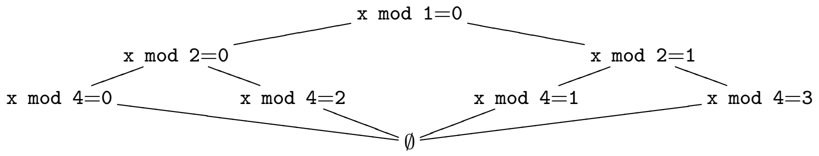

Figure 2. Lattice of abstract stores of Example 3

Example 3 *Given two variables x and y, let $\mathcal{C} = \{x = n | n \in \mathbb{N}\} \cup \{y = n | n \in \mathbb{N}\}$, and let $\rho_x : \wp(\Theta) \rightarrow \wp(\Theta)$ be a constraint abstraction which does not affect variable y, while the abstract value of $x = n$ is defined as follows. Let expression* `x mod a = b` *represent the set of stores which contain the constraint $x = n$, with n mod $a = b$. Then, the abstraction for $x = n$ is given in Figure 2. To formalize ρ_x, we consider the following sets of abstract stores, with $m \in \mathbb{N}$:*

- $(x \text{ mod } a = b, y = m) \overset{def}{=} \{\{x = b + ak, y = m\} | k \in \mathbb{N}\}$
- $(x \text{ mod } a = b) \overset{def}{=} \{\{x = b + ak\} | k \in \mathbb{N}\}$
- $(y = m) \overset{def}{=} \{\{y = m\}\}$

Using the lub operator of the lattice shown in Figure 2 (denoted below as $\bigsqcup$), we define operator $\bigsqcup_x$ over these sets as follows:

- $(x \, mod \, a{=}b, y{=}m) \bigsqcup_x (x \, mod \, c{=}d, y{=}m) \overset{def}{=} (x \, mod \, a{=}b \bigsqcup x \, mod \, c{=}d, y{=}m)$
- $(x \, mod \, a_1 = b_1) \bigsqcup_x (x \, mod \, a_2 = b_2) \overset{def}{=} (x \, mod \, a_1 = b_1) \bigsqcup (x \, mod \, a_2 = b_2)$
- $e_1 \bigsqcup_x e_2 \overset{def}{=} e_1 \cup e_2$, *otherwise.*

Now, ρ_x is defined as $\rho_x(\emptyset) = \emptyset$; $\rho_x(\{st\}) = e$ iff e is the smallest set of abstract stores such that $st \in e$; and $\rho_x(\{st_i | i \in I\}) = \bigsqcup_x \{\rho_x(\{st_i\}) | i \in I\}$.

The following definition introduces two dual entailment relations for abstract constraint systems. Roughly speaking, an abstract store is a set of concrete stores; in other words, each element of an abstract store is a concrete store.

Definition 4 *Let $\langle \mathcal{C}, \vdash \rangle$ be a simple constraint system and $\rho : \wp(\Theta) \rightarrow \wp(\Theta)$ be a constraint abstraction. Then, we define the* over- *and* under-approximated constraint systems *$\langle \Theta, \vdash^+_\rho \rangle$ and $\langle \Theta, \vdash^-_\rho \rangle$ where $\vdash^+_\rho, \vdash^-_\rho \subseteq \wp(\Theta) \times \wp(\Theta)$, by:*

(1) $sst_1 \vdash^+_\rho sst_2 \iff \exists u \in \rho(sst_1), \exists v \in sst_2$ *such that* $u \vdash v$.
(2) $sst_1 \vdash^-_\rho sst_2 \iff \forall u \in \rho(sst_1), \exists v \in sst_2$ *such that* $u \vdash v$.

The following proposition justifies the names of the new structures given in the previous definition.

Proposition 5 *Let $\langle \mathcal{C}, \vdash \rangle$ be a simple constraint system and $\rho : \wp(\Theta) \rightarrow \wp(\Theta)$ be a constraint abstraction. Then,*

(1) If $u \vdash v$, then $\{u\} \vdash^{+}_{\rho} \{v\}$. (2) If $\{u\} \vdash^{-}_{\rho} \{v\}$, then $u \vdash v$.

Example 6 *Consider the* `tccp` *program shown in Figure 1, and let $\mathcal{C}$ be the considered set of atomic constraints (defined in the obvious way). Define the set* $\texttt{msg}{=}\{\texttt{on},\texttt{off},\texttt{c}\}$. *Given $X, X' \in Var$, construct the sets $msg(X, X') = \{X = [A|X'] | A \in \texttt{msg}\}$ and $MSG = \cup_{X,X' \in Var} msg(X, X')$. We write $c \simeq c'$ iff $\exists X, X' \in Var$ such that $c, c' \in msg(X, X')$. Let $|u|$ denote the number of simple constraints in the store u. Then, we write $u_2 \simeq u'_2$ iff $|u_2| = |u'_2|$ and $\forall c \in u_2 . \exists c' \in u'_2$ such that $c \simeq c'$.*
A constraint abstraction $\rho : \wp(\Theta) \rightarrow \wp(\Theta)$ which abstracts the messages in MSG can be defined as follows. Divide each store $u \in \Theta$ into the subsets: $u_1 = u - MSG$, and $u_2 = u \cap MSG$, then

- $\rho(\{u_1 \cup u_2\}) = \{u_1 \cup u'_2 | u_2 \simeq u'_2\}$.
- $\rho(sst) = (\cup_{u \in sst}\, \rho(\{u\}))$.

For instance,

$$\rho(\{\{X = [\texttt{on}|X']\}\}) = \{\{X = [\texttt{off}|X']\}, \{X = [\texttt{on}|X']\}, \{X = [\texttt{c}|X']\}\}$$

Note that an implementation of this abstraction would substitute the three concrete constants `on`, `off` *and* `c` *by a new, abstract constant (for example,* `msg`*), thus making the abstract store simpler.*

Proposition 7 *Let $\langle \mathcal{C}, \vdash \rangle$ be a simple constraint system and $\rho : \wp(\Theta) \rightarrow \wp(\Theta)$ be a constraint abstraction. Then,*

(1) (Reflexivity for $\vdash^{+}_{\rho}$) $\forall sst \in \wp(\Theta).\ sst \vdash^{+}_{\rho} sst$.
(2) (Transitivity for $\vdash^{-}_{\rho}$) $\forall sst_1, sst_2, sst_3 \in \wp(\Theta).\ sst_1 \vdash^{-}_{\rho} sst_2$ and $sst_2 \vdash^{-}_{\rho} sst_3$ implies that $sst_1 \vdash^{-}_{\rho} sst_3$.

Intuitively, the set of formulae which follow from an abstract store by means of $\vdash^{+}_{\rho}$ is bigger than the one inferred by applying $\vdash^{-}_{\rho}$. It is worth noting that, in general, relation $\vdash^{+}_{\rho}$ is not transitive and $\vdash^{-}_{\rho}$ is not reflexive, as shown in the following example.

Example 8 *Consider again Example 3 extending the constraint system with the constraint $even(x)$, and redefining ρ_x conveniently. Then,*

(1) $\vdash^{+}_{\rho_x}$ is not transitive. $\{\{x = 8\}\} \vdash^{+}_{\rho_x} \{\{even(x)\}\}$ and $\{\{even(x)\}\} \vdash^{+}_{\rho_x} \{\{x = 6\}\}$, since $\{x = 6\} \in \rho_x(\{\{even(x)\}\})$ and $\{x = 6\} \vdash \{x = 6\}$. However, $\{\{x = 8\}\} \not\vdash^{+}_{\rho_x} \{\{x = 6\}\}$.
(2) $\vdash^{-}_{\rho_x}$ is not reflexive. $\{\{x = 2\}\} \not\vdash^{-}_{\rho_x} \{\{x = 2\}\}$, since $\{x = 6\} \in \rho_x(\{\{x = 2\}\})$ and $\{x = 6\} \not\vdash \{x = 2\}$.

The following definition introduces the abstract union operator $\sqcup^{\rho}$ for abstract constraint sets. Note that we remove the inconsistent stores (that may appear during an abstract computation) by a satisfiability test $u \cup v \not\vdash \mathit{false}$, where *false* is the empty constraint.

In order to simplify the notation, we define the operator $\otimes : \wp(\Theta) \times \wp(\Theta) \to \wp(\Theta)$ as $sst_1 \otimes sst_2 = \{u \cup v | u \in sst_1, v \in sst_2, u \cup v \not\vdash \mathit{false}\}$. In addition, given a store st we write $sst \otimes st$ for $sst \otimes \{st\}$.

Definition 9 *We define the operator* $\sqcup^\rho : \wp(\Theta) \to \wp(\Theta)$ *as* $sst_1 \sqcup^\rho sst_2 = \rho(sst_1 \otimes sst_2)$

The following proposition states that operator $\sqcup^\rho$ correctly approximates $\cup$.

Proposition 10 *For all* $u, v \in \Theta$, *and* $sst_1, sst_2 \in \wp(\Theta)$, *if* $\rho(\{u\}) \subseteq sst_1$ *and* $\rho(\{v\}) \subseteq sst_2$ *then* $\rho(\{u \cup v\}) \subseteq sst_1 \sqcup^\rho sst_2$.

In tccp, *cylindric constraint systems* are used, which are defined as follows.

Definition 11 $\langle \mathcal{C}, \vdash, \mathit{Var}, \exists \rangle$ *is a* cylindric constraint system *iff* $\langle \mathcal{C}, \vdash \rangle$ *is a simple constraint system, Var is a denumerable set of variables, and for each* $x \in \mathit{Var}$, *there exists a function* $\exists_x : \Theta \to \Theta$ *such that, for each* $u, v \in \wp(C)$:

(1) $u \vdash \exists_x u$
(2) $u \vdash v$ *then* $\exists_x u \vdash \exists_x v$
(3) $\exists_x(u \cup \exists_x v) = \exists_x u \cup \exists_x v$
(4) $\exists_x(\exists_y u) = \exists_y(\exists_x u)$

A *set of diagonal elements* for a cylindric constraint system is a family $\{\delta_{xy} \in \mathcal{C} | x, y \in var\}$ such that

(1) $\emptyset \vdash \delta_{xx}$
(2) If $y \neq x, z$ then $\delta_{xz} = \exists_x(\delta_{xy} \cup \delta_{yz})$.
(3) If $x \neq y$ then $\delta_{xy} \cup \exists_x(v \cup \delta_{xy}) \vdash v$.

Diagonal elements allow us to hide variables, representing local variables, as well as to implement parameter passing among predicates. Thus, quantifier $\exists_x$ and diagonal elements δ_{xy} allow us to properly deal with variables in constraint systems. Assuming that the original constraint system $\langle \mathcal{C}, \vdash \rangle$ to be abstracted is cylindric, and given a constraint abstraction $\rho : \wp(\Theta) \to \wp(\Theta)$, the over and under-approximated constraint systems $\langle \Theta, \vdash^+_\rho \rangle$ and $\langle \Theta, \vdash^-_\rho \rangle$ are not cylindric in general. Example 8 shows that some property of the underlying simple constraint system may be lost during the abstraction process. Moreover, the remaining properties concerning the existential quantifier or the diagonal elements may also be lost. An extensive study of the conditions that the abstraction ρ has to satisfy for the properties of cylindric systems be preserved can be found in [19], where a generalized semantics for concurrent logic languages is introduced. In short, some consistency properties are imposed to ρ to ensure that the existential quantification has the expected semantics after abstraction. We extend function $\exists_x$ to sets of stores by $\exists_x : \wp(\Theta) \to \wp(\Theta)$ where $\exists_x sst = \{\exists_x u | u \in sst\}$.

3.2 Abstract Semantics

As it is shown in [32], the ask-tell paradigm introduces some problems when we deal with abstraction. There, the abstract synchronization problem is addressed by means of two suitable program transformation that ignore or condense synchronization, respectively. When dealing with tccp, these kinds of transformations are even more difficult to apply due to the temporal dimension and the maximal parallelism of tccp, as opposed to the interleaving semantics of ccp.

In the following, we formalize a preliminary abstract operational semantics of tccp programs in terms of a transition relation that is similar to the operational semantics of the original tccp language. We will refer to this new transition system as *abstract operational semantics* or tccp$^{\alpha}$-calculus. Consistent with the original semantics, each transition involves the passage of time. In general, the abstracted agents are over-approximations of their concrete versions. However, the abstraction of the conditional agent has to be done with special care. The reason for this is that the non-determinism introduced when abstracting this agent cannot be handled in tccp instantaneously, since the execution of ask involves the consumption of one time unit. To solve this problem, we have defined a new agent ask! which allows us to introduce non-determinism without consuming time. This aspect distinguishes tccp from other unsophisticated modeling languages which don't have either non-determinism or time aspects.

In the following, we assume that an abstraction operator $\rho : \wp(\Theta) \to \wp(\Theta)$ has been provided and it has the consistency properties discussed in Section 3.1. We let $\vdash_{\rho}^{-}$ ($\vdash_{\rho}^{+}$) represent a suitable under- (over-) approximation of the entailment relation $\vdash$ of the constraint system. By abuse of notation, we drop the subindex ρ from $\vdash_{\rho}^{+}$, $\vdash_{\rho}^{-}$ and $\sqcup^{\rho}$ in order to simplify the presentation. For the same reason, in the sequel, we write $sst \vdash^{+} c$, $sst \vdash^{-} c$ and $sst \sqcup c$ for $sst \vdash^{+} \{\{c\}\}$, $sst \vdash^{-} \{\{c\}\}$ and $sst \sqcup \{\{c\}\}$, respectively.

We show the abstract transition rules for each agent in Figure 3.[3] A *configuration* of the form $\langle \Gamma, sst \rangle$ represents a computation state, where Γ is an agent and $sst \in \wp(\Theta)$ is an abstract store. We are assuming that the tccp system is closed under the usual structural equivalence relation where the parallelism operator is commutative and agents $A||$stop and A are equivalent.

Let us explain the main differences w.r.t. the concrete tccp semantics defined in [3]. The main points of the abstract semantics are the new ask! agent and the use of the two abstract entailment relations $\vdash^{+}$ and $\vdash^{-}$. For the conditional agent we use under-approximation, whereas over-approximation is more convenient for choice primitives. The abstract version of agent A is denoted by A^{α}, except for the parallel and hide operators because their abstract and concrete semantics coincide. There are two completely new rules (**R3a** and **R3b**), which define the semantics for the instantaneous choice agent (ask!). These rules state that, provided agent A_j can evolve to agent A'_j, the instantaneous choice can evolve to A'_j. It is important to remark the timing difference between rule **R2** and rule **R3a**. Both of them introduce non-determinism but a time unit is consumed in the first one before executing the agent in the body of the ask$^{\alpha}$ agent, whereas in the second rule, non-determinism is introduced instantaneously.

[3] In rule **R12**, the superscript in $\exists^{d}B$ represents the information d accumulated during the execution of the agent B. See [3] for details.

R1 $\langle \mathsf{tell}^\alpha(c), sst\rangle \longrightarrow_\alpha \langle \mathsf{stop}^\alpha, sst \sqcup c\rangle$

R2 $\langle \sum_{i=0}^{n} \mathsf{ask}^\alpha(c_i) \rightarrow A_i, sst\rangle \longrightarrow_\alpha \langle A_j, sst\rangle$ if $sst \vdash^+ c_j,\ \ 0 \leq j \leq n$

R3a $\dfrac{\langle A_j, sst\rangle \longrightarrow_\alpha \langle A'_j, sst'\rangle}{\langle \sum_{i=0}^{n} \mathsf{ask!}(c_i) \rightarrow A_i, sst\rangle \longrightarrow_\alpha \langle A'_j, sst'\rangle}$ if $sst \vdash^+ c_j,\ \ 0 \leq j \leq n$

R3b $\dfrac{\langle A_j, sst\rangle \not\longrightarrow_\alpha}{\langle \sum_{i=0}^{n} \mathsf{ask!}(c_i) \rightarrow A_i, sst\rangle \longrightarrow_\alpha \langle A_j, sst\rangle}$ if $sst \vdash^+ c_j,\ \ 0 \leq j \leq n$

R4 $\dfrac{\langle A, sst\rangle \longrightarrow_\alpha \langle A', sst'\rangle}{\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, sst\rangle \longrightarrow_\alpha \langle A', sst'\rangle}$ if $sst \vdash^- c$

R5 $\dfrac{\langle A, sst\rangle \not\longrightarrow_\alpha}{\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, sst\rangle \longrightarrow_\alpha \langle A, sst\rangle}$ if $sst \vdash^- c$

R6 $\dfrac{\langle B, sst\rangle \longrightarrow_\alpha \langle B', sst'\rangle}{\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, sst\rangle \longrightarrow_\alpha \langle B', sst'\rangle}$ if $sst \not\vdash^- c$

R7 $\dfrac{\langle B, sst\rangle \not\longrightarrow_\alpha}{\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, sst\rangle \longrightarrow_\alpha \langle B, sst\rangle}$ if $sst \not\vdash^- c$

R8 $\dfrac{\langle A, sst\rangle \longrightarrow_\alpha \langle A', sst'_1\rangle,\ \langle B, sst\rangle \longrightarrow_\alpha \langle B', sst'_2\rangle}{\langle A||B, sst\rangle \longrightarrow_\alpha \langle A'||B', sst'_1 \sqcup sst'_2\rangle}$

R9 $\dfrac{\langle A, sst\rangle \longrightarrow_\alpha \langle A', sst'\rangle,\ \langle B, sst\rangle \not\longrightarrow_\alpha}{\langle A||B, sst\rangle \longrightarrow_\alpha \langle A'||B, sst'\rangle}$

R10 $\dfrac{\langle A, sst_1 \sqcup \exists x sst_2\rangle \longrightarrow_\alpha \langle A', sst'\rangle}{\langle \exists^{sst_1} x A, sst_2\rangle \longrightarrow_\alpha \langle \exists^{sst'} x A', sst_2 \sqcup \exists x sst'\rangle}$

R11 $\langle p(x), sst\rangle \longrightarrow_\alpha \langle A, sst\rangle$ if $p(x)$:-$A \in D$

Figure 3. Abstract operational semantics

3.3 Program Abstraction

In this section, we give a first step towards a source-to-source transformation of tccp programs into abstract programs which represent an approximate model of the system. For each tccp agent A, we inductively construct a corresponding abstract tccp$^\alpha$ agent $\alpha(A)$ as is shown in Figure 4. An example of program abstraction is shown in Figure 5. Note that the transformed program which results from the abstraction process contains abstract agents, which are not pure tccp primitives.

The intuitive idea of the transformation of the conditional agent $\mathsf{now}\, c\, \mathsf{then}\, A$ $\mathsf{else}\, B$ is as follows. In order to mimic the possible conditional execution in the concrete model by an execution in the corresponding abstract model, we consider the following four possible cases, where $st \in \Theta$ and $sst \in \wp(\Theta)$ are, respectively, the concrete store and the abstract one, and $\rho(\{st\}) \subseteq sst$.

- If $st \vdash c$ and $sst \vdash^- c$, then A is executed in both the concrete and the abstract models.
- If $st \vdash c$ and $sst \not\vdash^- c$, then agent A is executed in the concrete model whereas any of the agents A or B could be executed in the abstract one.

Stop agent. $\alpha(\mathsf{stop}) = \mathsf{stop}^{\alpha}$. **Tell agent.** $\alpha(\mathsf{tell(c)})=\mathsf{tell}^{\alpha}(c)$.

Choice agent. $\alpha(\sum_{i=0}^{n} \mathsf{ask}(c_i) \to A_i) = \sum_{i=0}^{n} \mathsf{ask}^{\alpha}(c_i) \to \alpha(A_i)$.

Conditional agent. $\alpha(\mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B)=$

$\mathsf{now}^{\alpha}\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else}\ \mathsf{ask!}(c) \to \alpha(A) + \mathsf{ask!}(true) \to \alpha(B)$

Parallel agent. $\alpha(A||B) = \alpha(A)||\alpha(B)$.

Hiding agent. $\alpha(\exists x\, A) = \exists x\, \alpha(A)$, where x is a variable.

Procedure Call agent. $\alpha(p(x)) = p(x)$ where x is a variable of the constraint system and there exists a declaration $p(x)$:-A.

Declaration. $\alpha(D) = p(x)$:-$\alpha(A)$ if $D = p(x)$:-A and where x is a variable of the constraint system.
$\alpha(D){=}\alpha(D_1).\alpha(D_2)$ if $D{=}D_1.D_2$ and both,D_1 andD_2, are declarations.

Program. $\alpha(P) = \alpha(D).\alpha(A)$ where $P = D.A$, D is a declaration and A is an agent.

Figure 4. α-transformation for tccp programs

- If $st \not\vdash c$ but $sst \vdash^{+} c$, then agent B is executed in the concrete model, whereas any of the agents A or B could be executed in the abstract one.
- If $st \not\vdash c$ and $sst \not\vdash^{+} c$, then both, the abstract and the concrete models execute agent B.

Note that the availability of the two abstract entailment relations allows us to very accurately approximate the behavior of the conditional agent in the first and fourth cases above, whereas we are not able to achieve this accuracy in the other two cases. By using only $\vdash^{+}$, we would not have been able to achieve this precision in any case.The remaining agents are translated into the corresponding abstract versions in the natural way.

4 Correctness

In abstract model checking, correctness means that, whenever a property is true in the abstract model, it will also be true in the concrete one. In this section, we demonstrate that some additional conditions concerning the suspension behavior of the program are needed for the abstract semantics of tccp programs correctly approximate the standard one. Namely, we require that local suspension be preserved by the constraint approximation function ρ. Then, we show how the abstract semantics can be refined in order to correctly simulate suspension.

4.1 Correctness conditions

Given a tccp program (a process) P of the form $D.\Gamma_0$ and an initial configuration $\langle \Gamma_0, st_0 \rangle$, a *trace* t of P starting at $\langle \Gamma_0, st_0 \rangle$ is a sequence of configurations $t = \langle \Gamma_0, st_0 \rangle \longrightarrow \cdots$ which is built by applying the transition relation rules $\longrightarrow$ defined in [3]. Let $\mathcal{O}(P)(\langle \Gamma_0, st_0 \rangle)$ denote the corresponding standard operational semantics. We say that a concrete trace $t{=}\langle \Gamma_0, st_0 \rangle \longrightarrow \cdots \in \mathcal{O}(P)(\langle \Gamma_0, st_0 \rangle)$ is *erroneous* iff $\exists i \geq 0.st_i$ is not consistent.

```
user(C,A):- askα(A=[free|_]) → tellα(C=[on|_]) +
    askα(A=[free|_]) → tellα(C=[off|_]) +
    askα(A=[free|_]) → tellα(C=[c|_]) +
    askα(A=[free|_]) → tellα(true).
photocopier(C,A,MIdle,E,T):- ∃ Aux, Aux',T'(
   tellα(T=[Aux|T']) ||
   askα(true) → nowα(Aux>0) then
                    nowα (C=[on|_]) then tellα(E=[going|_] ∧ T=[MIdle|_] ∧ A=[free|_]) else
                         ask!(C=[on|_]) → tellα(E=[going|_] ∧ T=[MIdle|_] ∧ A=[free|_]) +
                         ask!(true) → now (C=[off|_]) then tellα(E=[stop|_] ∧ T=[MIdle|_] ∧ A=[free|_]) else
                                          ask!(C=[off|_]) → tellα(E=[stop|_] ∧ T=[MIdle|_] ∧ A=[free|_]) +
                                          ask!(true) → now (C=[c|_]) then tellα(E=[going|_] ∧ T=[MIdle|_] ∧ A=[free|_]) else
                                                           ask!(C=[c|_]) → tellα(E=[going|_] ∧ T=[MIdle|_] ∧ A=[free|_]) +
                                                           ask!(true) → tellα(Aux'=Aux-1) || tellα(T=[Aux'|_] ∧ A=[free|_])
               else ask!(Aux>0) → nowα(C=[on|_])then tellα(E=[going|_] ∧ T=[MIdle|_] ∧ A=[free|_]) else
                                       ask!(C=[on|_]) → tellα(E=[going|_] ∧ T=[MIdle|_] ∧ A=[free|_]) +
                                       ask!(true) → nowα(C=[off|_]) then tellα(E=[stop|_] ∧ T=[MIdle|_] ∧ A=[free|_]) else
                                                        ask!(C=[off|_]) → tellα(E=[stop|_] ∧ T=[MIdle|_] ∧ A=[free|_]) +
                                                        ask!(true) → nowα (C=[c|_]) then tellα(E=[going|_] ∧ T=[MIdle|_]
                                                                         ∧ A=[free|_])
                                                                    else ask!(C=[c|_]) → tellα(E=[going|_] ∧ T=[MIdle|_]
                                                                                             ∧ A=[free|_]) +
                                                                         ask!(true) → tellα(Aux=T1-1) || tellα(T=[Aux|_]
                                                                                                                ∧ A=[free|_]) +
                    ask!(true) → tellα(E=[stop|_]) || tellα(A=[free|_])).
system(MIdle,E,C,A,T):- ∃ E',C',A',T'(tellα(E=[_|E']) || tellα(C=[_|C']) ||
   tellα(A=[_|A']) || tellα(T=[_|T']) || user(C,A) ||
   askα(true)→photocopier(C,A',MIdle,T',E') ||
   askα(A'=[free|_])→system(MIdle,E',C',A',T') ||
   tellα(s(E',C',A',T'))).
initialize(MIdle):- ∃ E,C,A,T(tellα(A=[free|_]) || tellα(T=[MIdle|_]) ||
                              tellα(E=[off|_]) || system(MIdle,E,C,A,T)
                              tellα(s(E,C,A,T))).
```

Figure 5. Photocopier program after α-transformation

Similarly, given an abstraction ρ, let $\mathcal{A}_\rho(P^\alpha)(\langle\Gamma_0, sst_0\rangle)$ denote the set of abstract traces generated by the abstract program P^α by using the abstract operational semantics given by Figure 3. Note that abstract program P^α may include the new agent ask!

Given a trace $t = \langle\Gamma_0, st_0\rangle \longrightarrow \langle\Gamma_1, st_1\rangle \longrightarrow \cdots \in \mathcal{O}(P)(\langle\Gamma_0, st_0\rangle)$, we denote with $\alpha(t)$ the abstract trace obtained by point-wise applying the transformation α presented previously (Figure 4) to the agents in the configurations of t, and abstracting the corresponding stores using ρ; that is, $\alpha(t) = \langle\alpha(\Gamma_0), \rho(\{st_0\})\rangle \longrightarrow_\alpha$

$\langle\alpha(\Gamma_1), \rho(\{st_1\})\rangle \longrightarrow_\alpha \cdots$. Given two abstract traces of the form $t_1^\alpha{=}\langle\Gamma_0^\alpha, sst_{01}\rangle \longrightarrow_\alpha \langle\Gamma_1^\alpha, sst_{11}\rangle \longrightarrow_\alpha \cdots$ and $t_2^\alpha = \langle\Gamma_0^\alpha, sst_{02}\rangle \longrightarrow_\alpha \langle\Gamma_1^\alpha, sst_{12}\rangle \longrightarrow_\alpha \cdots$, we write $t_1^\alpha \sqsubseteq t_2^\alpha$ whenever $sst_{i1} \subseteq sst_{i2}$, for all $i \geq 0$.

Correctness Conditions (CC) The constraint abstraction function ρ satisfies the correctness conditions if it preserves the local suspension of the concrete configurations, that is, for all configuration Γ and each store st, if $\langle\Gamma, st\rangle \not\longrightarrow$ and $\rho(\{st\}) \subseteq sst$ then $\langle\alpha(\Gamma), sst\rangle \not\longrightarrow_\alpha$.

Lemma 12 *Consider a* `tccp` *program P and a constraint abstraction ρ satisfying* **CC**. *Let $\langle\Gamma, st\rangle$ and $\langle\Gamma', st'\rangle$ be two standard configurations such that $\langle\Gamma, st\rangle \longrightarrow \langle\Gamma', st'\rangle$. Then, for all $sst \in \wp(\Theta)$ with $\rho(\{st\}) \subseteq sst$ there exists $sst' \in \wp(\Theta)$ verifying that $\langle\alpha(\Gamma), sst\rangle \longrightarrow_\alpha \langle\alpha(\Gamma'), sst'\rangle$ and $\rho(\{st'\}) \subseteq sst'$.*

Theorem 13 *Consider a* `tccp` *program P, an initial configuration $\langle\Gamma_0, st_0\rangle$ and a constraint abstraction function ρ satisfying* **CC**. *Then, for each non-erroneous trace $t \in \mathcal{O}(P)(\langle\Gamma_0, st_0\rangle)$, there exists an abstract trace of the form $t^\alpha \in \mathcal{A}_\rho(\alpha(P))(\langle\alpha(\Gamma_0), \rho(\{st_0\})\rangle)$ such that $\alpha(t) \sqsubseteq t^\alpha$.*

Example 14 *The abstraction provided in Example 6 for the* `tccp` *program illustrated in Figure 1 satisfies* **CC**: *if stream* `C` *contains a message, then the concrete model never suspends nor does the abstract model. Moreover, if* `C` *has no message, then both the concrete and the abstract model suspend. Therefore, Theorem 13 can be applied to this example. This abstraction is useful for checking liveness properties like "the photocopier is switched off when it is inactive during* `MIdle` *time units" as shown in Example 27.*

Obviously, if **CC** doesn't hold, the abstraction may modify some time aspects, in such a way that abstract agents are not correctly synchronized, as illustrated by the following example.

Example 15 *Consider the abstraction ρ given in Figure 2 which considers the divisibility of variable X by 4. Let us demonstrate that ρ doesn't satisfy* ***CC***. *It suffices to find a concrete suspension computation which doesn't suspend in the abstract model. In the figure, the new agents* `A'` *and* `B'` *represent the possible evolution of processes* `A` *and* `B` *by the eventual application of rules* **R4** *or* **R6**.

Then, the abstract trace shown above doesn't model the real suspension behavior of the program.

Since **CC** is a quite demanding condition not easy to be checked, in the following section, a different approach to solve the above problem is obtained by instrumenting the abstract semantics to avoid the problem of correctly simulate suspension. Roughly speaking, we achieve this by introducing two new rules for correct abstract semantics of `tccp`. We redress the abstract semantics following the general approach of confusing quiescence and nontermination, which is a general theme in `ccp` semantics (e.g., that of determinate `ccp` in [31]). In our context, this is achieved by converting suspensions into infinite loops.

Concrete Trace

STORE	AGENTS
X=0	ask(X=4) $\rightarrow$ tell(Y=2) || ask($true$) $\rightarrow$ ask($true$) $\rightarrow$ now$^\alpha$ Y=2 then A else B
X=0	ask(X=4) $\rightarrow$ tell(Y=2) || ask(true) $\rightarrow$ now Y=2 then A else B
X=0	ask(X=4) $\rightarrow$ tell(Y=2) || now Y=2 then A else B
X=0	ask(X=4) $\rightarrow$ tell(Y=2) || B'

Abstract trace

STORE	AGENTS
x mod 4 = 0	ask$^\alpha$(X=4)$\rightarrow$ tell$^\alpha$(Y=2) || ask$^\alpha$(true)$\rightarrow$ ask$^\alpha$($true$)$\rightarrow$ α(now Y=2 then A else B)
x mod 4 = 0	tell$^\alpha$(Y=2) || ask$^\alpha$(true) $\rightarrow$ now$^\alpha$ Y=2 then $\alpha(A)$ else (ask!(Y=2) $\rightarrow \alpha(A)$ + ask!($true$) $\rightarrow \alpha(B)$)
x mod 4 = 0 $\sqcup$ Y=2	now$^\alpha$ Y=2 then $\alpha(A)$ else (ask!(Y=2) $\rightarrow \alpha(A)$ + ask!($true$) $\rightarrow \alpha(B)$)
x mod 4 = 0 $\sqcup$ Y=2	A'

Figure 6. An incorrect abstract model

4.2 A Correct Abstract Semantics

Namely, in order to simulate suspension in the abstract semantics, when a configuration containing an ask agent suspends in the concrete semantics, the corresponding abstract configuration is replicated in the new abstract semantics. Consider the transition system obtained by modifying the abstract semantics given in Figure 3 with the new rules given in Figure 7 as follows: rule **R0** and rule **R2'** are added, and rules **R3b**, **R5**, **R7** and **R9** are dropped.

R0 $\langle \mathsf{stop}^\alpha, sst\rangle \longrightarrow_\alpha \langle \mathsf{stop}^\alpha, sst\rangle$

R2' $\langle \sum_{i=0}^{n} \mathsf{ask}^\alpha(c_i) \rightarrow A_i, sst\rangle \longrightarrow_\alpha \langle \sum_{i=0}^{n} \mathsf{ask}^\alpha(c_i) \rightarrow A_i, sst\rangle \quad$ if $sst \not\vdash^- \{\{c_0\}, \cdots, \{c_n\}\}$

Figure 7. New rules for a correct abstract semantics of tccp

Roughly speaking, the refined abstract semantics given in Figure 7 solves this problem by identifying inactivity and nontermination. Thus, the usual behavior of the agent choice is slightly modified by non-deterministically allowing its repetition in the next time instant, when the concrete version of the agent may suspend.

The new semantics ($\mathcal{A}'_\rho$) gives us the desired correctness result.

Lemma 16 *Consider a* tccp *program P and a constraint abstraction ρ. Let $\langle \Gamma, st\rangle$ and $\langle \Gamma', st'\rangle$ be two standard configurations and $sst \in \wp(\Theta)$ such that $\rho(\{st\}) \subseteq sst$. Then,*

(1) If $\langle \Gamma, st\rangle \not\longrightarrow$, then there exists $sst' \in \wp(\Theta)$ such that $\langle \alpha(\Gamma), sst\rangle \longrightarrow_\alpha \langle \alpha(\Gamma), sst'\rangle$ and $sst \subseteq sst'$.

(2) If $\langle \Gamma, st\rangle \longrightarrow \langle \Gamma', st'\rangle$, then there exists $sst' \in \wp(\Theta)$ such that $\langle \alpha(\Gamma), sst\rangle \longrightarrow_\alpha \langle \alpha(\Gamma'), sst'\rangle$ and $\rho(\{st'\}) \subseteq sst'$.

Theorem 17 *Consider a* tccp *program P of the form $D.\Gamma_0$, an initial configuration $\langle \Gamma_0, st_0 \rangle$ and a constraint abstraction ρ. For each non-erroneous trace $t \in \mathcal{O}(P)(\langle \Gamma_0, st_0 \rangle)$, there exists an abstract trace $t^\alpha \in \mathcal{A}'_\rho(\alpha(P))(\langle \alpha(\Gamma_0), \rho(\{st_0\}) \rangle)$ such that $\alpha(t) \sqsubseteq t^\alpha$.*

Example 18 *Consider the example in Figure 6 again. The new abstract semantics now produces the correct approximation shown in Figure 8.*

New Abstract trace

STORE	AGENTS
`X mod 4 = 0`	ask$^\alpha$(X=4) $\rightarrow$ tell$^\alpha$(Y=2) || ask$^\alpha$($true$) $\rightarrow$ ask$^\alpha$(true) $\rightarrow$ α(now Y=2 then A else B)
`X mod 4 = 0`	ask$^\alpha$(X=4) $\rightarrow$ tell$^\alpha$(Y=2) || ask$^\alpha$(true)$\rightarrow$ now$^\alpha$ Y=2 then $\alpha(A)$ else (ask!(Y=2)$\rightarrow$ $\alpha(A)$ + ask!(true)$\rightarrow$ $\alpha(B)$)
`X mod 4 = 0`	ask$^\alpha$(X=4) $\rightarrow$ tell$^\alpha$(Y=2) || now$^\alpha$ Y=2 then $\alpha(A)$ else (ask!(Y=2) $\rightarrow$ $\alpha(A)$ + ask!(true) $\rightarrow$ $\alpha(B)$)
`X mod 4 = 0`	ask$^\alpha$(X=4) $\rightarrow$ tell$^\alpha$(Y=2) || B'

Figure 8. A correct abstract model

In the following section, we develop an abstraction-by-transformation technique which we propose as a natural implementation of our methodology.

5 Implementation of the abstract semantics

The source-to-source transformation from the original program into the abstract one (which is then translated back into the source language) is a well-known technique for integrating abstraction and model checking [16,23]. This permits the reuse of the existing model checkers of the original language. In this section, we study the difficulties of applying this method to tccp programs.

In Section 4, we showed how it is possible to correctly abstract tccp programs. Both an abstract semantics for the abstract model and a program transformation delivering the abstract program were formulated. Since we aim to complete a source-to-source transformation delivering an encoding of the abstract program in pure tccp syntax, in this section we develop an implementation of the abstract semantics in terms of the concrete one. In tccp, a pair $(\Theta, \vdash)$ consisting of a set of constraints together with and an entailment relation, determines a timed concurrent constraint system tccp$(\Theta, \vdash)$. Thus, a tccp source-to-source transformation consists of translating a concrete tccp$(\Theta, \vdash)$ program into a difference instance tccp$(\Theta', \vdash')$ of tccp. The abstraction process developed in the previous section defines a transformation α : tccp$(\Theta, \vdash) \rightarrow$ tccp$^\alpha$. This section is devoted to show how programs in tccp$^\alpha$ can be implemented in tccp$(\wp(\Theta), \vdash^+)$ and, eventually, back again in tccp$(\Theta, \vdash)$, as discussed at the end of the section.

5.1 Implementation of the abstract primitives

The implementation of the parallel and hiding operators is straightforward since their semantics in tccp$^\alpha$ coincides with that of tccp$(\wp(\Theta), \vdash^+)$. Similarly, the tell$^\alpha$ primitive is directly implemented by the concrete tell agent.

Following rule **R0**, the implementation of stop^α is given the following agent: $\alpha stop()$:-$\alpha stop()$.

In order to express agent now^α with the entailment relation $\vdash^+$, we define when an abstract store *sst* over-approximates a negative constraint $\neg c$ as follows: $sst \vdash^+ \neg c \Leftrightarrow sst \not\vdash^- c$. Considering this definition the implementation of the conditional abstract agent $\mathsf{now}^\alpha c$ then A else B will be now $\neg c$ then B else A.

The implementation of the semantics of the abstract choice agent, as defined by rules **R2** and **R2'**, is given by the procedure $\alpha choice(c_0; \cdots ; c_n, A_0; \cdots ; A_n)$ where

```
$\alpha choice(c_0; \cdots ; c_n, A_0; \cdots ; A_n)$:-
        $\mathsf{now}^\alpha$ $c_0; \cdots ; c_n$ $\mathsf{then}$ $\Sigma_{i=0}^n \mathsf{ask}$ $(c_i) \rightarrow (A_i)$
        $\mathsf{else}$ $\alpha choice(c_0; \cdots ; c_n, A_0; \cdots ; A_n)||$
            $(\Sigma_{i=0}^n \mathsf{ask}(c_i) \rightarrow (A_i) + \mathsf{ask}(\neg c_0 \wedge \cdots \wedge \neg c_n) \rightarrow \mathsf{stop})$
```

Roughly speaking, we consider the two cases specified by rules **R2** and **R2'**. Namely, if we are sure that no concrete suspension can occur, then the choice agent is executed. Otherwise, the *else* branch models both the possible suspension of the agent, by means of the call to $\alpha choice$, and, simultaneously, the possible evolution of the *choice* agent. Note that the last case of the definition avoids the agent suspension.

The transformation above intends to consider all possibilities of suspensions and no-suspension of choice agents. However, we can optimize the process by identifying a special case: we know that a choice agent containing one branch of the form $\mathsf{ask}(true) \rightarrow$ `A` will never suspend, thus we can simplify the transformation for this kind of agents by simply substituting the abstract version by the concrete one. In Section 5.3, we clarify the usefulness of this optimization.

5.2 Implementation of the instantaneous choice

Due to the introduction of the new ask! agent, which models instantaneous non-determinism, we need to define some elaborate mechanisms to achieve the pursued source-to-source transformation.

Now we need to eliminate the ask! agent. Let us first recall the transformation of the conditional agent proposed in Section 3.3 and explain its main drawback:

$$\alpha(\mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B) = \mathsf{now}^\alpha\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else}\, (\mathsf{ask!}(c) \rightarrow \alpha(A) + \mathsf{ask!}(true) \rightarrow \alpha(B))$$

If we substitute the ask! agent by the original ask, then the body agent (A or B) is executed in the concrete model in the current time instant, whereas, in the abstract model, a delay of one time unit is introduced, which enables other agents that could be eventually executed concurrently to modify the store prior to the body execution. This might cause a totally incorrect behavior of the implementation w.r.t. the abstract semantics. In Figure 9, we illustrate this undesired situation for the abstraction of $\mathsf{now}\,(X = 4)\,\mathsf{then}\,\mathsf{A}\,\mathsf{else}\,\mathsf{now}\,(Y = 2)\,\mathsf{then}\,\mathsf{B}\,\mathsf{else}\,\mathsf{C}$. In the first trace, we show the behavior according to the abstract semantics proposed in the previous section, whereas the second trace illustrates the behavior by using the abstract ask^α agent.

Abstract trace

STORE	AGENTs
	now$^\alpha$(X=4) then A else ask!(X=4) $\rightarrow$ A + ask!($true$) $\rightarrow$now$^\alpha$(Y=2) then B else ask!(Y=2) $\rightarrow$ B + ask!($true$) $\rightarrow$ C || tell$^\alpha$(Y=2)
Y=2	C

Abstract trace with ask$^\alpha$

STORE	AGENTs
	now$^\alpha$(X=4) then A else ask$^\alpha$(X=4) $\rightarrow$ A + ask$^\alpha$($true$) $\rightarrow$now$^\alpha$(Y=2) then B else else ask$^\alpha$(Y=2) $\rightarrow$ B + ask$^\alpha$($true$) $\rightarrow$ C || tell$^\alpha$(Y=2)
Y=2	now$^\alpha$(Y=2) then B else ask$^\alpha$(Y=2) $\rightarrow$ B + ask$^\alpha$($true$) $\rightarrow$ C
Y=2	B

Figure 9. The problem of the elimination of the ask! agent

We propose a solution to this problem which consists of performing a preprocessing which is then used to transform the original abstract program (with the ask! agent) into another one where the ask! agent does not occur. The idea is to "expand the time" in the transformed program in order to synchronize all actions.

In order to formalize our transformation, we first analyze the α-program and annotate each timing agent (a tell$^\alpha$, ask$^\alpha$ or *procedure call* agent) with an integer number k that represents the relative depth of the agent within the program. The annotated version of agent A is denoted by A_k. We also need to store the maximum depth (K) of the agent in the whole specification (and not only in the corresponding clause definition). This allows us to determine how many delays ($K - k$) must be introduced for each agent, each delay being associated to a simple ask agent.

In the original semantics of tccp($\Theta, \vdash$), when the store does not entail any condition in the guards of the choice agent, then the agent suspends and it is tried again in the subsequent time instant. In the abstract semantics, if the choice agent suspends, then we have to introduce the appropriate number of delays in order to ensure that the choice agent is retried in the correct time instant. During the annotation process, for the transformation of a *choice* $\sum_{i=0}^{n} \mathsf{ask}^\alpha(c_i) \rightarrow A_i$, we introduce an integer string label l on the arrow ($\sum_{i=0}^{n} \mathsf{ask}^\alpha(c_i) \xrightarrow{l} A_i$), which univocally distinguishes each occurrence of the choice agent in the program. We also need to record the depth k to which this agent occurs within the program. We define the annotation of a program as follows:

Definition 19 (Annotation Function) *Given a* tccp *program P of the form $D.A$, the annotated program P_k is obtained by recursively applying the following labelling function λ:*

$\lambda(P) = f(0, 0, D).f(0, 0, A)$

$$f(k,l,P) = \begin{cases} f(k,l,D), f(k,l,D) & P = D, D \\ p(x) \colon - f(k,l,A) & P = p(x) \colon - A \\ \mathsf{stop}^{\alpha}_{k} & P = \mathsf{stop}^{\alpha} \\ \mathsf{tell}^{\alpha}(c)_k & P = \mathsf{tell}^{\alpha}(c) \\ \sum_{i=0}^{n} \mathsf{ask}^{\alpha}(c_i)_k \xrightarrow{l} f(k, l'.(j+1), A_i) & P = \sum_{i=0}^{n} \mathsf{ask}^{\alpha}(c_i) \rightarrow A_i \ \ \textit{and}\ l = l'.j \\ \mathsf{now}^{\alpha}\ c\ \mathsf{then}\ f(k,l,A)\ \mathsf{else} & P = \mathsf{now}^{\alpha}\ c\ \mathsf{then}\ A\ \mathsf{else} \\ \quad \mathsf{ask}^{\alpha}(c) \rightarrow f(k+1,l,A) + & \quad \mathsf{ask!}(c) \rightarrow A + \\ \quad \mathsf{ask}^{\alpha}(true) \rightarrow f(k+1,l,B) & \quad \mathsf{ask!}(true) \rightarrow B \\ f(k, 1.l, A) || f(k, 2.l, B) & P = A || B \\ \exists X\ f(k,l,A) & P = \exists X\ A \\ p(x)_k & P = p(x) \end{cases}$$

Note that the annotation function only affects the *tell*, *choice* and *procedure call* agents since only in these cases a delay must be introduced. The remainder agents run instantaneously, both in the original and in the transformed program. Figure A.1 (in Appendix A) shows the annotated program resulting from applying the λ function to the α-program in Figure 5.

5.3 The source-to-source implementation of the abstract semantics

In the following, we complete the source-to-source transformation by compiling the abstract agents into $\mathsf{tccp}(\wp(\Theta), \vdash^{+})$. This is achieved by introducing the necessary delays in the abstract program following the labelling described in Section 5.2 and transforming the abstract agents as shown in Section 5.1. Notation $\mathsf{ask}^{K-k} \rightarrow$ indicates the replication $K-k$ times of the agent structure $\mathsf{ask}(true) \rightarrow$. We provide the transformation for each abstract agent in Figure 10. Even if agent now^{α} can be implemented by means of tccp agent now, as shown in Section 5.1, for the sake of simplicity we prefer not to remove it from the transformation shown in Figure 10. [4]

Given program P of the form $D.A$, where D is a set of procedure declarations $\bigcup_{i=1}^{n}\{p_i\}$,

$$T(P) = D'.T(A) \text{ where } D' = \bigcup_{i=1}^{n}\{T(p_i)\}$$

and the transformation for each procedure p_i of the form $p(x)$:-B is

$$T(p(x)\text{:-}B) = \{p(x)\text{:-}T(B)\} \cup D_{aux}$$

where D_{aux} is the set of auxiliary procedures which are introduced by the transformation of agent B.

The transformed program obtained for our leading example can be seen in Figure A.2 (in Appendix A). Note the transformation of choices of the form $\mathsf{ask}(true)$ does not need procedure αchoice, as explained in Section 5.1.

[4] In the Figure 10, constraint $c_0; \ldots; c_n$ in agent now^{α} denotes the abstract store $\{\{c_1\}, \cdots \{c_n\}\}$ which may be different from $c_1 \vee \cdots \vee c_n$. See [32].

Stop agent. $T(\mathsf{stop}^{\alpha}_{k}) = \mathsf{ask}^{K-k} \rightarrow \alpha stop()$
where the auxiliary procedure $\alpha stop()$ is $\alpha stop()$:-$\alpha stop()$.

Tell agent. $T(\mathsf{tell}^{\alpha}(c)_k) = \mathsf{ask}^{K-k} \rightarrow \mathsf{tell}(c)$.

Choice agent.

$T(\sum_{i=0}^{n} \mathsf{ask}^{\alpha}(c_i)_k \xrightarrow{l} A_i) = \mathsf{ask}^{(K-k)-1} \rightarrow \alpha choice_{k,l}(c_0;\ldots;c_n, A_0,\ldots,A_n)$

$\alpha choice_{k,l}\ (c_0;\ldots;c_n, A_0,\ldots,A_n)$:-
$\quad \mathsf{now}^{\alpha}\ (c_0;\ldots;c_n)$ then $\sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow T(A_i)$
$\quad$ else $[\mathsf{ask}^{K-k} \rightarrow \alpha choice_{k,l}(c_0;\ldots;c_n, A_0,\ldots,A_n)\ ||$
$\quad\quad (\sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow T(A_i)\ + \mathsf{ask}(\neg c_0 \wedge \cdots \wedge \neg c_n) \rightarrow \mathsf{stop})]$

Conditional agent.

$T(\mathsf{now}^{\alpha}\, c\, \mathsf{then}\, A\, \mathsf{else}\, (\mathsf{ask!}(c) \rightarrow B + \mathsf{ask!}(true) \rightarrow C)) =$
$\quad \mathsf{now}^{\alpha}\, c\, \mathsf{then}\, T(A)\, \mathsf{else}\, (\mathsf{ask}(c) \rightarrow T(B) + \mathsf{ask}(true) \rightarrow T(C))$

Parallel agent. $T(A||B) = T(A)||T(B)$.

Hiding agent. $T(\exists x\, A) = \exists x\, T(A)$, where x is a variable.

Procedure Call agent. $T(p(x)_k) = \mathsf{ask}^{K-k}\!\rightarrow p(x)$ where x is a variable of the constraint system and there exists a procedure declaration as $p(x)$:-A.

Figure 10. Implementation of tccp^{α}

Now we are ready to demonstrate the correctness of this program transformation for the standard observable of derived constraints in non-erroneous computations: we prove the equivalence of the observable before and after the program transformation.

Given a program P and an initial configuration $\langle \Gamma_0, st_0 \rangle$, the observable set Ob of P w.r.t. semantics $\mathcal{O}$ is the set $\{st_0 \cdot st_1 \cdots \mid t = \langle \Gamma_0, st_0 \rangle \longrightarrow \langle \Gamma_1, st_1 \rangle \longrightarrow\ \in \mathcal{O}(P)(\langle \Gamma_0, st_0 \rangle) and\ t\ is$ non-erroneous$\}$ of all sequences of stores that can be extracted from the non-erroneous traces of P . The abstract observable set Ob^{α} is defined in the obvious way w.r.t. semantics $\mathcal{A}'$. Similarly, the set of observable of the transformed program $T(\alpha(P))$ is defined as $Ob^{\tau} = \{sst_{0*(K+1)} \cdot sst_{1*(K+1)} \cdot sst_{2*(K+1)} \cdots \mid \langle \Gamma_0, sst_0 \rangle \longrightarrow_{\alpha} \langle \Gamma_1, sst_1 \rangle \longrightarrow_{\alpha} \langle \Gamma_2, sst_2 \rangle \longrightarrow_{\alpha} \cdots \in \mathcal{O}(T(\alpha(P))(\langle \alpha(\Gamma_0), \alpha(st_0) \rangle)))\}$.

Theorem 20 *Consider a* tccp *program P and an initial configuration $\langle \Gamma_0, st_0 \rangle$. Let $\alpha(P)$ be the program resulting from applying the α-transformation to P, and $T(\alpha(P))$ the resulting program from applying the T transformation to $\alpha(P)$. Then $Ob^{\alpha}(\alpha(P))(\langle \alpha(\Gamma_0), \alpha(st_0) \rangle) = Ob^{\tau}(T(\alpha(P)))(\langle \alpha(\Gamma_0), \alpha(st_0) \rangle)$.*

As shown in Figure 11, the transformation process is given in two steps: the abstraction α followed by implementation $T \circ \lambda$ where λ is the annotation function of Definition 19 and T is the transformation given in Figure 10. Now, assume that an injective mapping $\iota : \rho(\wp(\Theta)) \rightarrow \Theta$ exists such that $\forall sst \in \rho(\wp(\Theta))$, if $st \in sst, c \in \mathcal{C}$ and $st \vdash c$ then $\iota(sst) \vdash c$. That is, abstract stores can be represented in terms of the concrete constraint system and, in addition, $\vdash^{+}$ may be expressed by using $\vdash$. Then, the abstraction process given by $\iota \circ T \circ \lambda \circ \alpha$ is a full source-to-source transformation.

$$\begin{tikzcd}
\mathsf{tccp}(\Theta,\vdash) \arrow[r, "\alpha"] & \mathsf{tccp}^{\alpha} \arrow[d, "\lambda"] \\
 & {} \arrow[d, "T"] \\
 & \mathsf{tccp}(\wp(\Theta),\vdash^{+}) \arrow[uul, hook, "\iota"']
\end{tikzcd}$$

Figure 11. Source-to-source transformation

5.4 Precision of the abstraction

In abstract model checking, the main interest is in the construction of reduced models, to partially solve the state-explosion problem. However, excessively abstracting the original model may lead to generating very imprecise abstract models containing traces which do not correspond to any real behavior, also called spurious traces.

As shown in the previous section, our strategy to achieve a correct abstract semantics has, at the same time, a payoff related to the precision of the abstract model, as witnessed by the following example where we show that spurious traces are contained in the abstract model.

Example 21 *Let $\rho = \{\{\{X = 2n\}|n \geq 0\}, \{\{X = 2n + 1\}|n \geq 0\}\}$ be the usual even-odd abstraction function for natural variable X. Consider the agents $A = \mathsf{ask}(X = 2) \rightarrow \mathsf{stop}$ and $B = \mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{tell}(Y = 2)$. Figure 12 shows the abstract tree of all possible abstract executions (obtained using the abstract semantics given in Section 4.2) of $\langle \alpha(A)||\alpha(B), sst\rangle$ where $sst = \{\{X = 2n\}|n \geq 0\}$ and $sst' = sst \sqcup \{Y = 2\}$.*

$$\begin{tikzcd}
 & \langle \mathsf{ask}^{\alpha}(X{=}2){-}{>}\mathsf{stop}^{\alpha}||\mathsf{ask}^{\alpha}(true){-}{>}\mathsf{tell}^{\alpha}(Y{=}2),sst\rangle \arrow[dl] \arrow[d] & \\
\langle \mathsf{stop}^{\alpha}||\mathsf{tell}^{\alpha}(Y{=}2),sst\rangle \arrow[ddd] & \langle \mathsf{ask}^{\alpha}(X{=}2){-}{>}\mathsf{stop}^{\alpha}||\mathsf{tell}^{\alpha}(Y{=}2),sst\rangle \arrow[ddd, Rightarrow, dotted] \arrow[dr, Rightarrow, dotted] & \\
 & & \langle \mathsf{ask}^{\alpha}(X{=}2){-}{>}\mathsf{stop}^{\alpha},sst'\rangle \arrow[loop above] \arrow[dd, Rightarrow, dotted] \\
 & & \\
\langle \mathsf{stop}^{\alpha},sst'\rangle & \langle \mathsf{stop}^{\alpha},sst'\rangle & \langle \mathsf{stop}^{\alpha},sst'\rangle
\end{tikzcd}$$

Figure 12. Abstract execution of $\langle \alpha(A)||\alpha(B), sst\rangle$

Note that agent $\mathsf{ask}^{\alpha}(X = 2)$ may evolve using rules **R2** *and* **R2'**, *where the second one simulates the possible agent suspension. On the other hand, in the concrete model, there are only two possible execution paths showed below, which correspond to the concretizations of sst given by $\{X{=}2\}$ and $\{X{=}2n\}$ with $n{\neq}1$:*

$$\langle \mathsf{ask}(X{=}2) \rightarrow \mathsf{stop}||\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{tell}(Y{=}2), \{X{=}2\}\rangle \longrightarrow$$
$$\langle \mathsf{tell}(Y{=}2), \{X{=}2\}\rangle \longrightarrow \langle \mathsf{stop}, \{X{=}2, Y{=}2\}\rangle$$

$$\langle \mathsf{ask}(X{=}2) \rightarrow \mathsf{stop}||\mathsf{ask}(\mathsf{true}) \rightarrow \mathsf{tell}(Y{=}2), \{X{=}2n\}\rangle \longrightarrow$$
$$\langle \mathsf{ask}(X{=}2) \rightarrow \mathsf{stop}||\mathsf{tell}(Y{=}2), \{X{=}2n\}\rangle \longrightarrow \langle \mathsf{ask}(X{=}2) \rightarrow \mathsf{stop}, \{X{=}2n, Y{=}2\}\rangle$$

Thus, we can observe that the abstract tree of Figure 12 contains many spurious traces (those that end with a dotted double arrow), i.e., traces that do not correspond to any concrete execution. For instance,

$$\langle \mathsf{ask}^{\alpha}(X = 2) \rightarrow \mathsf{stop}^{\alpha}||\mathsf{ask}^{\alpha}(\mathsf{true}) \rightarrow \mathsf{tell}^{\alpha}(Y = 2), sst\rangle \longrightarrow_{\alpha}$$
$$\langle \mathsf{ask}^{\alpha}(X = 2) \rightarrow \mathsf{stop}^{\alpha}||\mathsf{tell}^{\alpha}(Y = 2), sst\rangle \longrightarrow_{\alpha} \langle \mathsf{stop}^{\alpha}, sst \sqcup Y = 2\rangle$$

is a spurious trace. Also note that the suspension of agent $\alpha(A)$ in the first step of the erroneous trace is inconsistent with the non-suspension of $\alpha(A)$ in the second step.

In order to avoid these spurious traces, we intend to restrict the application of rules **R2** and **R2'** in some specific situations. Roughly speaking, we do not want to re-consider rule **R2** to be applied to the new configuration until a parallel agent has introduced in the store information which might affect the satisfiability of the guards of the choice agent.

We formalize this improvement as follows. Rule **R2'** is replaced with **R2***, which substitutes the agent ask^α by ask^* so that **R2** cannot be applied until the execution of another agent unblocks it. We instrument this by a new rule **R8'**, which substitutes the auxiliary agent ask^* back to the original ask. Note that rule **R8** doesn't apply to agent ask^*.

$$\boxed{\begin{array}{ll}
\mathbf{R2}^* & \langle \sum_{i=0}^{n} \mathsf{ask}^\alpha(c_i) \to A_i, sst\rangle \longrightarrow_\alpha \langle \sum_{i=0}^{n} \mathsf{ask}^*(c_i) \to A_i, sst\rangle \quad \text{if } sst \not\vdash^- \{\{c_1\},\cdots,\{c_n\}\} \\
\mathbf{R8'} & \dfrac{\langle B, sst\rangle \longrightarrow_\alpha \langle B', sst'\rangle}{\langle \sum_{i=0}^{n} \mathsf{ask}^*(c_i) \to A_i)||B, sst\rangle \longrightarrow_\alpha \langle \sum_{i=0}^{n} \mathsf{ask}^\alpha(c_i) \to A_i||B', sst'\rangle} \quad \text{if } sst \neq sst'
\end{array}}$$

Figure 13. An improved abstract operational semantics of tccp

Example 22 (Example 21 continued) *With the new abstract semantics we get rid of the rightmost spurious trace of Figure 12 as shown in the following figure.*

$$\begin{tikzcd}
 & \langle \mathsf{ask}^\alpha(X{=}2){-}{>}\mathsf{stop}^\alpha||\mathsf{ask}^\alpha(true){-}{>}\mathsf{tell}^\alpha(Y{=}2),sst\rangle \arrow[dl] \arrow[d] & \\
\langle \mathsf{tell}^\alpha(Y{=}2),sst\rangle \arrow[dd] & \langle \mathsf{ask}^*(X{=}2){-}{>}\mathsf{stop}^\alpha||\mathsf{tell}^\alpha(Y{=}2),sst\rangle \arrow[d] & \\
 & \langle \mathsf{ask}^\alpha(X{=}2){-}{>}\mathsf{stop}^\alpha,sst'\rangle \arrow[d, dashed] \arrow[dr] & \\
\langle \mathsf{stop}^\alpha,sst'\rangle & \langle \mathsf{stop}^\alpha,sst'\rangle & \langle \mathsf{ask}^*(X{=}2){-}{>}\mathsf{stop}^\alpha,sst'\rangle
\end{tikzcd}$$

Figure 14. A refinement of the abstract execution of $\langle\alpha(A)||\alpha(B), sst\rangle$

It is immediate that this is a correct improvement of the abstract semantics of Section 4.2. Unfortunately, the improvement is not complete as witnessed by the spurious trace left in Figure 14.

The imprecision in abstract model checking is strongly related to the problem of incompleteness in abstract interpretation [21,20] and its solution, i.e., the elimination of spurious traces in the abstract model, may be achieved by refining the abstract domain. One of the most interesting and practical applications of these ideas is the counterexample-guided abstraction refinement method [8,9]. A different approach for refining abstract models is [18], which uses under as well as over-approximation of formulas in order to automatically discard some fictitious traces added by the abstraction.

These refinement techniques are orthogonal to ours and may even be combined in order to achieve better performances. In the sequel, we focus on the Clarke *et* *al.*'s methodology and sketch this combination by means of the leading example. Even if [8,9] follow the *predicate abstraction* approach, it is not difficult to adapt the method to our setting.

Given two agents A and A' and a set of concrete stores S, we define the set $post[\alpha(A), \alpha(A')](S) = \{st' | \exists st \in S. \langle A, st \rangle \longrightarrow \langle A', st' \rangle\}$. Let us assume that $\langle A_1^\alpha, sst_1 \rangle \longrightarrow_\alpha \cdots \longrightarrow_\alpha \langle A_n^\alpha, sst_n \rangle$ is a trace in the abstract model. Then we can slightly modify the *SplitPATH algorithm* of [8,9] to detect whether this abstract trace is spurious, as sketch in Figure 15 below.

```
$S := sst_1$
$j := 1$
while ($S \neq \emptyset$ and $j < n$) {
        $j := j + 1$
        $S_{prev} := S$
        $S := post[A_{j-1}^\alpha, A_j^\alpha](S)$}
if $S \neq \emptyset$ then output "non spurious trace"
else output $j, S_{prev}$
```

Figure 15. The SplitPATH algorithm

Then, if the algorithm reports that the abstract trace is erroneous, it is possible to sketch a refinement of the abstraction, by partitioning the set sst_{j-1} into the three sets [8]: 1) dead-end states $S_D = S_{prev}$, 2) bad states $S_B = \{st \in sst_{j-1} | \exists st'. \langle A_{j-1}, st \rangle \longrightarrow \langle A_j, st' \rangle\}$, and 3) irrelevant states $S_I = sst_{j-1} - (S_D \cup S_B)$. The key idea of the refinement is to refine the abstraction so that the dead-end states and the bad states do not correspond to the same abstract state. Then the spurious trace would be eliminated.

Now, we illustrate how this method can be used to eliminate the spurious trace of Example 22.

Example 23 *By applying the SplitPATH algorithm to the abstract (spurious) trace of Figure 14, we successively assign the sets sst,* $\{\{X = 2n\} | n \neq 1\}$, $\{\{X = 2n, Y = 2\} | n \neq 1\}$ *and* $\emptyset$ *to variable* S. *This means that the analyzed trace is spurious. In order to refine the abstraction, we split* sst' *into the sets* $S_D = \{\{X = 2n, Y = 2\} | n \neq 1\}$, $S_B = \{\{X = 2, Y = 2\}\}$ *and* $S_I = \emptyset$. *Thus, to avoid this trace, a refinement* $\tilde{\rho}$ *of the abstraction function* ρ *should separate* $\{X = 2\}$ *from the rest of concrete stores. The most abstract definition for* $\tilde{\rho}$ *is* $\{\{\{X = 2\}\}, \{\{X = 2n\} | n \neq 1\}, \{\{X = 2n+1\} | n \geq 0\}\}$. *With this refinement, the abstract tree of Figure 14 is split into the two abstract trees of Figure 16 where the spurious trace has been removed. In this figure,* sst_1 *and* sst_2 *are the abstract stores* $\{\{X = 2\}\}$ *and* $\{\{X = 2n\} | n \neq 1\}$, *respectively.*

6 Abstracting properties

In order to check temporal properties in the abstract model, we need to provide a suitable approximation for them. In this section, we first recall the temporal linear logic introduced in [4] to analyze properties of tccp programs. Then, the standard satisfaction relation $\models$ which gives meaning to these temporal

$$\begin{tikzcd}
\langle \mathsf{ask}^{\alpha}(X{=}2){-}{>}stop||\mathsf{ask}^{\alpha}(true){-}{>}\mathsf{tell}^{\alpha}(Y{=}2), sst_1\rangle \arrow[d] & \langle \mathsf{ask}^{\alpha}(X{=}2){-}{>}\mathsf{stop}^{\alpha}||\mathsf{ask}^{\alpha}(true){-}{>}\mathsf{tell}^{\alpha}(Y{=}2), sst_2\rangle \arrow[d] \\
\langle \mathsf{tell}^{\alpha}(Y{=}2), sst_1\rangle \arrow[d] & \langle \mathsf{ask}^{*}(X{=}2){-}{>}\mathsf{stop}^{\alpha}||\mathsf{tell}^{\alpha}(Y{=}2), sst_2\rangle \arrow[d] \\
\langle \mathsf{stop}^{\alpha}, sst_1 \sqcup Y{=}2\rangle & \langle \mathsf{ask}^{\alpha}(X{=}2){-}{>}\mathsf{stop}^{\alpha}, sst_2 \sqcup Y{=}2\rangle \arrow[d] \\
 & \langle \mathsf{ask}^{*}(X{=}2){-}{>}\mathsf{stop}^{\alpha}, sst_2 \sqcup Y{=}2\rangle
\end{tikzcd}$$

Figure 16. Abstract executions with $\tilde{\rho}$

formulas is properly abstracted to meet the abstract models constructed to this point. Namely, we formalize two abstract relations $\models^{+}$ and $\models^{-}$ which over- and under-approximate $\models$, respectively.

6.1 Temporal Logic

Let $\langle \mathcal{C}, \vdash \rangle$ be a constraint system, and $c, d \in \mathcal{C}$. The original temporal logic of [4] introduces two modalities $\mathcal{K}(c)$ (knows) and $\mathcal{B}(c)$ (believes). $\mathcal{B}(c)$ is satisfied when the process assumes constraint c, and $\mathcal{K}(c)$ holds if c is known by the process. These modalities are interpreted on execution traces given as infinite sequences $\langle c_0, d_0\rangle \cdots \langle c_n, d_n\rangle \cdots$, where constraint c_i is the input from the external environment and d_i represents what is produced by the process itself. This permits to distinguish, at each time instant, the internal information produced by the program from the external information introduced by the environment. Modalities $\mathcal{K}$ and $\mathcal{B}$ are conceived to properly deal with these two data flows.

However, when analyzing programs by model checking, it is usual to assume that models are completely specified, i. e., the environment is considered a part of the model to be analyzed. Under this assumption, the external information does not exist independently, and the second component of each pair, in an execution trace, coincides with the first component of the following one, thus modalities $\mathcal{K}$ and $\mathcal{B}$ coincide. In the rest of this section, we simply consider sequences of constraints $s = c_0 \cdot c_1 \cdots c_n \cdots$ (that is, we disregard the component Γ of configurations $\langle \Gamma, c\rangle$ in the tccp execution traces). Note that, for the sequence s of constraints produced by a tccp execution, $c_i \subseteq c_{i+1}$ or equivalently $c_{i+1} \vdash c_i$.

The syntax of the temporal logic of [4] is

$$\phi ::= c \mid \neg\phi \mid \phi \wedge \phi \mid \exists x \phi \mid \bigcirc \phi \mid \phi \,\mathcal{U}\, \phi$$

The rest of standard propositional connectives and linear temporal operators are defined in terms of the above operators in the usual way: $\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\Diamond\phi = true\,\mathcal{U}\,\phi$ and $\Box\phi = \neg\Diamond\neg\phi$.

The truth value of temporal formulae is defined with respect to a sequence of constraints s and the constraint system $\langle \mathcal{C}, \vdash \rangle$. Each element in the sequence represents the store at a time instant. Given a sequence $s = c_0 \cdot c_1 \cdots c_n \cdots$, for all $i \geq 0$, we define $s^i = c_i \cdot c_{i+1} \cdots$. Following [4,5], given temporal formulas ϕ, ϕ_1 and ϕ_2, the satisfaction relation $\models$ is defined as follows

(1) $s \models c$ iff $c_0 \vdash c$
(2) $s \models \neg\phi$ iff $s \not\models \phi$
(3) $s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$
(4) $s \models \exists x\phi$ iff $s' \models \phi$, for some s' such that $\exists_x s = \exists_x s'$
(5) $s \models \bigcirc\phi$ iff $s^1 \models \phi$
(6) $s \models \phi_1 \mathcal{U} \phi_2$ iff for some $i \geq 0$. $s^i \models \phi_2$ and for all $0 \leq j < i$. $s^j \models \phi_1$

where notation $\exists_x s$ means $\exists_x c_0 \cdot \exists_x c_1 \cdots \exists_x c_n \cdots$.

6.2 Abstracting the satisfaction relation

The temporal logic defined above is parameterized w.r.t. the underlying constraint system $\langle \mathcal{C}, \vdash \rangle$. Given a constraint abstraction ρ, in Section 3.1 we have formalized two dual abstract constraint systems $\langle \wp(\mathcal{C}), \vdash_\rho^- \rangle$ and $\langle \wp(\mathcal{C}), \vdash_\rho^+ \rangle$. Following the same idea, in this section we introduce two satisfaction relations, called $\models^+$ and $\models^-$, which allow us to check properties in the abstract model. Namely, relation $\models^+$ is useful to refute properties of the concrete model, whereas $\models^-$ allows us to ensure that the concrete model does satisfy certain property.

Given $c \in \mathcal{C}$, an abstract formula is

$$\phi^\alpha ::= \{c\} \mid \neg\phi^\alpha \mid \phi^\alpha \wedge \phi^\alpha \mid \exists x\phi^\alpha \mid \bigcirc \phi^\alpha \mid \phi^\alpha \; \mathcal{U} \; \phi^\alpha$$

Since the transformations of a temporal formula ϕ into its abstract version ϕ^α, and vice-versa, are straightforward, in the rest of the section we use ϕ to denote both formulas.

The main difficulty in abstracting the satisfiability relation $\models$ is in dealing with the satisfiability of negated formulae (case (2) above). Note that, in the tccp context, negation of a constraint (or a formula) means that the store cannot deduce such a formula, but this does not necessarily mean that the contrary of the constraint is satisfied by the store. For example, if $\neg(x = 2)$ holds, $x \neq 2$ might not be held. In order to handle this, we define the abstract satisfiability of a negated formula $\models^+ \neg\phi$ in terms of $\models^- \phi$, and vice-versa.

Formally, given a sequence s^α of abstract stores of the form $s^\alpha = sst_0 \cdot sst_1 \cdots$ and a temporal formula ϕ, the abstract relations $\models_\rho^-$ and $\models_\rho^+$ are defined from $\vdash_\rho^-$ and $\vdash_\rho^+$ in the obvious way (as $\models$ was defined from $\vdash$ in subsection 6.1), except for case (2) which cannot be approximated in that way but by introducing two "inter-crossing" rules instead:

$$\begin{array}{lcl} s^\alpha \models_\rho^+ \neg\phi & \text{iff} & s^\alpha \not\models_\rho^- \phi \\ s^\alpha \models_\rho^- \neg\phi & \text{iff} & s^\alpha \not\models_\rho^+ \phi \end{array}$$

Relation $\models_\rho^+$ is an over-approximation of $\models$, which means that it is very *generous* when analyzing temporal properties because it is sufficient that $s \models \phi$ hold for a single concretization s of an abstract sequence of stores s^α, in order to have that $s^\alpha \models_\rho^+ \phi$. Dually, relation $\models_\rho^-$ is an under-approximation of $\models$, which means that it is necessary for $s \models \phi$ to hold for all concretizations s of s^α, in order for $s^\alpha \models_\rho^- \phi$. However, the logical negation of these relations does not match the expected meaning of negation in the tccp context, since combining the standard negation with $\models_\rho^+$ results in a relation $\not\models_\rho^+$ that is too demanding to mean over-approximation, and dually combining the standard negation with $\models_\rho^-$ results in a relation $\not\models_\rho^-$ that is too coarse to mean under-approximation. By interchanging the corresponding abstract satisfaction relations, we have countervailed this effect, as formalized in Proposition 25.

The following definition is auxiliary. The concretization of a sequence of abstract stores is defined as follows.

Definition 24 *Given an abstract sequence of stores* $s^\alpha = sst_0 \cdot sst_1 \cdots$ *where* $sst_i \in \wp(\wp(\mathcal{C}))$ *for* $i \geq 0$*, we define the concretization of* s^α *as the set* $\gamma(s^\alpha) = \{c_0 \cdot c_1 \cdots \mid c_i \in sst_i \text{ for all } i \geq 0\}$.

Proposition 25 *Given an abstract sequence of stores* $s^\alpha = sst_0 \cdot sst_1 \cdots$*, a sequence of concrete stores* $s = c_0 \cdot c_1 \cdots \in \gamma(s^\alpha)$ *and a temporal formula* ϕ*, then*

$$\begin{aligned} &(a)\ s \models \phi && \Rightarrow && s^\alpha \models_\rho^+ \phi \\ &(b)\ s^\alpha \models_\rho^- \phi && \Rightarrow && s \models \phi \end{aligned}$$

Now we can prove the correctness of our abstract model-checking methodology. That is, in the framework presented here, there are not false positives and, moreover, if we refute the property, then the refuting behavior of the concrete program is immediately guaranteed. By abusing notation, we write $P \models \phi$ if $s \models \phi$ for all $s \in Ob(P)$. Dually we write $P \not\models \phi$ if $s \not\models \phi$ for all $s \in Ob(P)$. We define $\alpha(P) \models^+ \phi$ and $\alpha(P) \models^- \phi$ analogously.

Theorem 26 *Given a* tccp *program* P *of the form* $D.\Gamma_0$*, an initial configuration* $\langle \Gamma_0, st_0 \rangle$*, and a constraint abstraction* ρ*. Then, given a temporal formula* ϕ*:*

(1) If $\alpha(P) \models_\rho^- \phi$ *then* $P \models \phi$*.*
(2) If $\alpha(P) \not\models_\rho^+ \phi$ *then* $P \models \neg\phi$*.*

Example 27 *Consider the following critical property for the photocopier program illustrated in Figure 1: "Photocopier is always turned-off when no message is sent by user during* `MIdle` *time units".*

We have divided this property into two parts which can be independently specified and proved:

- *Property 1: "Time to deadline is decreased by one each time that no message is sent by the user".*
  Using using the temporal logic presented in this section, this property is
  $\phi_1 = \Box\ (\exists V\, (\mathtt{fixedstate} \wedge (\mathtt{nomsg}\ \mathcal{U}\ \mathtt{newtime}\ \rightarrow\ \mathtt{decreasedbyone}))$
  where

(1) $\exists V$ *is* $\exists \mathtt{C}, \mathtt{T}, \mathtt{A}, \mathtt{E}, \mathtt{T1}, \mathtt{T2}, \mathtt{T}', \mathtt{T}''$ *and represents the selected (existentially quantified) program variables in a fixed point during the execution*
(2) `fixedstate` *is* $\mathtt{s(C, T, A, E)} \wedge \mathtt{T} = [\mathtt{T1}|\mathtt{T}']$*, meaning that the previous variables correspond to the same program iteration,* `T1` *being the lasting time to deadline,*

(3) $\texttt{nomsg}$ *is* $\neg\mathtt{C} = [\mathtt{on}|\_] \wedge \neg\mathtt{C} = [\mathtt{off}|\_] \wedge \neg\mathtt{C} = [\mathtt{c}|\_]$, *meaning that no message has been sent through C*

(4) $\texttt{newtime}$ *is* $\mathtt{T}' = [\mathtt{T2}|\mathtt{T}'']$, *which means that time has been updated, and*

(5) $\texttt{decreasedbyone}$ *is* $\mathtt{T2} = \mathtt{T1} - 1$.

- *Property 2: "Photocopier is always turned-off when deadline has expired".*
  A possible specification of this property is:
  $\phi_2 = \Box\ (\exists V(\texttt{fixedstate}\ \wedge(\texttt{deadline} \rightarrow \Diamond\ \texttt{turned-off}))$, *where*

(1) $\exists V$ *is* $\exists \mathtt{C}, \mathtt{T}, \mathtt{A}, \mathtt{E}, \mathtt{T}', \mathtt{E}'$ *and represents the selected (existentially quantified) program variables.*

(2) $\texttt{fixedstate}$ *is* $\mathtt{s}(\mathtt{C}, \mathtt{T}, \mathtt{A}, \mathtt{E}) \wedge \mathtt{E} = [\_|\mathtt{E}']$ *meaning that the previous variables correspond to the same program iteration.*

(3) $\texttt{deadline}$ *is* $\mathtt{T} = [\mathtt{0}|\mathtt{T}']$, *meaning that time has expired, and*

(4) $\texttt{turned-off}$ *is* $\mathtt{E}' = [\mathtt{stop}|-]$, *which means that photocopier has turned-off.*

These two properties can be independently checked in the abstract photocopier program by using, e.g., the constraint abstraction ρ given in Example 6. This is because, in Property 1, we are only interested to know whether there is a message in stream C*, whereas Property 2 does not refer to* C*. Therefore, if we can prove that $\alpha(P) \models_\rho^- \phi_1 \wedge \phi_2$ then, by Theorem 26, we obtain $P \models \phi_1 \wedge \phi_2$ as desired.*

Observe that constraints `s(E,C,A,T)` *in the photocopier program are used to bind together the values of system variables which correspond to the same program iteration, this being the (sequence of) actions given by the user request (through stream* C*) as well as the response of the photocopier when carrying out the corresponding task.*

Finally, if we fail in the attempt to prove or to refute a property by applying Theorem 26 (1) or (2), respectively, the abstract trace which causes the failure of the corresponding criterium can be delivered as a counterexample to the considered property. As we shown in Section 5.4, if this abstract trace is spurious, it can be used to refine the abstraction and improve the accuracy of our model checking methodology. We can iterate this process if necessary so that the criteria of Theorem 26 can be hopefully applied.

7 Related Work

The idea of using over and under approximations in specifications is due to Larsen and Thomsen in Modal Transition systems [26]. The main idea in this work is to construct a double labeled transition system, for modeling over- and under-approximations, respectively. A notion of refinement was proposed for abstracting models and then a combination of symbolic representation and theorem proving was instrumented to verify the properties. However, the authors did not consider how to obtain the initial abstract model of the system. The relationship between our construction and the underlying concrete constraint system can in fact be seen as an initial MTS refinement. In a similar sense, Section 6 can be also thought of as an independent rediscovery of using MTSs in the semantics of temporal logic, which was first presented in [6,22,24]. The use of MTSs as abstractions was also explored in [22] and [13]. In constrast to these approaches, where two abstract models, an over- and an under-approximation of the system, are constructed, we build just one over-approximated model though using both over and under-approximation to improve the accuracy of the abstract model. This allows us to verify universal properties as well as refute existential ones.

As we have shown, approximating tccp semantics is not routine work, as abstraction may modify some time aspects. This boils down to correctly simulating the suspension behavior, which makes the whole construction non straightforward. In fact, approximating suspension is also a major problem in ccp-like languages, as discussed in [32]. In Section 4.2, we instrument the semantics to avoid the problems of correctly simulating suspension by introducing new rules for correct abstract semantics of tccp. This allows us to overcome the lack of correctness of the abstract semantics w.r.t. the concrete one and to provide what we consider the best possible correct approximation of the concrete semantics which can be implemented in pure tccp.

8 Conclusions and Future Work

As it was highlighted in [32], in the context of concurrent constraint programming, the semantics of choice agent makes the construction of accurate abstract models more complex than in other paradigms. The mechanism for synchronization through blocking ask is, in some way, in contradiction to the conditions for the correctness of program abstraction needed to realize the abstraction. On the one hand, in order to simulate synchronization, we have to handle stronger constraints which guarantee that suspension in the abstract model implies suspension in the original one. On the other hand, as it is typical in abstract interpretation, weaker constraints must be added in order to correctly abstract the behavior of the tell agent. In tccp, the problem of abstracting synchronization is even more involved because all agents in execution are completely synchronized by the time notion of tccp.

This work provides a first foundation for effective model checking of tccp programs by means of correct abstract analysis and program transformation. We summarize the main contributions of the work as follows: 1) We have proposed an abstract model-checking methodology that mitigates the state explosion problem in tccp model checking. Due to the double, logical as well as temporal dimension of tccp, the abstraction of the conditional agent introduces some specific difficulties which have been solved by combining over and under-approximation in the abstract semantics. This idea is novel since only over-approximations are typically used when approximating models in the data abstraction approach; 2) We present the first formal proof for the total correctness of a refined abstract semantics which models the suspension behavior of processes; 3) We develop a source-to-source transformation for tccp programs that is the basis for a natural implementation of our method; 4) We have sketched two automatic improvements of the abstract semantics which allow us to get more accurate approximations. We have shown that both improvements do not interfere with the instrumented semantics; on the contrary the source-to-source implementation is shown to be independent of the considered abstraction; and 5) Finally, we have developed an approximation technique for checking the satisfiability of the temporal properties that must be verified, which completes our methodology.

There are several directions for future work. As this paper is mainly concerned with foundations, an implementation of the framework is desirable in order to support appropriate experimentation. Work on such an implementation has already started, and we expect some feedback that will enable further improvements in our method.

Acknowledgments

We thank the anonymous referees for the useful remarks and suggestions which helped to improve the paper.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *Proc. 2002 Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 2002.

[2] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *Proc. ACM Int'l Symp. POPL 2002*, pages 1–3, New York, 2002. ACM Press.

[3] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.

[4] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In G. Smolka, editor, *Proceedings of 8th Int'l Symp. on Temporal Representation and Reasoning*, pages 227–233. IEEE Computer Society Press, 2001.

[5] F. S. de Boer, M. Gabbrielli, and M. C. Meo. Proving Correctness of Timed Concurrent Constraint Programs. *ACM Transactions on Computational Logic*, 5(4), 2004. To appear.

[6] G. Bruns and P. Godefroid. Generalized Model Checking: Reasoning about Partial State Spaces. In C. Palamidessi, editor, *11th Int'l Conf. on Concurrency Theory CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2001.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[8] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, number 1855 in Lecture Notes in Computer Science, pages 154–169. Springer Verlag, 2000.

[9] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50:752–794, 2003.

[10] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16:1512-1542, 1994.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Int'l Symp. POPL*, pages 238–252, New York, 1977. ACM Press.

[12] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. 6th ACM Symp. POPL*, pages 269–282, New York, 1979. ACM Press.

[13] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[14] M. Falaschi, A. Policriti, and A. Villanueva. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language - I. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Selected papers from 2000 Joint Conference on Declarative Programming*, volume 48 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science Publishers, 2000.

[15] M. Falaschi and A. Villanueva. Automatic Verification of Timed Concurrent Constraint Programs. *Theory and Practice of Logic Programming*, 2004. To appear.

[16] M.M. Gallardo, J. Martínez, P. Merino, and E. Pimentel. αSPIN:a Tool for Abstract Model Checking. *Software Tools for Technology Transfer*, 5:165-184, 2003.

[17] M.M Gallardo, P. Merino, and E. Pimentel. A Generalized Semantics of Promela for Abstract Model Checking. *Formal Aspects of Computing*, 16:166–193, 2004.

[18] M.M. Gallardo, Merino P., and Pimentel E. Refinement of LTL Formulas for Abstract Model Checking . In *Proc. of Static Analysis Symposium (SAS 2002)*, number 2477 in Lecture Notes in Computer Science, pages 395–410. Springer Verlag, 2002.

[19] R. Giacobazzi, S.K. Debray, and G Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–247, 1995.

[20] R. Giacobazzi and E. Quintarelli. Incompleteness, Counterexamples, and Refinements in Abstract Model Checking. In *Proc. of Static Analysis Symposium (SAS 2001)*, number 2126 in Lecture Notes in Computer Science, pages 356–376. Springer Verlag, 2001.

[21] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Interpretations Complete. *Journal of the ACM*, 47(2):361–416, 2000.

[22] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-Based Model Checking Using Modal Transition Systems. In *12th Int'l Conf. on Concurrency Theory CONCUR 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2001.

[23] J. Hatcliff, M. Dwyer, C. Pasareanu, and Robby. Foundations of the Bandera Abstraction Tools. In *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 172–203, 2002.

[24] M. Huth, R. Jagadeesan, and D.A. Schmidt. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In David Sands, editor, *10th European Symposium on Programming ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2001.

[25] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *In Proceedings of the 14th Annual ACM Simp. POPL*, pages 111–119, 1987.

[26] Kim Guldstrand Larsen and Bent Thomsen. A Modal Process Logic. In *Third Annual Symposium on Logic in Computer Science, LICS '88*, pages 203–210. IEEE Computer Society Press, 1988.

[27] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6:1–35, 1995.

[28] M. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Proceedings of the 4th International Conference on Logic Programming*, pages 858–876, 1987.

[29] V. A. Saraswat. *Concurrent Constraint Programming Languages.* The MIT Press, Cambridge, MA, 1993.

[30] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proc. 9th Annual IEEE Symp. on Logic in Computer Science*, pages 71–80, New York, 1994. IEEE.

[31] V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proc. Eighteenth Annual ACM Symposium on Principles of Programming Languages POPL'91*, pages 333–352. ACM, 1991.

[32] E. Zaffanella, R. Giacobazzi, and G. Levi. Abstracting Synchronization in Concurrent Constraint Programming. *Journal of Functional and Logic Programming*, 1997(6), 1997.

A Source-to-source transformation. An example

Figure A.1 shows the annotated version of the α-program of Figure 5; note that the maximum depth of an agent in the original program is $K = 4$. The final, transformed program is given in Figure A.2.

```
user(C,A):- ask$^\alpha$(A=[free|_])$_0$ $\xrightarrow{0}$ tell$^\alpha$(C=[on|_])$_0$ +
    ask$^\alpha$(A=[free|_])$_0$ $\xrightarrow{0}$ tell$^\alpha$(C=[off|_])$_0$ +
    ask$^\alpha$(A=[free|_])$_0$ $\xrightarrow{0}$ tell$^\alpha$(C=[c|_])$_0$ +
    ask$^\alpha$(A=[free|_])$_0$ $\xrightarrow{0}$tell$^\alpha$($true$).
photocopier(C,A,MIdle,E,T):- $\exists$ Aux,Aux',T'(tell$^\alpha$(T=[Aux|T'])$_0$ || ask$^\alpha$($true$)$\xrightarrow{1}$(
   now$^\alpha$(T=[Aux|_] $\wedge$ Aux>0) then
       now$^\alpha$(C=[on|_]) then tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_0$ else
           ask!(C=[on|_]) $\rightarrow$ tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_1$ +
           ask!($true$) $\rightarrow$ now(C=[off|_]) then tell$^\alpha$(E=[stop|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_1$ else
                          ask!(C=[off|_]) $\rightarrow$ tell$^\alpha$(E=[stop|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_2$ +
                          ask!($true$) $\rightarrow$ now(C=[nc|_]) then tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_2$ else
                                          ask!(C=[c|_]) $\rightarrow$ tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_3$ +
                                          ask!($true$) $\rightarrow$ tell$^\alpha$(Aux'=Aux-1)$_3$ || tell$^\alpha$(T=[Aux'|_] $\wedge$ A=[free|_])$_3$
   else ask!(Aux>0) $\rightarrow$ now$^\alpha$ (C=[on|_]) then tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_1$ else
                        ask!(C=[on|_]) $\rightarrow$ tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_2$ +
                        ask!($true$) $\rightarrow$ now$^\alpha$(C=[off|_]) then tell$^\alpha$(E=[stop|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_2$ else
                                        ask!(C=[off|_]) $\rightarrow$ tell$^\alpha$(E=[stop|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_3$ +
                                        ask!($true$) $\rightarrow$ now$^\alpha$ (C=[nc|_]) then tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_3$ else
                                                        ask!(C=[c|_]) $\rightarrow$ tell$^\alpha$(E=[going|_] $\wedge$ T=[MIdle|_] $\wedge$ A=[free|_])$_4$ +
                                                        ask!($true$) $\rightarrow$ tell$^\alpha$(Aux'=Aux-1)$_4$ || tell$^\alpha$(T=[Aux'|_] $\wedge$ A=[free|_])$_4$ +
       ask!($true$) $\rightarrow$ tell$^\alpha$(E=[stop|_])$_1$ || tell$^\alpha$(A=[free|_])$_1$)).
system(MIdle,E,C,A,T):- $\exists$ E',C',A',T'(tell$^\alpha$(E=[_|E'])$_0$ || tell$^\alpha$(C=[_|C'])$_0$ ||
   tell$^\alpha$(A=[_|A'])$_0$ || tell$^\alpha$(T=[_|T'])$_0$ || user(C,A)$_0$ ||
   ask$^\alpha$($true$)$_0$ $\xrightarrow{2}$photocopier(C,A',MIdle,T',E')$_0$ ||
   ask$^\alpha$(A'=[free|_])$_0$ $\xrightarrow{3}$system(MIdle,E',C',A',T')$_0$) ||
   tell$^\alpha$(s(E',C',A',T')).
initialize(MIdle):- $\exists$ E,C,A,T(tell$^\alpha$(A=[free|_])$_0$ || tell$^\alpha$(T=[MIdle|_])$_0$ ||
                          tell$^\alpha$(E=[off|_])$_0$ || system(MIdle,E,C,A,T)$_0$) ||
                          tell$^\alpha$(s(E',C',A',T')).
```

Figure A.1. Annotation of the photocopier α-program

```
user(C,A):- ask(true)→ ask(true)→ ask(true)→  αchoice_0 (A=[free|_];A=[free|_];A=[free|_];A=[free|_],tell(C=[on|_]);tell(C=[off|_]);tell(C=[c|_]);tell(true)).
αchoice_0 (A=[free|_],A=[free|_],A=[free|_],A=[free|_],tell(C=[on|_]),tell(C=[off|_]),tell(C=[c|_]),tell(true)):-
         now^α (A=[free|_]  ∨  A=[free|_]  ∨  A=[free|_]  ∨  A=[free|_]) then
              (ask(A=[free|_])  →  ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(C=[on|_]) +
               ask(A=[free|_])  → ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(C=[off|_]) +
               ask(A=[free|_])  → ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(C=[c|_]) +
               ask(A=[free|_])  → ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(true))
         else ask(true)→ ask(true)→ ask(true)→ ask(true)→  αchoice_0 (A=[free|_],A=[free|_],A=[free|_],A=[free|_],tell(C=[on|_]),tell(C=[off|_]),tell(C=[c|_]),tell(true)) ||
              ((ask(A=[free|_])  →  ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(C=[on|_]) +
                ask(A=[free|_])  → ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(C=[off|_]) +
                ask(A=[free|_])  → ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(C=[c|_]) +
                ask(A=[free|_])  → ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(true)) +
               ask(¬A=[free|_]∧¬A=[free|_]∧¬A=[free|_]∧¬A=[free|_])→ stop)
photocopier(C,A,MIdle,E,T):-  ∃ Aux,Aux',T'(ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(T=[Aux|T'])  ||
                                              ask(true)→ ask(true)→ ask(true)→ ask(true)→ ask(true)→
                                                      now^α (Aux>0)  then
                                                          now^α (C=[on|_])  then ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) else
                                                               ask(C=[on|_])  → ask(true)→ ask(true)→ ask(true)→ tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) +
                                                               ask(true)→now^α (C=[off|_]) then ask(true)→ ask(true)→ ask(true)→ tell(E=[stop|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) else
                                                                             ask(C=[off|_])  → ask(true)→ ask(true)→ tell(E=[stop|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) +
                                                                             ask(true)→now^α (C=[c|_])  then ask(true)→ ask(true)→ tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) else
                                                                                           ask(C=[c|_])  → ask(true)→ tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) +
                                                                                           ask(true)  → ask(true)→ tell(Aux'=Aux-1)  || tell(T=[Aux'|_]  ∧  A=[free|_])
                                                      else ask(Aux>0)  → now^α (C=[on|_])  then ask(true)→ ask(true)→ ask(true)→ tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) else
                                                                        ask(C=[on|_])  → ask(true)→ ask(true)→ tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) +
                                                                        ask(true)  → now^α (C=[off|_])  then ask(true)→ ask(true)→ tell(E=[stop|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) else
                                                                                      ask(C=[off|_])  → ask(true)→ tell(E=[stop|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) +
                                                                                      ask(true)  → now^α (C=[c|_])  then ask(true)→ tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) else
                                                                                                     ask(C=[c|_])  → tell(E=[going|_]  ∧  T=[MIdle|_]  ∧  A=[free|_]) +
                                                                                                     ask(true)  → tell(Aux'=Aux-1)  || tell(T=[Aux'|_]  ∧  A=[free|_]) +
                                                         ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(E=[stop|_])  || ask(true)→ ask(true)→ ask(true)→ tell(A=[free|_]))
system(MIdle,E,C,A,T):-  ∃ E',C',A',T'(ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(E=[_|E'])  ||
   ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(C=[_|C'])  || ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(A=[_|A'])  ||
   ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(T=[_|T'])  || ask(true)→ ask(true)→ ask(true)→ ask(true)→user(C,A)  ||
   ask(true)→ask(true)→ ask(true)→ ask(true)→ ask(true)→  ask(true)→ ask(true)→ ask(true)→ ask(true)→photocopier(C,A',MIdle,T',E')  ||
   ask(true)→ ask(true)→ ask(true)→  αchoice_3 (A'=[free|_],system(MIdle,E',C',A',T'))  ||
   ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(s(E',C',A',T')).
αchoice_3 (A'=[free|_],system(MIdle,E',C',A',T'))  :-
         now^α (A'=[free|_])then  ask(A'=[free|_])→ ask(true)→ ask(true)→ ask(true)→ ask(true)→system(MIdle,E',C',A',T')
         else ask(true)→ ask(true)→ ask(true)→ ask(true)→  αchoice_3 (A'=[free|_],system(MIdle,E',C',A',T'))  ||
             (ask(A'=[free|_])→ ask(true)→ ask(true)→ ask(true)→ ask(true)→system(MIdle,E',C',A',T')  +
              ask(¬A'=[free|_])  → stop).
initialize(MIdle):-
          ∃ E,C,A,T(ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(A=[free|_])  || ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(T=[MIdle|_])  ||
          ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(E=[off|_])  || ask(true)→ ask(true)→ ask(true)→ ask(true)→  system(MIdle,E,C,A,T))  ||
          ask(true)→ ask(true)→ ask(true)→ ask(true)→ tell(s(E,C,A,T)).
```

Figure A.2. Transformation of the photocopier α-program

B Proofs

This appendix contains the proofs of all results of the paper.

Proposition 2 Relation $\vdash$ has the following properties:

(1) (Reflexivity) $\forall u \in \Theta . u \vdash u$.
(2) (Transitivity) $\forall u, v, w \in \Theta . u \vdash v, v \vdash w$ implies that $u \vdash w$.

Proof.

(1) (Reflexivity) It follows trivially from $C1$.
(2) (Transitivity) Consider $C_w \in w$. By hypothesis, since $v \vdash w$ we have that $v \vdash C_w$. Analogously, $u \vdash v$ implies that for all $C_v \in v.\, u \vdash C_v$, and since $v \vdash C_w$, by $C2$, we obtain $u \vdash C_w$. Therefore, for all $C_w \in w.\, u \vdash C_w$, that is, $u \vdash w$. □

Proposition 5 Let $\langle \mathcal{C}, \vdash \rangle$ be a simple constraint system and $\rho : \wp(\Theta) \to \wp(\Theta)$ be a constraint abstraction. Then,

- If $u \vdash v$, then $\{u\} \vdash^+_\rho \{v\}$.
- If $\{u\} \vdash^-_\rho \{v\}$, then $u \vdash v$.

Proof. By definition, since ρ is extensive. □

Proposition 7 Let $\langle \mathcal{C}, \vdash \rangle$ be a simple constraint system and $\rho : \wp(\Theta) \to \wp(\Theta)$ be a constraint abstraction. Then,

(1) (Reflexivity for $\vdash^+_\rho$) $\forall sst \in \wp(\Theta).\, sst \vdash^+_\rho sst$.
(2) (Transitivity for $\vdash^-_\rho$) $\forall sst_1, sst_2, sst_3 \in \wp(\Theta).\, sst_1 \vdash^-_\rho sst_2$ and $sst_2 \vdash^-_\rho sst_3$ implies that $sst_1 \vdash^-_\rho sst_3$.

Proof.

(1) Since ρ is extensive and $\vdash$ reflexive (Proposition 2), we have that $u \in \rho(sst)$ and $u \vdash u$. Therefore, $sst \vdash^+_\rho sst$.
(2) Consider $u_1 \in \rho(sst_1)$. By hypothesis, $sst_1 \vdash^-_\rho sst_2$ and $sst_2 \vdash^-_\rho sst_3$, hence, we have that there exists $u_2 \in sst_2$ such that $u_1 \vdash u_2$. Using the definition of $\vdash^-_\rho$ again, and since ρ is extensive, we have that there exists $u_3 \in sst_3$ such that $u_2 \vdash u_3$. Finally, by the transitivity of $\vdash$ (Proposition 2), we infer $u_1 \vdash u_3$, which implies that $sst_1 \vdash^-_\rho sst_3$. □

Proposition 10 For all $u, v \in \Theta$, and $sst_1, sst_2 \in \wp(\Theta)$, if $\rho(\{u\}) \subseteq sst_1$ and $\rho(\{v\}) \subseteq sst_2$ then $\rho(\{u \cup v\}) \subseteq sst_1 \sqcup^\rho sst_2$.

Proof. Immediate. □

Lemma 12 Consider a *tccp* program P and a constraint abstraction ρ satisfying **CC**. Let $\langle \Gamma, st\rangle$ and $\langle \Gamma', st'\rangle$ be two standard configurations such that $\langle \Gamma, st\rangle \longrightarrow \langle \Gamma', st'\rangle$. Then, for all $sst \in \wp(\Theta)$ with $\rho(\{st\}) \subseteq sst$ there exists $sst' \in \wp(\Theta)$ verifying that $\langle \alpha(\Gamma), sst\rangle \longrightarrow_\alpha \langle \alpha(\Gamma'), sst'\rangle$ and $\rho(\{st'\}) \subseteq sst'$.

Proof. We reason by induction on the standard agents Γ which do not suspend accordingly to the original tccp semantics.

- If $\langle \mathsf{tell(c)}, st\rangle \longrightarrow \langle \emptyset, st \cup \{c\}\rangle$. Define $sst' = sst \sqcup c$. Using **R1**, we obtain $\langle \mathsf{tell}^\alpha(c), sst\rangle \longrightarrow_\alpha \langle \emptyset, sst'\rangle$. Now, it is easy to prove that $\rho(\{st \cup \{c\}\}) \subseteq sst \sqcup c$.
- Applying the standard semantics of tccp, if $\langle \sum_{i=0}^n \mathsf{ask}(c_i) \rightarrow A_i, st\rangle \longrightarrow \langle A_j, st\rangle$ then $st \vdash c_j$. Thus, using **R2** (with $sst \vdash^+ c_j$) we have that $\langle \sum_{i=0}^n \mathsf{ask}^\alpha(c_i) \rightarrow \alpha(A_i), sst\rangle \longrightarrow_\alpha \langle \alpha(A_j), sst\rangle$.
- if $\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle \longrightarrow \langle A', st'\rangle$ and $st \vdash c$, then using the standard semantics of tccp, one of the following cases occurs:
  - $\langle A, st\rangle \longrightarrow \langle A', st'\rangle$. By induction, we have that $\langle \alpha(A), sst\rangle \longrightarrow \langle \alpha(A'), sst'\rangle$ and $\rho(\{st'\}) \subseteq sst'$. Now, if $sst \vdash^- c$, then applying **R4** we have that $\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else}\, (\mathsf{ask!}(c) \rightarrow \alpha(A) + \mathsf{ask!}(true) \rightarrow \alpha(B)), sst\rangle \longrightarrow_\alpha \langle \alpha(A'), sst'\rangle$. Otherwise, if $sst \not\vdash^- c$ and $st \vdash c$, then $sst \vdash^+ c$. Now, applying **R3a** (with $sst \vdash^+ c$) and **R6**, we obtain the same final configuration $(\langle \alpha(A'), sst'\rangle$.
  - If $\langle A, st\rangle \not\longrightarrow$, then from **CC** we know that $\langle \alpha(A), sst\rangle \not\longrightarrow_\alpha$. Now, using the same arguments as in the previous case, if $sst \vdash^- c$, by **R5** we obtain $\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else}(\mathsf{ask!}(c) \rightarrow \alpha(A) + \mathsf{ask!}(true) \rightarrow \alpha(B)), sst\rangle \longrightarrow_\alpha \langle \alpha(A), sst\rangle$. Otherwise, if $sst \not\vdash^- c$ then, following a similar reasoning as in the previous case, by applying **R3b** (with $sst \vdash^+ c$) and **R6**, we obtain the same result.
- if $\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle \longrightarrow \langle B', st'\rangle$ and $st \not\vdash c$, using the standard semantics of tccp, then one of the following cases occurs:
  - $\langle B, st\rangle \longrightarrow \langle B', st'\rangle$. By induction, we have that $\langle \alpha(B), sst\rangle \longrightarrow \langle \alpha(B'), sst'\rangle$ and $\rho(\{st'\}) \subseteq sst'$. By definition, if $st \not\vdash c$ then $sst \not\vdash^- c$. Therefore, applying **R6** and **R3** ($sst \vdash^+ true$) we have that $\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else}\, (\mathsf{ask!}(c) \rightarrow \alpha(A) + \mathsf{ask!}(true) \rightarrow \alpha(B)), sst\rangle \longrightarrow_\alpha \langle \alpha(B'), sst'\rangle$.
  - If $\langle B, st\rangle \not\longrightarrow$, then using **CC** we have that $\langle \alpha(B), sst\rangle \not\longrightarrow_\alpha$. Now, using similar arguments as in the previous case, if $sst \not\vdash^- c$, by **R7** and **R3a** ($sst \vdash^+ true$) we obtain $\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else}\, (\mathsf{ask!}(c) \rightarrow \alpha(A) + \mathsf{ask!}(true) \rightarrow \alpha(B)), sst\rangle \longrightarrow_\alpha \langle \alpha(B), sst\rangle$.
- if $\langle A||B, st\rangle \longrightarrow \langle A'||B', st'\rangle$ using the standard semantics of tccp, then one of the following cases occurs:
  - $\langle A, st\rangle \longrightarrow \langle A', st'_1\rangle$ and $\langle B, st\rangle \longrightarrow \langle B', st'_2\rangle$, and $st' = st'_1 \cup st'_2$. By induction, we have that $\langle \alpha(A), sst\rangle \longrightarrow_\alpha \langle \alpha(A'), sst'_1\rangle$ and $\rho(\{st'_1\}) \subseteq sst'_1$ and $\langle \alpha(B), sst\rangle \longrightarrow_\alpha \langle \alpha(B'), sst'_2\rangle$ and $\rho(\{st'_2\}) \subseteq sst'_2$. Therefore, applying **R8** we have that $\langle \alpha(A)||\alpha(B), sst\rangle \longrightarrow_\alpha \langle \alpha(A')||\alpha(B), sst'_1 \sqcup sst'_2\rangle$. Finally, using Proposition 10, we obtain that $\rho(\{st'_1 \cup st'_2\}) \subseteq sst'_1 \sqcup sst'_2$.

- Cases $\langle A, st\rangle \longrightarrow \langle A', st'_1\rangle$ and $\langle B, st\rangle \not\longrightarrow$, and $st' = st'_1$ and $\langle A, st\rangle \not\longrightarrow$ and $\langle B, st\rangle \longrightarrow \langle B', st'_2\rangle$, and $st' = st'_2$ are proved using induction, **CC** and rule **R9**.

- if $\langle \exists^{st_1} x\, A, st_2\rangle \longrightarrow \langle \exists^{st'} x\, A', st_2 \cup \exists x\, st'\rangle$, and $\rho(\{st_2\}) \subseteq sst_2$, then using the standard semantics of tccp, we have that $\langle A, st_1 \cup \exists x\, st_2\rangle \longrightarrow \langle A', st'\rangle$. Let $sst = st_1 \sqcup \exists x\, sst_2$. By construction, $\rho(\{st_1 \cup \exists x\, st_2\}) \subseteq sst$. Now, applying induction, there exists $sst' \in \wp(\wp(\mathcal{C}))$, such that $\langle \alpha(A), \exists x\, sst_2 \sqcup st_1\rangle \longrightarrow \langle \alpha(A'), sst'\rangle$, and $\rho(\{st'\}) \subseteq sst'$. Using **R10**, we obtain that $\langle \exists^{\{st_1\}} \alpha(A), sst_2\rangle \longrightarrow_\alpha \langle \exists^{sst'} x\, \alpha(A'), sst_2 \sqcup \exists x\, sst'\rangle$. Finally, by Proposition 10, we have that $\rho(\{st_2 \cup \exists x\, st'\}) \subseteq sst_2 \sqcup \exists x\, sst'$.
- Case $\langle p(x), st\rangle \longrightarrow \langle A, st\rangle$ is immediate due to the fact that the store is not modified during the execution of this agent. □

Theorem 13 Consider a *tccp* program P, an initial configuration $\langle \Gamma_0, st_0\rangle$ and a constraint abstraction ρ satisfying **CC**. Then, for each non-erroneous trace $t \in \mathcal{O}(P)(\langle \Gamma_0, st_0\rangle)$, there exists an abstract trace $t^\alpha \in \mathcal{A}_\rho(\alpha(P))(\langle \alpha(\Gamma_0), \rho(\{st_0\})\rangle)$ such that $\alpha(t) \sqsubseteq t^\alpha$.

Proof. Consider $t = \langle \Gamma_0, st_0\rangle \longrightarrow \langle \Gamma_1, st_1\rangle \longrightarrow \cdots$. The abstract trace $t^\alpha = t^\alpha_0 \longrightarrow_\alpha t^\alpha_1 \longrightarrow_\alpha \cdots$ is inductively constructed as follows:

- We define $t^\alpha_0 = \langle \Gamma_0, \rho(\{st_0\})\rangle$.
- Assume that $\langle \Gamma_i, st_i\rangle \longrightarrow \langle \Gamma_{i+1}, st_{i+1}\rangle$, and $\rho(\{st_i\}) \subseteq sst_i$. By Lemma 12, there exists an abstract store sst_{i+1} such that $\langle \alpha(\Gamma_i), sst_i\rangle \longrightarrow_\alpha \langle \alpha(\Gamma_{i+1}), sst_{i+1}\rangle$ and $\rho(\{st_{i+1}\}) \subseteq sst_{i+1}$. Therefore, we define $t^\alpha_{i+1} = \langle \alpha(\Gamma_{i+1}), sst_{i+1}\rangle$ and the result follows.
- Assume that $\langle \Gamma_i, st_i\rangle \not\longrightarrow$, and that $\rho(\{st_i\}) \subseteq sst_i$ then, by **CC**, $\langle \alpha(\Gamma_i), sst_i\rangle \not\longrightarrow_\alpha$. □

Lemma 16 Consider a *tccp* program P and a constraint abstraction ρ. Let $\langle \Gamma, st\rangle$ and $\langle \Gamma', st'\rangle$ be two standard configurations and $sst \in \wp(\Theta)$ such that $\rho(\{st\}) \subseteq sst$. Then,

(1) If $\langle \Gamma, st\rangle \not\longrightarrow$, then there exists $sst' \in \wp(\Theta)$ such that $\langle \alpha(\Gamma), sst\rangle \longrightarrow_\alpha \langle \alpha(\Gamma), sst'\rangle$ and $sst \subseteq sst'$.
(2) If $\langle \Gamma, st\rangle \longrightarrow \langle \Gamma', st'\rangle$, then there exists $sst' \in \wp(\Theta)$ such that $\langle \alpha(\Gamma), sst\rangle \longrightarrow_\alpha \langle \alpha(\Gamma'), sst'\rangle$ and $\rho(\{st'\}) \subseteq sst'$.

Proof.

(1) We reason by induction on the agents which may suspend:
 - Case $\Gamma = \mathsf{stop}$ is proved by **R0**, taking $sst = \rho(\{st\})$.
 - Consider $sst = \rho(\{st\})$. If $\langle \sum_{i=0}^n \mathsf{ask}(c_i) \rightarrow A_i, st\rangle \not\longrightarrow$, using the standard semantics of tccp, we have that for all $j.\ st \not\vdash c_j$ which implies that $sst \not\vdash^- \{\{c_1\}, \ldots, \{c_n\}\}$. Therefore, using **R2'** , we have that $\langle \sum_{i=0}^n \mathsf{ask}^\alpha(c_i) \rightarrow \alpha(A_i), sst\rangle \longrightarrow_\alpha \langle \sum_{i=0}^n \mathsf{ask}^\alpha(c_i) \rightarrow \alpha(A_i), sst\rangle$.
 - Finally, if $\langle A||B, st\rangle \not\longrightarrow$, then we have that $\langle A, st\rangle \not\longrightarrow$ and $\langle B, st\rangle \not\longrightarrow$.

Applying the previous results inductively this means that there exists $sst_1, sst_2 \in \wp(\Theta)$ such that $\langle \alpha(A), sst\rangle \longrightarrow_\alpha \langle \alpha(A), sst_1\rangle$, $\langle \alpha(B), sst\rangle \longrightarrow_\alpha \langle \alpha(B), sst_2\rangle$, $sst \subseteq sst_1$ and $sst \subseteq sst_1$. That is, $sst \subseteq sst_1 \cap sst_2$ which implies that $sst \subseteq sst_1 \sqcup sst_2$. In addition, using **R7**, we obtain $\langle \alpha(A)||\alpha(B), sst\rangle \longrightarrow_\alpha \langle \alpha(A)||\alpha(B), sst_1 \sqcup sst_2\rangle$

(2) Similar to Lemma 12, except for the following cases:

- if $\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle \longrightarrow \langle A, st\rangle$, then, by the standard semantics of tccp, we have that $st \vdash c$ and $\langle A, st\rangle \not\longrightarrow$. Then, using (1), there exists $sst' \in \wp(\Theta)$ such that $sst \subseteq sst'$ and $\langle \alpha(A), sst\rangle \longrightarrow_\alpha \langle \alpha(A), sst'\rangle$.
  - If $sst \vdash^- c$ then by rule **R4** we have that $\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else}\, (\mathsf{ask!}(c) \rightarrow \alpha(A) + \mathsf{ask!}(true) \rightarrow \alpha(B)), sst\rangle \longrightarrow_\alpha \langle \alpha(A), sst'\rangle$.
  - if $sst \not\vdash^- c$ then applying **R3** ($sst \vdash^+ c$) and **R6**, we obtain the same result.
- if $\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle \longrightarrow \langle B, st\rangle$, by the standard semantics of tccp, we have that $st \not\vdash c$ and $\langle B, st\rangle \not\longrightarrow$. Then, using (1), there exists $sst' \in \wp(\Theta)$ such that $sst \subseteq sst'$ and $\langle \alpha(B), sst\rangle \longrightarrow_\alpha \langle \alpha(B), sst'\rangle$. Since $st \not\vdash c$ then $sst \not\vdash^- c$, then, by rules **R6** and **R3** ($sst \vdash^+ true$) we have that $\langle \mathsf{now}^\alpha\, c\, \mathsf{then}\, \alpha(A)\, \mathsf{else} (\mathsf{ask!}(c) \rightarrow \alpha(A) + \mathsf{ask!}(true) \rightarrow \alpha(B)), sst\rangle \longrightarrow_\alpha \langle \alpha(B), sst'\rangle$.
- if $\langle A||B, st\rangle \longrightarrow \langle A||B', st'\rangle$, by the standard semantics of tccp, we have that $\langle A, st\rangle \not\longrightarrow$ and that $\langle B, st\rangle \longrightarrow \langle B', st'\rangle$. Now, on the one hand, by (1), there exists $sst'_1 \in \wp(\wp(\mathcal{C}))$ such that $sst \subseteq sst'_1$ and $\langle \alpha(A), sst\rangle \longrightarrow_\alpha \langle \alpha(A), sst'_1\rangle$. On the other hand, by induction there exists $sst'_2 \in \wp(\wp(\mathcal{C}))$ such that $\langle \alpha(B), sst\rangle \longrightarrow_\alpha \langle \alpha(B'), sst'_2\rangle$ and $\rho(\{st'\}) \in sst'_2$. Finally, applying rule **R8** we obtain $\langle \alpha(A)||\alpha(B), sst\rangle \longrightarrow_\alpha \langle \alpha(A)||\alpha(B'), sst'_1 \sqcup sst'_2\rangle$. To finish the proof, it is sufficient to note that $st \in sst \subseteq sst'_1$, $st' \in sst'_2$ and, since stores in tccp are monotonic, $st \subseteq st'$. Hence $st' \in sst'_1 \sqcup sst'_2$.
- Case $\langle A||B, st\rangle \longrightarrow \langle A'||B, st'\rangle$ is similar to the previous one. □

Theorem 17 Consider a *tccp* program P of the form $D.\Gamma_0$, an initial configuration $\langle \Gamma_0, st_0\rangle$ and a constraint abstraction ρ. For each non-erroneous trace $t \in \mathcal{O}(P)(\langle \Gamma_0, st_0\rangle)$, there exists an abstract trace $t^\alpha \in \mathcal{A}'_\rho(\alpha(P))(\langle \alpha(\Gamma_0), \rho(\{st_0\})\rangle)$ such that $\alpha(t) \sqsubseteq t^\alpha$.

Proof. We assume that each non-erroneous execution trace $t = t_0 \longrightarrow \cdots \in \mathcal{O}(P)(\langle \Gamma_0, st_0\rangle)$ is infinite (we infinitely repeat the last configuration if necessary.) Consider $t = \langle \Gamma_0, st_0\rangle \longrightarrow \langle \Gamma_1, st_1\rangle \longrightarrow \cdots$. Let $t^\alpha = t^\alpha_0 \longrightarrow_\alpha t^\alpha_1 \longrightarrow_\alpha \cdots$ be inductively constructed as follows:

- Let us define $t^\alpha_0 = \langle \Gamma_0, \rho(\{st_0\})\rangle$.
- Assume that $\langle \Gamma_i, st_i\rangle \longrightarrow \langle \Gamma_{i+1}, st_{i+1}\rangle$, and that $\rho(\{st_i\}) \subseteq sst_i$. By Lemma 16, there exists an abstract store sst_{i+1} such that $\langle \alpha(\Gamma_i), sst_i\rangle \longrightarrow \langle \alpha(\Gamma_{i+1}), sst_{i+1}\rangle$ and $\rho(\{st_{i+1}\}) \subseteq sst_{i+1}$. Therefore, we can define $t^\alpha_{i+1} = \langle \alpha(\Gamma_{i+1}), sst_{i+1}\rangle$, and the result follows. □

Theorem 20 Consider a *tccp* program P and an initial configuration $\langle \Gamma_0, st_0 \rangle$. Let $\alpha(P)$ be the program resulting from applying the α-transformation to P, and $T(\alpha(P))$ the resulting program from applying the T transformation to $\alpha(P)$. Then $Ob^{\alpha}(\alpha(P))(\langle \alpha(\Gamma_0), \alpha(st_0)\rangle) = Ob^{\tau}(T(\alpha(P)))(\langle \alpha(\Gamma_0), \alpha(st_0)\rangle)$.

Proof. We say that two configurations Γ and Δ are equivalent if they are syntactically equal. Let K be the maximum depth of an agent in the program. We need to prove that, at each execution point $n*(K+1)$, the nth configuration of a derivation in the α-semantics is equivalent to the $n*(K+1)$th configuration of the corresponding trace using the semantics of the transformed program. Let k be the annotated depth of agent A. Then, the annotated agent corresponding to A is denoted by A_k and d denotes the number of delays introduced during the transformation $(K-k)$. We proceed by structural induction on the agents of the α-program.

- If $\Gamma = \mathsf{stop}^{\alpha}_k$, then the transformed agent is $\Delta = \mathsf{ask}^d \rightarrow \mathsf{stop}$ where d is the number of delays. Following the correct semantics defined above, $\langle \Gamma, sst\rangle \longrightarrow_{\alpha} \langle \Gamma, sst\rangle$. On the other hand, $\langle \Delta, sst\rangle \longrightarrow_{\alpha} \langle \mathsf{ask}^{d-1} \rightarrow \mathsf{stop}, sst\rangle \longrightarrow^{k}_{\alpha} \cdots \langle \mathsf{stop}, sst\rangle$. Thus, the configuration at position $k+1$ coincides with the abstract configuration obtained in the α-program execution.
- If $\Gamma = \mathsf{tell}_k(c)$, then the transformed agent is $\Delta = \mathsf{ask}^d \rightarrow \mathsf{tell}(c)$. Following the semantics, $\langle \Gamma, sst\rangle \longrightarrow_{\alpha} \langle \emptyset, sst \sqcup c\rangle$ whereas $\langle \Delta, sst\rangle \longrightarrow_{\alpha} \langle \mathsf{ask}^{d-1} \rightarrow \mathsf{tell}(c), sst\rangle \longrightarrow^{d-1}_{\alpha} \cdots \langle \mathsf{tell}(c), sst\rangle \longrightarrow_{\alpha} \langle \mathsf{stop}, sst \sqcup c\rangle$, and the result follows.
- If $\Gamma = \sum_{i=0}^{n} \mathsf{ask}^{\alpha}(c_i)_k \rightarrow A_i$, then the transformed agent is $\Delta = \mathsf{ask}^{d-1} \rightarrow \alpha choice_{k,l}(c_0;\ldots;c_n, A_0;\ldots;A_n)$ where $\alpha choice_{k,l}(c_0;\ldots;c_n, A_0;\ldots;A_n)$:- $\mathsf{now}^{\alpha}\,(c_0;\ldots;c_n)\,\mathsf{then}\,(\sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow T(A_i)\,\mathsf{else}\,\Delta'$, and $\Delta' = \mathsf{ask}^d \rightarrow \alpha choice_{k,l}(c_0;\ldots;c_n, A_0;\ldots;A_n) || \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i + \mathsf{ask}(\neg c_0 \wedge \cdots \wedge \neg c_n) \rightarrow \mathsf{stop}$. Following the semantics we consider two cases:
  - · If $sst \vdash^{-} c_j$ with $0 \leq j \leq n$, then $\langle \Gamma, sst\rangle \longrightarrow_{\alpha} \langle A_j, sst\rangle$. On the other hand, we have that
    $\langle \Delta, sst\rangle \longrightarrow_{\alpha} \langle \mathsf{ask}^{d-2} \rightarrow \alpha choice_{k,l}(c_0;\ldots;c_n, A_0;\ldots;A_n), sst\rangle \longrightarrow^{d-2}_{\alpha} \langle \alpha choice_{k,l}(c_0;\ldots;c_n, A_0;\ldots;A_n), sst\rangle \longrightarrow_{\alpha} \langle T(A_j), sst\rangle$. By hypothesis, we assume that A_j is equivalent to $T(A_j)$ thus we obtain the expected result.
  - · If $sst \not\vdash^{+} c_j$ for all $0 \leq j \leq n$, then $\langle \Gamma, sst\rangle \longrightarrow_{\alpha} \langle \Gamma, sst\rangle$ and we have that $\langle \Delta, sst\rangle \longrightarrow_{\alpha} \langle \mathsf{ask}^{d-2} \rightarrow \alpha choice_{k,l}(c_0;\ldots;c_n, A_0;\ldots;A_n), sst\rangle \longrightarrow^{d-2}_{\alpha} \langle \Delta', sst\rangle \longrightarrow_{\alpha} \langle \mathsf{ask}^{d-1} \rightarrow \alpha choice_{k,l}(c_0,\ldots,c_n, A_0,\ldots,A_n), sst\rangle$, and the result is proved by induction.
  - · If $sst \not\vdash^{-} \{\{c_0\},\ldots,\{c_n\}\}$ but $sst \vdash^{+} c_j$ for some $0 \leq j \leq n$ then $\langle \Gamma, sst\rangle \longrightarrow_{\alpha} \langle \Gamma, sst\rangle$, and also $\langle \Gamma, sst\rangle \longrightarrow_{\alpha} \langle A_j, sst\rangle$, which correspond to the previous two cases.
- If $\Gamma = \mathsf{now}^{\alpha}\, c\, \mathsf{then}\, A_k \mathsf{else}(\mathsf{ask!}(c) \rightarrow A_{k+1} + \mathsf{ask!}(true) \rightarrow B_{k+1})$, then $\Delta = \mathsf{now}^{\alpha}\, c\, \mathsf{then}\, \mathsf{ask}^d \rightarrow T(A_k)\,\mathsf{else}\,(\mathsf{ask}(c) \rightarrow \mathsf{ask}^{d-1} \rightarrow T(A_{k+1}) + \mathsf{ask}(true) \rightarrow \mathsf{ask}^{d-1} \rightarrow T(B_{k+1}))$. We distinguish three cases:
  - · if $sst \vdash^{-} c$, then $\langle \Gamma, sst\rangle \longrightarrow_{\alpha} \langle A', sst\rangle$. By hypothesis we assume that A' is equivalent to the corresponding transformed agent $T(A')$. We have that $\langle \Delta, sst\rangle \longrightarrow_{\alpha} \langle \mathsf{ask}^{d-1} \rightarrow A, sst\rangle \longrightarrow^{d-1}_{\alpha} \langle T(A_k), sst\rangle \longrightarrow_{\alpha} \langle T(A'), sst\rangle$. Thus, the $k+1$th configuration is equivalent to $\langle A', sst\rangle$.

- · if $sst \not\vdash^- c$ and $sst \not\vdash^+ c$, then $\langle \Gamma, sst\rangle \longrightarrow_\alpha \langle B', sst\rangle$ and $\langle \Delta, sst\rangle \longrightarrow_\alpha \langle \mathsf{ask}^{d-1} \rightarrow T(B_{k+1}), sst\rangle \longrightarrow_\alpha^{d-1} \langle T(B_{k+1}), sst\rangle \longrightarrow_\alpha \langle T(B'), sst\rangle$, and the result holds.
- · if $sst \not\vdash^- c$ and $sst \vdash^+ c$ and reasoning in the same way, we obtain the expected result.

- If $\Gamma = A||B$, then the transformed agent Δ is $T(A)||T(B)$. By hypothesis, we assume that A is equivalent to $T(A)$ and that B and $T(B)$ are also equivalent. Moreover, we know that annotation does not affect this agent, hence both agents (A and B) have the same depth. Therefore the result follows directly.
- If $\Gamma = \exists A$ or $\Gamma = p(x)$, we proceed similarly to the previous case. □

Proposition 25 Given an abstract sequence of stores $s^\alpha = sst_0 \cdot sst_1 \cdots$, a sequence of concrete stores $s = c_0 \cdot c_1 \cdots \in \gamma(s^\alpha)$ and a temporal formula ϕ, then

$$(a)\ s \models \phi \quad \Rightarrow \quad s^\alpha \models_\rho^+ \phi$$

$$(b)\ s^\alpha \models_\rho^- \phi \quad \Rightarrow \quad s \models \phi$$

Proof. By induction on the structure of ϕ.

(1) Case $\phi = c \in \mathcal{C}$.
  (a) By definition, $s \models c \Rightarrow c_0 \vdash c$, and since $c_0 \in sst_0$ using the definition of $\vdash_\rho^+$, we have that $sst_0 \vdash_\rho^+ c$, that is, $s^\alpha \models_\rho^+ c$.
  (b) By definition, $s^\alpha \models_\rho^- c \Rightarrow sst_0 \vdash_\rho^- c$, which means that, for all $st \in sst_0$, $st \vdash c$. Since $c_0 \in sst_0$, we have that $c_0 \vdash c$, or equivalently, that $s \models c$.
(2) Case $\neg\phi$.
  (a) By definition of $\models$, $s \models \neg\phi \Rightarrow s \not\models \phi$. By induction hypothesis, $s \not\models \phi \Rightarrow s^\alpha \not\models_\rho^- \phi$ which is equivalent to $s^\alpha \models_\rho^+ \neg\phi$.
  (b) By definition, $s^\alpha \models_\rho^- \neg\phi \Rightarrow s^\alpha \not\models_\rho^+ \phi$. Now applying the induction hypothesis, we obtain that $s \not\models \phi$ or equivalently, that $s \models \neg\phi$.
(3) Case $\phi_1 \wedge \phi_2$.
  (a) If $s \models \phi_1 \wedge \phi_2$ then, by definition, we have that $s \models \phi_1$ and $s \models \phi_2$. Applying the induction hypothesis, we obtain $s^\alpha \models_\rho^+ \phi_1$ and $s^\alpha \models_\rho^+ \phi_2$ and, by the definition of $\models_\rho^+$, this leads to $s^\alpha \models_\rho^+ \phi_1 \wedge \phi_2$.
  (b) Similarly, using the rule (3) of the definitions for $\models$ and $\models_\rho^-$ and applying the induction hypothesis.
(4) Case $\exists x\phi$.
  (a) If $s \models \exists x\phi$ then, by definition, there exists a concrete sequence r such that $\exists_x r = \exists_x s$ and $r \models \phi$. Assume that $r = r_0 \cdot r_1 \cdots$, and construct the sequence $s^\alpha \cup r$ as the sequence of abstract stores $sst_0 \cup \{r_0\} \cdot sst_1 \cup \{r_1\} \cdots$. By construction, $r \in \gamma(s^\alpha \cup r)$ and since $r \models \phi$, by induction hypothesis, we obtain that $s^\alpha \cup r \models_\rho^+ \phi$. On the other hand, it is easy to prove that $\exists_x(s^\alpha \cup r) = \exists_x s^\alpha$: clearly, $\exists_x s^\alpha \sqsubseteq \exists_x(s^\alpha \cup r)$ and, inversely, since $s \in \gamma(s^\alpha)$ and $\exists_x s = \exists_x r$, we have that, for all $i \geq 0.\exists_x r_i \in \exists_x sst_i$, which implies that $\exists_x(s^\alpha \cup r) \sqsubseteq \exists_x s^\alpha$.
  Thus, we have found an abstract sequence of stores $s^\alpha \cup r$ such that $\exists_x(s^\alpha \cup r) = \exists_x s^\alpha$ and $s^\alpha \cup r \models_\rho^+ \phi$ which, by definition of $\models_\rho^+$, implies that $s^\alpha \models_\rho^+ \exists x\phi$.

(b) Assume now that $s^\alpha \models_\rho^- \exists x\phi$. Then, by definition of $\models_\rho^-$, there exists an abstract sequence $r^\alpha = r_0^\alpha \cdot r_1^\alpha \cdots$ such that $\exists_x r^\alpha = \exists_x s^\alpha$ and $r^\alpha \models_\rho^- \phi$. Since $\exists_x r^\alpha = \exists_x s^\alpha$ and, by hypothesis, $s = c_0 \cdot c_1 \cdots \in \gamma(s^\alpha)$, we can select for each $i \geq 0$ a constraint $r_i \in r_i^\alpha$ such that $\exists_x r_i = \exists_x c_i$. Let $r = r_0 \cdot r_1 \cdots$ be a sequence of stores. By construction, $r \in \gamma(r^\alpha)$ and by induction hypothesis, since $r^\alpha \models_\rho^- \phi$, we obtain that $r \models \phi$. Thus, we have found a concrete sequence r such that $\exists_x r = \exists_x s$ and $r \models \phi$ which, by definition of $\models$ implies that $s \models \exists x\phi$.

(5) Case $\bigcirc\phi$. Trivial considering that, if $s \in \gamma(s^\alpha)$ then $s^1 \in \gamma(s^{\alpha 1})$

(6) Case $\phi_1 \mathcal{U} \phi_2$. Similar to case (5) considering now that, if $s \in \gamma(s^\alpha)$ then $\forall j \geq 0 . s^j \in \gamma(s^{\alpha j})$. □

Theorem 26 Consider a tccp program P of the form $D.\Gamma_0$, an initial configuration $\langle \Gamma_0, st_0 \rangle$, and a constraint abstraction ρ. Then, given a temporal formula ϕ

(1) If $\alpha(P) \models_\rho^- \phi$ then $P \models \phi$.
(2) If $\alpha(P) \not\models_\rho^+ \phi$ then $P \not\models \phi$.

Proof. By definition, given $s \in Ob(P)$, there exists a concrete trace $t = \langle \Gamma_0, st_0 \rangle c \langle \Gamma_1, st_1 \rangle \longrightarrow \cdots \in \mathcal{O}(P)(\langle \Gamma_0, st_0 \rangle)$ such that $s = st_0 \cdot st_1 \cdots$. By Theorem 17, there exists $t^\alpha = \langle \alpha(\Gamma_0), \rho(\{st_0\}) \rangle \longrightarrow_\alpha \langle \alpha(\Gamma_1), sst_1 \rangle \longrightarrow_\alpha \cdots \in \mathcal{A}'(\alpha(P)(\langle \alpha(\Gamma_0), \rho(\{st_0\}) \rangle))$ such that $\alpha(t) \sqsubseteq t^\alpha$. Let $s^\alpha = \rho(\{st_0\}) \cdot sst_1 \cdots$. Then, $s_0 \in \rho(\{s_0\})$, and since $\alpha(t) \sqsubseteq t^\alpha$ we have $st_i \in sst_i$ for all $i > 0$. Therefore, $s \in \gamma(s^\alpha)$. Now, by applying Proposition 25, we obtain the two assertions:

- if $s^\alpha \models_\rho^- \phi$ then $s \models \phi$.
- if $s^\alpha \not\models_\rho^+ \phi$ then $s \not\models \phi$. □