Towards an Effective Decision Procedure for LTL formulas with Constraints

Marco Comini[1], Laura Titolo[1], and Alicia Villanueva[2] ⋆

[1] DIMI, Università degli Studi di Udine,
{marco.comini,laura.titolo}@uniud.it
[2] DSIC, Universitat Politècnica de València
villanue@dsic.upv.es

Abstract. This paper presents an ongoing work that is part of a more wide-ranging project whose final scope is to define a method to validate LTL formulas w.r.t. a program written in the timed concurrent constraint language *tccp*. *tccp* is a logic concurrent constraint language based on the concurrent constraint paradigm of Saraswat, thus notions such as non-determinism, dealing with partial information in states and the monotonic evolution of the information are inherent to *tccp* processes.

In order to check an LTL property for a process, our approach lays on the abstract diagnosis technique. The concluding step of this technique needs to check the validity of an LTL formula (with constraints) in an effective way.

In this paper, we present a decision method for the validity of temporal logic formulas (with constraints) built by our abstract diagnosis technique.

1 Introduction

Modeling and verifying concurrent systems by hand can be *really* complicated. Thus, the development of automatic formal methods is essential. One of the most known techniques for formal verification is model checking, that was originally introduced in [3,15] to automatically check if a finite-state system satisfies a given property. It consisted in an exhaustive analysis of the state-space of the system, therefore, the state-explosion problem was its main drawback. We are interested in an alternative way of validating a linear temporal formula ϕ w.r.t. a program P which does not require to build any model at all.

The *ccp* paradigm is different from other programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. With this notion, programs can deal with partial information by using an underlying constraint system that handles constraints on variables. Within this family, [6] introduced *tccp* by adding to the original *ccp* model the notion of time

⋆ This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN, ref. TIN 2010-21062-C02-0, and by Generalitat Valenciana, ref. PROMETEO2011/052.

and the ability to capture the absence of information. With these features, one can specify behaviors typical of reactive systems such as *timeouts* or *preemption* actions. All these features make *tccp* a suitable high-level language to model concurrent reactive systems.

As already said, our final goal is to define a method that avoids the need to build the model of a system in order to check the validity of some temporal property. Our proposal consists in an extension of the abstract diagnosis technique [5] where the abstract domain is formed by LTL formulas. The final step of our approach consists in checking whether a given formula, built from the program (abstract) semantics and the specification, is valid. By choosing an appropriate instance of the general framework, the resulting formula belongs to a fragment of an LTL logic with constraints (meaning that the logic is parametric w.r.t. a constraint system) that is decidable [17].

In this work, we present a decision procedure to check the validity of such formulas. As we show through this paper, the considered logic has some differences w.r.t. the classic LTL logic due to the constraint nature and to the fact that models for *tccp* programs have some special characteristics (inherited from the *ccp* paradigm).

The rest of the paper is organized as follows. In Section 2, we introduce the *tccp* language. The particularities of the language affect the definition of the decision procedure. Then, Section 3 defines the constraint system linear temporal logic. We give the intuition of the abstract diagnosis technique in Section 4. We do not provide all the details of the technique since they are beyond the scope of this paper. Our decision method is described in Section 5 and, finally, Section 6 concludes.

2 The small-step operational behavior of the *tccp* language

The *tccp* language [6] is particularly suitable to specify both reactive and time critical systems. As the other languages of the *ccp* paradigm [16], it is parametric w.r.t. a cylindric constraint system which handles the data information of the program in terms of constraints. The computation progresses as the concurrent and asynchronous activity of several agents that can (monotonically) accumulate information in a *store*, or query some information from it. Briefly, a cylindric constraint system[3] is an algebraic structure $\mathbf{C} = \langle \mathcal{C}, \leq, \otimes, \mathit{true}, \mathit{false}, \mathit{Var}, \tilde{\exists} \rangle$ composed of a set of constraints $\mathcal{C}$ such that $(\mathcal{C}, \leq)$ is a complete algebraic lattice where $\otimes$ is the *lub* operator and *false* and *true* are respectively the greatest and the least element of $\mathcal{C}$; *Var* is a denumerable set of variables and $\tilde{\exists}$ existentially quantifies variables over constraints. The *entailment* $\vdash$ is the inverse of order $\leq$.

[3] See [6,16] for more details on cylindric constraint systems.

Given a cylindric constraint system $\mathbf{C}$ and a set of process symbols Π, the syntax of agents is given by the grammar:

$$A ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid A \parallel A \mid \exists x\, A \mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A \mid \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ A \mid p(\vec{x})$$

where c, $c_1, \ldots, c_n$ are finite constraints in $\mathbf{C}$; $p_{/m} \in \Pi$, $x \in \mathit{Var}$ and $\vec{x} \in \mathit{Var} \times \cdots \times \mathit{Var}$. A *tccp* program P is an object of the form $D\,.\,A$, where A is an agent, called *initial agent*, and D is a set of *process declarations* of the form $p(\vec{x}) \coloneq A$ (for an agent A), where $\vec{x}$ denotes a generic tuple of variables.

The notion of time is introduced by defining a discrete and global clock. The ask, tell and process call agents take one time-unit to be executed.

Intuitively, the skip agent represents the successful termination of the agent computation. The $\mathsf{tell}(c)$ agent adds the constraint c to the current store and stops. It takes one time-unit, thus the constraint c is visible to other agents from the following time instant. The store is updated by means of the $\otimes$ operator of the constraint system. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$ consults the store and non-deterministically executes (at the following time instant) one of the agents A_i whose corresponding guard c_i holds in the current store; otherwise, if no guard is satisfied by the store, the agent suspends. The agent $\mathsf{now}\ c\,\mathsf{then}\ A_1\ \mathsf{else}\ A_2$ behaves in the current time instant like A_1 (respectively A_2) if c is (respectively is not) satisfied by the store. The satisfaction is checked by using the $\vdash$ operator of the constraint system. Note that this agent can process negative information: it can capture when some information is not present in the store since the agent A_2 is executed both when $\neg c$ is satisfied, but also when neither c nor $\neg c$ are satisfied. $A_1 \parallel A_2$ models the parallel composition of A_1 and A_2 in terms of maximal parallelism (in contrast to the interleaving approach of *ccp*), i.e., all the enabled agents of A_1 and A_2 are executed at the same time. The agent $\exists x\, A$ is used to make variable x local to A. To this end, it uses the $\tilde{\exists}$ operator of the constraint system. Finally, the agent $p(\vec{x})$ takes from D a declaration of the form $p(\vec{x}) \coloneq A$ and executes A at the following time instant. For the sake of simplicity, we assume that the set D of declarations is closed w.r.t. parameter names.

The *operational semantics* of *tccp* is formally described by a transition system $T = (\mathit{Conf}, \to)$. Configurations in Conf are pairs $\langle A, c\rangle$ representing the agent to be executed (A) and the current global store (c). The transition relation $\to$ models the passage of one time unit. For more details on the operational semantics of the language, the reader can consult [6].

The small-step behavior of a *tccp* program $D.A$ is formed by a set of traces in $\mathbf{C}^{\omega}$ representing how the global constraint store evolves at each time instant:

$$\begin{aligned}\mathcal{B}^{ss}[\![D.A]\!] \coloneq &\big\{c_0 \cdot c_1 \cdots c_n \cdots c_n \cdots \;\big|\, \langle A, c_0\rangle \to \langle A_1, c_1\rangle \to \ldots \to \langle A_n, c_n\rangle \not\to \big\} \cup \\ &\big\{c_0 \cdot c_1 \cdots c_i \cdots \big|\, \langle A, c_0\rangle \to \langle A_1, c_1\rangle \to \ldots \to \langle A_i, c_i\rangle \to \ldots \big\}\end{aligned}$$

The following program example is used in Section 4 to illustrate our abstract diagnosis technique for LTL formulas.

Example 1. The program consists of a single process declaration $D \coloneqq \{p(y) \coloneq A\}$ where the body of the process is defined as

$$A \coloneqq \exists x\, (\mathsf{now}\ y = 1\ \mathsf{then}\ (\mathsf{tell}(x = 5) \parallel p(y))\ \mathsf{else}\ \mathsf{tell}(y = 1))$$

Intuitively, variable x is defined local to the process $p(y)$. This process constrains variable x to the value 5 in the local store provided that $y = 1$ is entailed by the current constraint store. In such a case, it also recursively calls itself. If $y = 1$ does not hold in the current state, then it is ensured that it will hold at the following time instant (by means of the tell agent).

If we run the program with initial store $y = 1$, its behavior is represented by the trace $y = 1 \cdot (y = 1 \wedge x = 5) \cdots (y = 1 \wedge x = 5) \cdots$. We note that when the program ends, the last store is infinitely replicated (so all traces are infinite). Let us now consider the case when the initial store does not entail the guard in the conditional agent: if we run the program with initial store $y > 0$, then the program generates the trace $y > 0 \cdot y = 1 \cdots y = 1 \cdots$, where $y = 1$ is added at the second time instant.

3 Constraint System Linear Temporal Logic

In this section, we define a variation of the classical Linear Temporal Logic [13]. Following [14,7,8,17], the idea is to replace atomic propositions by constraints of the underlying constraint system. This logic is the basis for the definition of the abstract semantics needed in our abstract diagnosis technique.

Definition 1 (csLTL formulas). *Given a cylindric constraint system* $\mathbf{C}$*,* $c \in \mathbf{C}$ *and* $x \in \mathit{Var}$*, formulas of the* Constraint System Linear Temporal Logic *over* $\mathbf{C}$ *are defined by using the grammar:*

$$\phi \Coloneqq \dot{\mathit{true}} \mid \dot{\mathit{false}} \mid c \mid \dot{\neg}\, \phi \mid \phi \mathbin{\dot{\wedge}} \phi \mid \dot{\exists}_x\, \phi \mid \bigcirc \phi \mid \phi\, \mathcal{U}\, \phi.$$

We denote with csLTL *the set of all temporal formulas over* $\mathbf{C}$*.*

The formulas $\dot{\mathit{true}}$, $\dot{\mathit{false}}$, $\dot{\neg}\, \phi$, and $\phi_1 \mathbin{\dot{\wedge}} \phi_2$ have the classical logical meaning. The atomic formula $c \in \mathbf{C}$ states that c has to be entailed by the current store. $\dot{\exists}_x\, \phi$ is the existential quantification over the set of variables Var. $\bigcirc \phi$ states that ϕ holds at the next time instant, while $\phi_1\, \mathcal{U}\, \phi_2$ states that ϕ_2 eventually holds and in all previous instants ϕ_1 holds. In the sequel, we use $\phi_1 \mathbin{\dot{\vee}} \phi_2$ as a shorthand for $\dot{\neg}\, \phi_1 \mathbin{\dot{\wedge}} \dot{\neg}\, \phi_2$; $\phi_1 \mathbin{\dot{\rightarrow}} \phi_2$ for $\dot{\neg}\, \phi_1 \mathbin{\dot{\vee}} \phi_2$; $\diamond \phi$ for $\dot{\mathit{true}}\ \mathcal{U}\ \phi$ and $\square \phi$ for $\dot{\neg} \diamond \dot{\neg}\, \phi$. A *constraint formula* is an atomic formulas c or its negation $\dot{\neg}\, c$. Formulas of the form $\bigcirc \phi$ and $\dot{\neg} \bigcirc \phi$ are called *next* formulas. Finally, formulas of the form $\phi_1\, \mathcal{U}\, \phi_2$ (or $\diamond \phi$ or $\dot{\neg}(\square \phi)$) are called *eventualities*.

The truth of a formula $\phi \in \mathsf{csLTL}$ is defined w.r.t. a trace $s \in \mathbf{C}^\omega$. As usually done, given $s = c_0 \cdot c_1 \cdot c_2 \cdots \in \mathbf{C}^\omega$, s^i denotes the sub-sequence $c_i \cdot c_{i+1} \ldots$ and $s(i)$ denotes the i-th constraint c_i.

Definition 2. *For each $\phi, \phi_1, \phi_2 \in$ csLTL, $c \in \mathbf{C}$ and $s \in \mathbf{C}^\omega$, the satisfaction relation $\models$ is defined as:*

$$s \models \dot{true} \text{ and } s \not\models \dot{false} \tag{3.1a}$$

$$s \models c \qquad \text{iff } s(1) \vdash c \tag{3.1b}$$

$$s \models \dot{\neg}\, \phi \qquad \text{iff } s \not\models \phi \tag{3.1c}$$

$$s \models \phi_1 \mathbin{\dot{\wedge}} \phi_2 \qquad \text{iff } s \models \phi_1 \text{ and } s \models \phi_2 \tag{3.1d}$$

$$s \models \dot{\exists}_x\, \phi \qquad \text{iff exists } s' \text{ such that } \tilde{\exists}_x\, s' = \tilde{\exists}_x\, s \text{ and } s' \models \phi \tag{3.1e}$$

$$s \models \bigcirc \phi \qquad \text{iff } s^1 \models \phi \tag{3.1f}$$

$$s \models \phi_1 \,\mathcal{U}\, \phi_2 \qquad \text{iff } \exists i \geq 1.\, s^i \models \phi_2 \text{ and } \forall j < i.\, s^j \models \phi_1 \tag{3.1g}$$

We define $[\![\phi]\!] := \{s \mid s \models \phi\}$ and we say that ϕ is valid *if and only if $[\![\phi]\!] = \mathbf{C}^\omega$, and that ϕ is* satisfiable *if and only if $[\![\phi]\!] \neq \varnothing$.*

Let us show some temporal properties that can be expressed by csLTL formulas.

Example 2. The formula $\Box(x > 0 \mathbin{\dot{\rightarrow}} \bigcirc z > 1)$ expresses that always in the future, whenever $x > 0$ is entailed by the store, then $z > 1$ is entailed at the following time instant.

The formula $\dot{\exists}_x(\Diamond(y = 1 \mathbin{\dot{\wedge}} x = 5))$ expresses that, eventually in the future, $y = 1$ is entailed by the global constraint store and, at the same time instant, there exists a local variable x such that $x = 5$ is entailed by the local constraint store.

4 Abstract diagnosis of temporal properties

Abstract diagnosis is a semantic based method to identify bugs in programs. It was originally defined for logic programming [4] and then extended to other paradigms [1,2,9,5]. This technique is based on the definition of an abstract semantics for the program which must be a sound approximation of its behavior. Then, given an abstract specification $\mathcal{S}$ of the expected behavior of the program, the abstract diagnosis technique automatically detects the errors in the program by checking if the result of one computation of the semantics evaluation function (where the procedure calls are interpreted over $\mathcal{S}$) is "contained" in the specification itself.

A first approach to the abstract diagnosis of *tccp* was presented in [5] by using as specifications sets of abstract traces. The main drawback of that proposal was that specifications were given in terms of traces, thus they can be tedious to write. In order to overcome that problem, we have defined an abstract semantics (a semantics evaluation function) in terms of csLTL formulas. This allows us to express the intended behavior in a more compact way, by means of a csLTL formula.

The semantics evaluation function $\mathcal{A}[\![A]\!]$, given an agent A and an interpretation $\mathcal{I}$ (for the process symbols of A), builds a csLTL formula representing a correct approximation of the small-step behavior of A.[4] The semantics of the declarations is given in terms of the fixpoint of a semantics evaluation function $\mathcal{D}[\![D]\!]$ which associates to each procedure declaration $p(\vec{x})$ the logic disjunction of the abstract semantics of every agent A such that $p(\vec{x}) \mathrel{:-} A$ belongs to the declaration D (i.e., $\mathcal{D}[\![D]\!]_{\mathcal{I}}(p(\vec{x})) \coloneqq \dot{\bigvee}_{p(\vec{x}):-A\in D}\, \mathcal{A}[\![A]\!]_{\mathcal{I}}$).

The *abstract diagnosis* technique determines exactly the “originating” symptoms and, in the case of incorrectness, the faulty process declaration in the program. This is captured by the definitions of *abstractly incorrect process declaration* and *abstract uncovered element*. Informally, a process declaration D is abstractly incorrect if it derives a wrong abstract element $\phi_t \in \mathsf{csLTL}$ from the intended semantics $\mathcal{S}$. Dually, ϕ_t is uncovered if the declarations cannot derive it from the intended semantics.

We show here the main result of abstract diagnosis, which determines the form of formulas that we need to check for validity.

Theorem 1. *Consider a set of declarations D and an abstract specification $\mathcal{S}$.*

1. *If there are no abstractly incorrect process declarations in D (i.e., $\mathcal{D}[\![D]\!]_{\mathcal{S}} \dot{\to} \mathcal{S}$), then D is partially correct w.r.t. $\mathcal{S}$.*
2. *Let D be partially correct w.r.t. $\mathcal{S}$. If D has abstract uncovered elements then D is not complete.*

Therefore, in order to check partial correctness of a program, it is sufficient to check the implication $\mathcal{D}[\![D]\!]_{\mathcal{S}} \dot{\to} \mathcal{S}$.

Because of the approximation, it can happen that a (concretely) correct declaration is abstractly incorrect. Hence, abstract incorrect declarations are in general just a warning about a possible source of errors. However, an abstract correct declaration cannot contain an error; therefore, no (manual) inspection is needed for declarations which are not signalled. Moreover, it happens that all concrete errors—that are “visible”—are detected, as they lead to an abstract incorrectness or abstract uncovered.

Let us now illustrate how the technique works by means of a simple example.

Example 3. Assume we need to check that the program in Example 1 satisfies that the constraint $y = 1$ is eventually entailed by the store. The intended specification for the process p corresponding to this property is $\mathcal{S}(p(y)) \coloneqq \Diamond(y = 1)$.

The csLTL-semantics $\mathcal{D}$ for $p(y)$ using the given specification as interpretation is

$$\mathcal{D}[\![D]\!]_{\mathcal{S}}(p(y)) = \dot{\exists}_x \big((y = 1 \mathbin{\dot{\wedge}} \bigcirc x = 5 \mathbin{\dot{\wedge}} \bigcirc(\Diamond\, y = 1)) \mathbin{\dot{\vee}} (\dot{\neg}\, y = 1 \mathbin{\dot{\wedge}} \bigcirc y = 1)\big)$$

Note that the resulting formula has a clear correspondence with the behavior of the program. We have two disjuncts, one for each branch of the conditional

[4] Since it may help the reader to follow the upcoming discussions, we have included in the appendix the definition of $\mathcal{A}$ although we know that some operators and notation remain undefined. Since the scope of this paper is not to convince the reader about the correctness of that semantics, we beg him to accept it as it is.

in the body of the declaration. Moreover, since the conditional agent is in the scope of a local variable x, both disjuncts are enclosed within an existential quantification. The computation of the csLTL-semantics is compositional, based on the structure of the program. The first disjunct corresponds to the case when the guard of the conditional agent is satisfied ($y = 1$), thus in the following time instant two things happen: $x = 5$ is added to the store, thus $x = 5$ is entailed, and also is entailed the interpretation for the process call p because a recursive call is run. This illustrates how the intended specification is used as the interpretation of process calls.

Now, following Theorem 1, to check whether the process $p(y)$ satisfies the property, it is sufficient to show that the csLTL formula $\mathcal{D}[\![D]\!]_{\mathcal{S}}(p(y)) \dot{\rightarrow} \mathcal{S}(p(y))$ is valid.

5 An automatic decision procedure for csLTL

In order to make our abstract diagnosis approach effective, we need to define an automatic decision procedure to check the validity of the csLTL formulas that show up when checking a property. In particular, we need to handle csLTL formulas of the form $\psi \dot{\rightarrow} \phi$, where ψ corresponds to the computed approximated behavior of the program, and ϕ is the abstract intended behavior of the process.

Formally, ϕ and ψ are defined by the following grammars.

$$\phi := \dot{true} \mid \dot{false} \mid c \mid \dot{\neg}\, \phi \mid \phi \mathbin{\dot{\wedge}} \phi \mid \bigcirc \phi \mid \phi\, \mathcal{U}\, \phi$$

$$\psi := c \mid \dot{\neg}\, c \mid \bigcirc \psi \mid \psi_1 \mathbin{\dot{\wedge}} \psi_2 \mid \psi_1 \mathbin{\dot{\vee}} \psi_2 \mid \dot{\exists}_x\, \psi \mid \bigcirc \phi$$

Due to the definition of the abstract semantics (see in particular the definition of $\mathcal{A}$ in appendix), ψ cannot be an arbitrary csLTL formula. We know that the until operator can occur only in the scope of a next operator (it can show up only thanks to the intended specification in a process call, whose execution has a delay of one time instant). It also happens that negation can be applied to arbitrary formulas if they are within a next operator, otherwise it can be applied only to constraints.

We impose a restriction on the specification ϕ: we do not allow the use of existential quantifications. Actually, this restriction is quite natural in our context since, in general, we are interested in proving properties related to the *visible* behavior of the program, not to the local variables. In contrast, negation can be applied to any formula ϕ (not only to constraints).

In this section, we extend the tableau construction for Propositional LTL (PLTL) of [10,12] in order to deal with our formulas. We need to adapt the method to our context due to three issues:

1. The structures on which the logic is interpreted are different. In our case, traces (sequences of states) are monotonic, meaning that the information in each state always increases.
2. The logic itself is a bit different from PLTL since propositions are replaced by constraints in **C**.

3. We have to handle existential quantification over variables of the underlying constraint system. This does not mean that we are dealing with a first-order logic as will become clear later

In the following, we first present the basic rules that are used during the construction of the tree associated to the tableau. Then we present the algorithm that implements the process of construction of the tree.

5.1 Basic rules for a csLTL tableau

Classic tableux algorithms are based on the systematic construction of a graph which is used to check the satisfiability of the formula. In [10,12], the authors present an algorithm that does not need to use auxiliary structures such as graphs to decide about the satisfaction of the formula, and this makes this approach more suitable for automatization.

A tableu procedure is defined by means of rules that build a tree whose nodes are labeled with sets of formulas. If all branches of the tree are *closed*, then the formula has no models. Otherwise, we can obtain a model that satisfies the formula from the open branches. Let us introduce the basic rules for the csLTL case. As usual, we present just the minimal set of rules.

Given a set of formulas $\varGamma$. Conjunctions are α-formulas and disjunctions β-formulas. Fig. 1 presents the rules for $\alpha-$ and $\beta-$formulas.

	α	α_1
R1	$\dot{\neg}\,\dot{\neg}\,\phi$	ϕ
R2	$\phi_1 \mathbin{\dot{\wedge}} \phi_2$	ϕ_1, ϕ_2

	β	β_1	β_2
R3	$\dot{\neg}(\phi_1 \mathbin{\dot{\wedge}} \phi_2)$	$\dot{\neg}\,\phi_1$	$\dot{\neg}\,\phi_2$
R4	$\dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)$	$\dot{\neg}\,\phi_1, \dot{\neg}\,\phi_2$	$\phi_1, \dot{\neg}\,\phi_2, \dot{\neg}\bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)$
R5	$\phi_1 \,\mathcal{U}\, \phi_2$	ϕ_2	$\phi_1, \dot{\neg}\,\phi_2, \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)$
R6	$\varGamma, \phi_1 \,\mathcal{U}\, \phi_2$	$\varGamma, \phi_2$	$\varGamma, \phi_1, \dot{\neg}\,\phi_2, \bigcirc((\varGamma^* \mathbin{\dot{\wedge}} \phi_1) \,\mathcal{U}\, \phi_2)$

Fig. 1. α- and β-formulas rules

Tables are interpreted as follows. Each row in a table represents a rule. Each time that an $\alpha-$rule is applied to a node of the tree, a formula of the node matching the pattern in column α is replaced in a child node by the corresponding α_1. For the β-rules, two children nodes are generated, one for each column β_1 and β_2.

Almost all the rules are standard. However, Rule R6 uses the so-called context $\varGamma^*$, which is defined as $\varGamma^* := \dot{\bigvee}_{\gamma \in \varGamma} \dot{\neg}\,\gamma$. The use of contexts is the mechanism to detect the loops that allows one to close branches with eventually formulas. This kind of rules were first introduced in [11]. The idea is that, by using contexts, loops where no formula changes are *discarded* to close a branch.

Note that there is no rule defined for the $\bigcirc$ operator. In fact, the $\mathsf{next}(\varPhi)$ function transforms a set of formulas $\varPhi$ into another: $\mathsf{next}(\varPhi) := \{\phi \,|\, \bigcirc\phi \in \varPhi\} \cup \{\dot{\neg}\,\phi \,|\, \dot{\neg}\bigcirc\phi \in \varPhi\} \cup \{c \,|\, c \in \varPhi, c \in \mathbf{C}\}$. This operator is different from the corresponding one of PLTL in that, in addition to keeping the internal formula of the next formulas, it also *passes* the constraints that are entailed at the current time instant to the following one. This makes sense for *tccp* computations since, as already mentioned, the store in a computation is monotonic, thus no information can be removed and it happens that, always, c implies $\bigcirc c$.

A second main difference w.r.t. the PLTL case regards the existential quantification. To handle it, we define the $\mathsf{hide}(X, \phi)$ operator that, given $X \subseteq Var$ and $\phi \in$ csLTL, removes from ϕ all the information regarding the variables in X. For instance, $\mathsf{hide}(\{x\}, \bigcirc(x = 4 \mathbin{\dot\wedge} y < x)) = \bigcirc(y < 4))$. In the following, we abuse of notation by writing $\mathsf{hide}(x, \phi)$ when X is the singleton $\{x\}$.

5.2 Construction of the csLTL tableau

In this section, we present the algorithm than builds the tableau for our formulas following the ideas of [10,12].

Definition 3 (csLTL tableau). *A csLTL tableau T is a tree where each node n is labeled with a set of csLTL formulas $F(n)$. The root is labeled with the singleton set $\{\phi\}$ for the formula ϕ whose satisfiability/validity is needed to check, and the children of a node are obtained by applying the basic rules of Subsection 5.1.*

Definition 4. *A node in the tableau is* inconsistent *if it contains*

- *a couple of formulas $\phi, \dot\neg\,\phi$, or*
- *the formula $\dot{false}$, or*
- *a couple of constraint formulas $c, \dot\neg\, c'$ such that $c \vdash c'$.*

The last condition for inconsistence of a node is particular to the *ccp* context. Since we are dealing with constraints that model partial information, it is possible that we have an *implicit* inconsistence, in the sense that we need the entailment relation to detect it.

The algorithm marks nodes when they cannot be further processed. In particular, a node is marked when it is inconsistent (closed) or when it contains just constraint formulas (open). Nodes with no children are *leaf* nodes.

Let us first introduce the notion of *fulfilled* eventuality formula in a path of the tableau. This notion characterizes the satisfaction of the eventuality formulas and allows us to close nodes when they contain such kind of formulas.

Definition 5. *Let T be a tableau and $p = n_1, n_2, \ldots, n_j$ a path in T. Any eventuality $\phi_1 \,\mathcal{U}\, \phi_2 \in F(n_i)$, with $1 \le i \le j$, is fulfilled in p if there exists a k such that $i \le k \le j$ and $\phi_2 \in F(n_k)$.*

Intuitively, the formula is fulfilled in the path if we can reach (following the path) a node where ϕ_2 is true.

When dealing with eventualities, to determine which rule R5 or R6 has to be applied in a node, it is necessary to *distinguish* eventualities. The idea is to mark which is the eventuality that is being unfolded in the path. Then, the rule R6 is applied only to *distinguished* eventualities; in any other case, the rule R5 is used. If a node does not contain any distinguished eventuality, then the algorithm distinguishes one of them and rule R6 is chosen to be applied to it. Each node of the tableau has at most one distinguished eventuality.

Now we are ready to show how the tableau is built. The algorithm repeatedly selects an unmarked leaf node l labelled with the set of formulas $F(l)$ and applies, in order, one of the points shown below.

1. If l is an inconsistent node, then mark it as closed ($\times$).
2. If $F(l)$ is a set of constraint formulas, mark l as open ($\odot$).
3. If $F(l) = F(l')$ for l' ancestor of l, take the oldest ancestor l'' of l that is labeled with $F(l)$ and check if each eventuality in the path between l'' and l is fulfilled in such path. If they are all fulfilled, then mark l as open ($\odot$).
4. If none of the points above applied, choose $\phi \in F(l)$ such that ϕ is not a *next* formula. Then,
   - if ϕ is an α-formula (let $\phi = \alpha$), create a new node l' as a child of l and label it as $F(l') = (F(l) \smallsetminus \{\alpha\}) \cup \alpha_1$ by using the corresponding rule in Fig. 1,
   - if ϕ is a β-formula (let $\phi = \beta$), create two new nodes l' and l'' as children of l and label them as $F(l') = (F(l) \smallsetminus \{\beta\}) \cup \beta_1$ and $F(l'') = (F(l) \smallsetminus \{\beta\}) \cup \beta_2$ by using the corresponding rule in Fig. 1. Moreover, if β is an eventuality, we have three possible cases:
     - if β is the distinguished eventuality in $F(l)$, then apply Rule R6 to β and distinguish the formula inside the *next* formula in β_2;
     - if β is not distinguished, but there is another distinguished formula, then apply Rule R5 to β and maintain the existing distinguished formula in β_1 and β_2;
     - otherwise, distinguish β and apply Rule R6 to β and distinguish the formula inside the *next* formula in β_2;
   - if ϕ is an $\exists$-formula ($\phi = \dot{\exists}_x\, \phi'$), then check if the formula $\mathsf{hide}(\mathit{Var} \smallsetminus \{x\}, \phi')$ is satisfiable (by constructing another csLTL tableau). If it is satisfiable, then create a new node l' as a child of l and label it as $F(l') = (F(l) \smallsetminus \{\phi\}) \cup \{\mathsf{hide}(x, \phi')\}$, otherwise, mark l as closed ($\times$).
5. If $F(l)$ is formed only by constraint formulas and *next* formulas, apply the next operator: create a new node l' as child of l and label it as $F(l') = \mathsf{next}(F(l))$.

Each branch of the tree can be seen as divided into stages, where each stage is a set of consecutive nodes which are obtained by applying $\alpha-$ or $\beta-$rules. When the next rule is applied, we move from one stage to the following one in the branch. Moreover, when the hiding rule is applied, an auxiliary tableau is built for a given (smaller) formula of the node. The case for the hiding rule is particular to the *ccp* context. Since models of formulas are traces that belong to the semantics of a *tccp* program, the idea is to try to determine whether there exists a model for the local information x. In that case, we know that we can find a trace that coincides with the original one but for the information x, which means that the whole formula is true.

This strategy for the existential quantification works under the assumption that, if we have a conditional agent nested within a hiding agent (in a single time instant), then the variable of the existential quantification does not appear in the guard of the conditional agent. Actually, it holds that for that kind of programs there exists an equivalent one in which this situation does not happen, thus this is not a real limitation.

The construction terminates when every leaf is marked. A tableau whose construction has terminated is called *complete*. A complete tableu with all the leaves marked × is said to be *closed*, otherwise it is *open*.

A strategy for the construction of the tableau is called *fair* if an open branch does not contain an unfulfilled eventuality that has never been distinguished. A fair strategy makes that the construction always terminates since the constraint formulas, the possible context Γ, and (by induction) the possible auxiliary tableaux for the elimination of the existential quantification which can occur in the construction, are finite. This result is similar to the corresponding result for the PLTL tableau.

We aim to use the classical results of satisfiability (to be proven in our case) to apply the algorithm to the final step of our abstract diagnosis technique. In particular, we know that

- if there exists a closed tableau for $\phi \in \mathsf{csLTL}$ then ϕ is unsatisfiable, and
- if there exists an open tableau for $\phi \in \mathsf{csLTL}$ then ϕ is satisfiable.

Since we are interested in checking the validity of a formula of the form $\phi \dot{\rightarrow} \psi$, we build the tableau for its negation and check whether the resulting tableau is closed. That means that the negation is unsatisfiable, thus our implication is valid. In other words, we build the tableau for the initial formula $\phi \dot{\wedge} \dot{\neg} \psi$. Then, we check if the resulting tableau is closed.

Example 4. Consider the formula of our guiding example (introduced in Example 3) $\dot{\exists}_x \phi \dot{\rightarrow} \Diamond(y = 1)$, where $\phi = (y = 1 \dot{\wedge} \bigcirc x = 5 \dot{\wedge} \bigcirc(\Diamond y = 1)) \dot{\vee} (\dot{\neg} y = 1 \dot{\wedge} \bigcirc y = 1)$. The tableau in Fig. 2, with $F(\mathit{root}) = \dot{\exists}_x \phi \dot{\wedge} \dot{\neg} \Diamond(y = 1)$ shows its validity. Arrows labeled with α and β correspond to the application of α and β rules, respectively; arrows labeled with X represent the application of the next operator. Finally, arrows labeled with $\exists$ correspond to the elimination of the existential quantification for the formula $\dot{\exists}_x \phi$.

In the example, the first step uses the rule for the conjunction. Then, the second step involves the construction of an auxiliary tableau (shown in Fig. 3) for checking the satisfiability of $\mathsf{hide}(\mathit{Var} \setminus \{x\}, \phi) = \bigcirc x = 5$. Since the auxiliary tableau is open, the formula representing the local information is satisfiable, which means that there exists a model for it, thus it exists a trace (that we do not compute) that makes the whole formula true.

The third node of Fig. 2 corresponds to the descendent of the existential quantification, being $(y = 1 \dot{\wedge} \bigcirc(\Diamond y = 1)) \dot{\vee} (\dot{\neg} y = 1 \dot{\wedge} \bigcirc y = 1)$ the formula resulting by removing the information about the local variable. This formula is then selected for a β step (disjunction). The branch on the left is closed after two steps since $y = 1$ and $\dot{\neg} y = 1$ both belong to the node labeling.

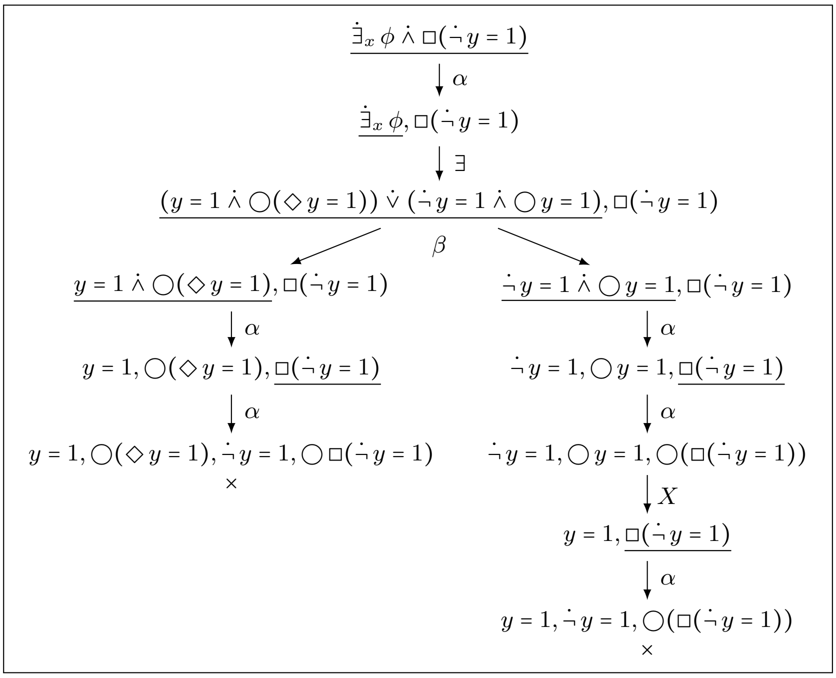
∃̇x φ ∧̇ □(¬̇ y = 1)
α
∃̇x φ, □(¬̇ y = 1)
∃
(y = 1 ∧̇ ◯(◇ y = 1)) ∨̇ (¬̇ y = 1 ∧̇ ◯ y = 1), □(¬̇ y = 1)
β
y = 1 ∧̇ ◯(◇ y = 1), □(¬̇ y = 1)
¬̇ y = 1 ∧̇ ◯ y = 1, □(¬̇ y = 1)
α
α
y = 1, ◯(◇ y = 1), □(¬̇ y = 1)
¬̇ y = 1, ◯ y = 1, □(¬̇ y = 1)
α
α
y = 1, ◯(◇ y = 1), ¬̇ y = 1, ◯ □(¬̇ y = 1)
×
¬̇ y = 1, ◯ y = 1, ◯(□(¬̇ y = 1))
X
y = 1, □(¬̇ y = 1)
α
y = 1, ¬̇ y = 1, ◯(□(¬̇ y = 1))
×

Fig. 2. Tableau for $\dot{\exists}_x\, \phi \mathbin{\dot{\rightarrow}} \Diamond\, y = 1$ of Example 4.

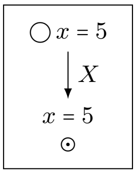

Fig. 3. Tableau for $\bigcirc\, x = 5$ of Example 4.

The branch on the right first flattens the conjunction and then applies the next rule. Note that the negation of a constraint is not kept in the following time instant. We recall that negation means "not entailment" (in contrast to meaning that *the contrary* is true), thus, in the future the constraint could be entailed.

Since also this second branch is closed, we know that the formula is not satisfiable, thus our implication is valid.

In the context of abstract diagnosis, this proves that the program is abstractly correct w.r.t. the LTL specification.

Example 5. Now, suppose that we want to check, for the program introduced in Example 1, that the constraint $y = 1$ is always entailed by the store. The corresponding specification is $\mathcal{S}'(p(y)) = \Box(y = 1)$.

The csLTL-semantics $\mathcal{D}$ for $p(y)$ using the given specification as interpretation is given by the formula $\dot{\exists}_x \big((y = 1 \mathbin{\dot{\wedge}} \bigcirc x = 5 \mathbin{\dot{\wedge}} \bigcirc(\Box\, y = 1)) \mathbin{\dot{\vee}} (\dot{\neg}\, y = 1 \mathbin{\dot{\wedge}} \bigcirc y = 1)\big)$.

Let us abbreviate the body of the existential quantification as ϕ'. To check whether the process $p(y)$ is correct w.r.t. the property, we have to show that $\dot{\exists}_x \, \phi' \mathrel{\dot{\rightarrow}} \Box(y = 1)$ is valid.

The tableau in Fig. 4 shows the satisfiability of the formula $\dot{\exists}_x \, \phi' \mathrel{\dot{\wedge}} \Diamond(\dot{\neg}\, y = 1)$. This means that its negation, $\dot{\exists}_x \, \phi' \mathrel{\dot{\rightarrow}} \Box(y = 1)$, is not valid. In the context of abstract diagnosis, although the formula is not satisfied by the program, because of the loss of precision due to the approximation, this is only a warning about the possible incorrectness of the program w.r.t. the LTL specification.

Notice that the second step involves the construction of the same auxiliary tableau of Example 4 (shown in Fig. 3) for checking the satisfiability of $\mathsf{hide}(\mathit{Var} \setminus \{x\}, \phi') = \bigcirc x = 5$. Furthermore, Rule R6 is applied twice to deal with the distinguished eventuality $\Diamond(\dot{\neg}\, y = 1)$.

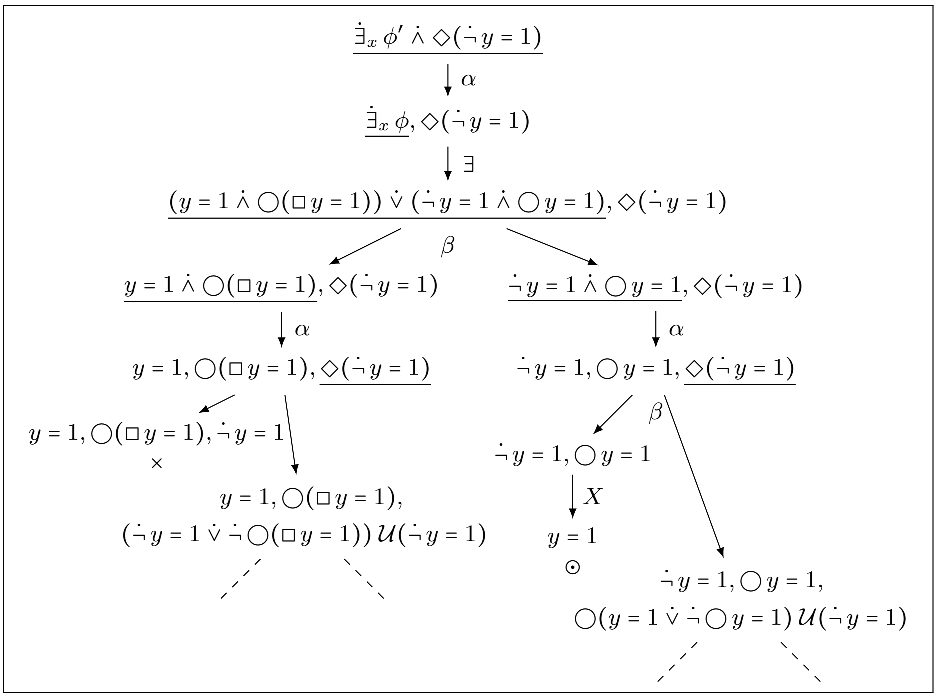
$\underline{\dot{\exists}_x \, \phi' \mathrel{\dot{\wedge}} \Diamond(\dot{\neg}\, y = 1)}$
α
$\underline{\dot{\exists}_x \, \phi}, \Diamond(\dot{\neg}\, y = 1)$
$\exists$
$\underline{(y = 1 \mathrel{\dot{\wedge}} \bigcirc(\Box\, y = 1)) \mathrel{\dot{\vee}} (\dot{\neg}\, y = 1 \mathrel{\dot{\wedge}} \bigcirc\, y = 1)}, \Diamond(\dot{\neg}\, y = 1)$
β
$\underline{y = 1 \mathrel{\dot{\wedge}} \bigcirc(\Box\, y = 1)}, \Diamond(\dot{\neg}\, y = 1)$
$\underline{\dot{\neg}\, y = 1 \mathrel{\dot{\wedge}} \bigcirc\, y = 1}, \Diamond(\dot{\neg}\, y = 1)$
α
α
$y = 1, \bigcirc(\Box\, y = 1), \underline{\Diamond(\dot{\neg}\, y = 1)}$
$\dot{\neg}\, y = 1, \bigcirc\, y = 1, \underline{\Diamond(\dot{\neg}\, y = 1)}$
$y = 1, \bigcirc(\Box\, y = 1), \dot{\neg}\, y = 1$
$\times$
β
$\dot{\neg}\, y = 1, \bigcirc\, y = 1$
X
$y = 1$
$\odot$
$y = 1, \bigcirc(\Box\, y = 1),$
$(\dot{\neg}\, y = 1 \mathrel{\dot{\vee}} \dot{\neg} \bigcirc(\Box\, y = 1))\, \mathcal{U}(\dot{\neg}\, y = 1)$
$\dot{\neg}\, y = 1, \bigcirc\, y = 1,$
$\bigcirc(y = 1 \mathrel{\dot{\vee}} \dot{\neg} \bigcirc\, y = 1)\, \mathcal{U}(\dot{\neg}\, y = 1)$

Fig. 4. Tableau for $\dot{\exists}_x \, \phi' \mathrel{\dot{\rightarrow}} \Box\, y = 1$ of Example 5

6 Conclusions and Future Work

In this paper, we have introduced a decision procedure for a class of csLTL formulas. csLTL is a linear temporal logic that replaces propositional formulas by constraint formulas, thus in order to determine the validity of a formula with no temporal constructs, it uses the entailment relation of the underlying constraint system.

The decision procedure is an adaptation of the tableau defined in [10,12]. The main differences of our algorithm w.r.t. the propositional case are due to the constraint nature of the behavior of *tccp*.

This decision procedure is the last step of a method to validate LTL formulas for *tccp* programs. We still need to formally prove the correctness result of our algorithm. This is ongoing work. Once we have these results, we would have a completely automatic abstract diagnosis instance for a subset of LTL formulas.

As future work, we plan to implement this algorithm and integrate it with the verification method. We also plan to explore other instances of the method based on logics for which decision procedures or (semi)automatic tools exists. If we were able to find a decision procedure for a more expressive fragment of the logic, then we could define a more precise abstract semantics and accurate specifications.

References

1. M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag.
2. G. Bacci and M. Comini. Abstract Diagnosis of First Order Functional Logic Programs. In M. Alpuente, editor, *Logic-based Program Synthesis and Transformation, 20th International Symposium*, volume 6564 of *Lecture Notes in Computer Science*, pages 215–233, Berlin, 2011. Springer-Verlag.
3. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
4. M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.
5. M. Comini, L. Titolo, and A. Villanueva. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 11(4-5):487–502, 2011.
6. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.
7. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for Reasoning about Timed Concurrent Constraint Programs. In *TIME '01: Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning (TIME'01)*, page 227, Washington, DC, USA, 2001. IEEE Computer Society.
8. F. S. de Boer, M. Gabbrielli, and M. C. Meo. Proving correctness of Timed Concurrent Constraint Programs. *CoRR*, cs.LO/0208042, 2002.
9. M. Falaschi, C. Olarte, C. Palamidessi, and F. D. Valencia. Declarative Diagnosis of Temporal Concurrent Constraint Programs. In V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 271–285. Springer-Verlag, 2007.

10. J. Gaintzarain, M. Hermo, P. Lucio, and M. Navarro. Systematic semantic tableaux for pltl. *Electronic Notes in Theoretical Computer Science*, 206:59–73, 2008.
11. J. Gaintzarain, M. Hermo, P. Lucio, M. Navarro, and F. Orejas. A cut-free and invariant-free sequent calculus for pltl. In J. Duparc and T. A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 481–495. Springer, 2007.
12. J. Gaintzarain, M. Hermo, P. Lucio, M. Navarro, and F. Orejas. Dual systems of tableaux and sequents for pltl. *The Journal of Logic and Algebraic Programming*, 78(8):701–722, 2009.
13. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.
14. C. Palamidessi and F. D. Valencia. A Temporal Concurrent Constraint Programming Calculus. In *7th International Conference on Principles and Practice of Constraint Programming (CP'01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2001.
15. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
16. V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, Mass., 1993.
17. F. D. Valencia. Decidability of infinite-state timed ccp processes and first-order ltl. *Theoretical Computer Science*, 330(3):577–607, 2005.

A Abstract semantics evaluation function for agents

The following function is the core definition for the correct abstract semantics for *tccp* in the domain of csLTL formulas. Actually, this version of the semantics is an instance of the general framework in which we are restricted to a decidable subset of the csLTL logic. For this reason, the semantics for the choice agent is a correct, but not precise, semantics of the agent's behavior.

Definition 6 (csLTL abstract Semantics).
Given $A \in \mathbb{A}^{\Pi}_{\mathbf{C}}$ *and* $\mathcal{I} \in \mathbb{I}_{\mathbb{F}}$*, we define the* csLTL semantics evaluation $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ *by structural induction as follows.*

$$\mathcal{A}[\![\mathsf{skip}]\!]_{\mathcal{I}} \coloneqq \mathit{true} \tag{A.1a}$$

$$\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}} \coloneqq \bigcirc\, c \tag{A.1b}$$

$$\mathcal{A}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}} \coloneqq \dot{\bigvee}_{i=1}^{n} (c_i \mathbin{\dot\wedge} \bigcirc \mathcal{A}[\![A_i]\!]_{\mathcal{I}}) \mathbin{\dot\vee} (\dot{\bigwedge}_{i=1}^{n} \dot\neg\, c_i) \tag{A.1c}$$

$$\mathcal{A}[\![\mathsf{now}\ c\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2]\!]_{\mathcal{I}} \coloneqq (c \mathbin{\dot\wedge} \mathcal{A}[\![A_1]\!]_{\mathcal{I}}) \mathbin{\dot\vee} (\dot\neg\, c \mathbin{\dot\wedge} \mathcal{A}[\![A_2]\!]_{\mathcal{I}}) \tag{A.1d}$$

$$\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{I}} \coloneqq \mathcal{A}[\![A_1]\!]_{\mathcal{I}} \mathbin{\dot\wedge} \mathcal{A}[\![A_2]\!]_{\mathcal{I}} \tag{A.1e}$$

$$\mathcal{A}[\![\exists x\, A]\!]_{\mathcal{I}} \coloneqq \dot\exists_x\, \mathcal{A}[\![A]\!]_{\mathcal{I}} \tag{A.1f}$$

$$\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{I}} \coloneqq \bigcirc \mathcal{I}(p(\vec{x})) \tag{A.1g}$$

Let $D \in \mathbb{D}^{\Pi}_{\mathbf{C}}$*. We define the (monotonic) immediate consequence operator* $\mathcal{D}[\![D]\!] \colon \mathbb{I}_{\mathbb{F}} \rightarrow \mathbb{I}_{\mathbb{F}}$ *as*

$$\mathcal{D}[\![D]\!]_{\mathcal{I}}(p(\vec{x})) \coloneqq \dot{\bigvee} \, \{\mathcal{A}[\![A]\!]_{\mathcal{I}} \mid p(\vec{x}) \mathrel{:\!-} A \in D\}$$