Modeling Hybrid Systems in the Concurrent Constraint Paradigm

Damián Adalid María del Mar Gallardo Laura Titolo
Dept. Lenguajes y Ciencias de la Computación
E.T.S.I. Informática University of Málaga[*]
[damian,gallardo,laura.titolo]@lcc.uma.es

Hybrid systems, which combine discrete and continuous dynamics, require quality modeling languages to be either described or analyzed. The Concurrent Constraint paradigm (*ccp*) is an expressive declarative paradigm, characterized by the use of a common constraint store to communicate and synchronize concurrent agents. In this paradigm, the information is stated in the form of constraints, in contrast to the variable/value style typical of imperative languages. Several extensions of *ccp* have been proposed in order to model reactive systems. One of these extensions is the Timed Concurrent Constraint Language (*tccp*) that adds to *ccp* a notion of discrete time and new features to model time-out and preemption actions.

The goal of this paper is to explore the expressive power of *tccp* to describe hybrid systems. We introduce the language Hy-*tccp* as a conservative extension of *tccp*, by adding a notion of continuous time and new constructs to describe the continuous dynamics of hybrid systems. In this paper, we present the syntax and the operational semantics of Hy-*tccp* together with some examples that show the expressive power of our new language.

1 Introduction

In the last years, concurrent, reactive and hybrid systems have become essential to model a large number of modern applications. Often, systems of this kind are classified as critical, i.e., an error in the software can have tragic consequences in terms of human lives or money. This is the case of avionic or automotive software, e-banking, or financial applications.

Description, verification and analysis of concurrent and reactive systems are very hard tasks, due to the concurrent execution of different agents and to issues of synchronization. In the case of hybrid systems, these phases are even harder due to the combination of discrete and continuous dynamics and the presence of real-valued variables. Therefore, it is important to develop high-level description languages that allow these systems to be modeled with enough precision and at the same time that ease the application of formal methods techniques.

Many formalisms have been developed to describe concurrent systems. One of these is the *Concurrent Constraint paradigm* (*ccp*) [10], a simple but powerful model for concurrent systems. It differs from other paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. In this paradigm, the agents running in parallel communicate by means of a global constraint store. The *Timed Concurrent Constraint Language* [2] (*tccp* in short) is a concurrent logic language obtained by extending *ccp* with the notion of time and a suitable mechanism to model time-outs and preemptions.

[*]This work has been supported by the Andalusian Excellence Project P11-TIC7659 and the Spanish Ministry of Economy and Competitiveness project TIN2012-35669

In this paper, we present the language Hy-*tccp*: an extension of *tccp* over continuous time. Hy-*tccp* is a non-deterministic and synchronous language that incorporates continuous variables that follow dynamics determined by an ordinary differential equation (ODE). Its declarative nature facilitates a high level description of hybrid systems in the style of hybrid automata [8]. Furthermore, its logical nature facilitates the development of semantics based program manipulation tools for hybrid systems (verifiers, analyzers, debuggers…). Parallel composition of hybrid automata is naturally supported in Hy-*tccp* due to the existence of a global shared store and to the synchronization mechanism inherited from *tccp*. By defining Hy-*tccp*, we show that the extension of a declarative constraint language with continuous dynamics is not only possible, but it leads to a powerful and expressive language able to describe complex hybrid systems.

In this paper, we have only considered the modeling of multi-rated [5] hybrid systems, i.e., systems whose continuous variables follow a constant dynamics. However, in the future we aim to relax this restriction in order to describe more complex dynamics such as those defined by rectangular sets.

The paper is organized as follows. In Section 2, we briefly introduce the language *tccp* and the essential aspects of hybrid automata. In Section 3, we introduce the new language Hy-*tccp* together with its operational semantics, and we describe the new features that have been added to *tccp* in order to model hybrid systems. Section 4 contains some examples to highlight the expressive power of Hy-*tccp*. Section 5 presents some related work and, finally, Section 6 concludes the paper and outlines future work.

2 Background

In this section we present some background to clarify the contributions of the paper. In Subsection 2.1, we introduce the language *tccp*, the starting point for the definition of Hy-*tccp*. In Subsection 2.2, we introduce the basic notions of hybrid automata, which is the formalism commonly used to describe hybrid systems.

2.1 The Timed Concurrent Constraint Language

The *Timed Concurrent Constraint Language* (*tccp*, [2]) is a time extension of *ccp*. It adds to *ccp* the notion of time and the ability to capture the absence of information. With these features, one can specify behaviors typical of concurrent and reactive systems.

The computation in *tccp* proceeds as the concurrent execution of several agents that can monotonically add constraints in a global *store* or query information from it. As are all the languages from the *cc* paradigm, *tccp* is parametric w.r.t. a *cylindric constraint system*.

Definition 2.1 (Cylindric constraint system [2]) *A cylindric constraint system is an algebraic structure of the form:*

$$\mathbf{C} = \langle \mathcal{C}, \leq, \wedge, true, false, Var, \exists \rangle$$

such that:

1. $\langle \mathcal{C}, \leq, \wedge, true, false \rangle$ *is a complete lattice where* $\wedge$ *is the least upper bound (lub) operator, and true and false are, respectively, the least and the greatest elements of* $\mathcal{C}$. *We often use the inverse order* $\vdash$ *(the* entailment *relation) instead of* $\leq$ *over constraints. Formally* $\forall c, d \in \mathcal{C}\ c \leq d \Leftrightarrow d \vdash c$.
2. *Var is a denumerable set of variables.*

3. *For each element $x \in Var$, a function (also called cylindric operator) $\exists_x{:}\mathcal{C} \to \mathcal{C}$ is defined such that, for any $c,d \in \mathcal{C}$ the following axioms hold:*
   (a) $c \vdash \exists_x c$
   (b) *if* $c \vdash d$ *then* $\exists_x c \vdash \exists_x d$
   (c) $\exists_x(c \wedge \exists_x d) = \exists_x c \wedge \exists_x d$
   (d) $\exists_x(\exists_y c) = \exists_y(\exists_x c)$

The entailment relation $\vdash$ intuitively states that if c contains more information than d then $c \vdash d$. The *lub* operator $\wedge$ merges the information from two constraints (e.g. $x > 0 \wedge x > 5 \wedge y = 9 := x > 5 \wedge y = 9$ and $x = 0 \wedge x = 7 := false$). The *cylindrification* (or *hiding*) operator is defined in terms of a general notion of existential quantifier. It is used to project away information about the considered variable in order to make it local to the constraint and hide it from the context (e.g. $\exists_x(x = 0 \wedge y = x \wedge z > 7) := y = 0 \wedge z > 7$).

The *tccp* global store is *monotonic* in the sense that once a constraint is added to the store, it cannot be removed. Thus, given the store $x > 0 \wedge y > 2$ we can add the information $x > 5$ and obtain the store $x > 5 \wedge y > 2$. Furthermore, by adding $x = 0$ we obtain the inconsistent store *false* since the constraint $x = 0$ is in contradiction with the information already present in the store.

The syntax of *tccp* agents is given by the grammar:

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

where $c, c_1, \ldots, c_n$ are finite constraints in $\mathcal{C}$, p is a process symbol, and $\vec{x} \in Var \times \cdots \times Var$. A *tccp* program is a pair $D.A$, where A is the initial agent and D is a set of *process declarations* of the form $p(\vec{x}) :- A$.

The *operational semantics* of *tccp* [2] is described by a transition system $T = (Conf, \to)$. Configurations in *Conf* are pairs $\langle A, c\rangle$ representing the agent A to be executed in the current global store c. The transition relation $\to\, \subseteq Conf \times Conf$ is the least relation satisfying the rules in Figure 1. Each transition step takes exactly one time-unit. The notion of time is introduced by defining a global clock which synchronizes all agents.

As can be seen from the rules, the stop agent represents the successful termination of the computation. The $\mathsf{tell}(c)$ agent adds the constraint c to the current store by means of the $\wedge$ operator and then stops. It takes one time-unit, thus the constraint c is visible to the other agents from the following time instant. 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 is entailed by the current store; otherwise, if no guard is entailed by the store, the agent suspends. The conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves (in the current time instant) like A (respectively B) if c is (respectively is not) entailed by the store. This conditional agent is able to process *negative information* (lack of some information): it can capture when some information is not present in the store since the agent B is executed both when $\neg c$ is satisfied, but also when neither c nor $\neg c$ are satisfied. $A \parallel B$ models the parallel composition of A and B in terms of maximal parallelism, i.e., all the enabled agents of A and B are executed at the same time. The agent $\exists x A$ makes variable x local to A, to this end, it uses the $\exists$ operator of the constraint system. More specifically, it behaves like A with x considered local, i.e., the information on x provided by the external environment is hidden to A, and the information on x produced by A is hidden to the external world. In the corresponding rule, the store l in the agent $\exists^l x A$ represents the store local to A. This auxiliary operator is linked to the hiding construct by setting the initial local store to *true*, thus $\exists x A := \exists^{true} x A$. Finally, the agent $p(\vec{x})$ takes from D a declaration of the form $p(\vec{x}) :- A$ and then executes A.

$$\frac{}{\langle \mathsf{tell}(c), d\rangle \rightarrow \langle \mathsf{stop}, c\wedge d\rangle} \qquad \frac{\exists 1\le j\le n. d\vdash c_j}{\langle \sum_{i=1}^{n} \mathsf{ask}(c_i)\rightarrow A_i, d\rangle \rightarrow \langle A_j, d\rangle}$$

$$\frac{\langle A,d\rangle \rightarrow \langle A',d'\rangle, d\vdash c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d\rangle \rightarrow \langle A', d'\rangle} \qquad \frac{\langle A,d\rangle \not\rightarrow, d\vdash c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d\rangle \rightarrow \langle A, d\rangle}$$

$$\frac{\langle B,d\rangle \rightarrow \langle B',d'\rangle, d\not\vdash c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d\rangle \rightarrow \langle B', d'\rangle} \qquad \frac{\langle B,d\rangle \not\rightarrow, d\not\vdash c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d\rangle \rightarrow \langle B, d\rangle}$$

$$\frac{\langle A,d\rangle \rightarrow \langle A',d'\rangle \quad \langle B,d\rangle \rightarrow \langle B',c'\rangle}{\langle A\parallel B, d\rangle \rightarrow \langle A'\parallel B', d'\wedge c'\rangle} \qquad \frac{\langle A,d\rangle \rightarrow \langle A',d'\rangle \quad \langle B,d\rangle \not\rightarrow}{\begin{array}{c}\langle A\parallel B, d\rangle \rightarrow \langle A'\parallel B, d'\rangle \\ \langle B\parallel A, d\rangle \rightarrow \langle B\parallel A', d'\rangle\end{array}}$$

$$\frac{\langle A, l\wedge \exists_x d\rangle \rightarrow \langle B, l'\rangle}{\langle \exists^{l} xA, d\rangle \rightarrow \langle \exists^{l'} xB, d\wedge \exists_x l'\rangle} \qquad \frac{p(\vec{x}) :- A \in D}{\langle p(\vec{x}), d\rangle \rightarrow \langle A, d\rangle}$$

Figure 1: The transition system for *tccp*.

2.2 Introduction to hybrid automata

Many real systems have complex behaviors and evolve following both discrete and continuous dynamics. These systems are called hybrid systems. For instance, a cooler system is a hybrid system: it has two discrete states (*on* or *off*) that are chosen according to the temperature of the room, which evolves continuously over time.

Hybrid automata [8] are an extension of finite-state automata used to describe hybrid systems. Intuitively, the discrete behavior of a hybrid automaton is defined by means of a finite set of discrete states (called *locations*) and a set of (instantaneous) *discrete transitions* from one location to another. The continuous behavior of hybrid automata is described at each location by means of some Ordinary Differential Equations (ODEs) which describe how continuous variables evolve over time (*continuous transitions*).

Definition 2.2 (Hybrid automaton) *A* hybrid automaton *H is a tuple*

$$\langle Loc, T, \Sigma, X, Init, Inv, Flow, Jump\rangle$$

where:

- *Loc is a finite set $\{loc_1, \dots, loc_n\}$ of discrete states (locations).*
- *$T \subseteq Loc \times Loc$ is a finite set of discrete transitions.*
- *Σ is a set of event names, associated with a labelling function $\Lambda : T \rightarrow \Sigma$.*
- *$X = \{x_1, \dots, x_m\}$ is a finite set of real-valued variables. The set $\dot{X} = \{\dot{x_1}, \dots, \dot{x_m}\}$ represents the first derivatives of the elements in X. In addition, the set $X' = \{x'_1, \dots, x'_m\}$ represents the updates of the variables when a discrete transition takes place. In this section, we assume that discrete variables are continuous variables whose derivative is zero at all locations.*
- *The functions $Init$, Inv and $Flow$ assign predicates to each location $loc \in Loc$. $Init(loc)$ establishes the possible initial values for the continuous variables at location loc. $Inv(loc)$ constrains the values of the continuous variables at location loc. $Flow(loc)$ contains the differential equations describing the evolution of the continuous variables at location loc.*

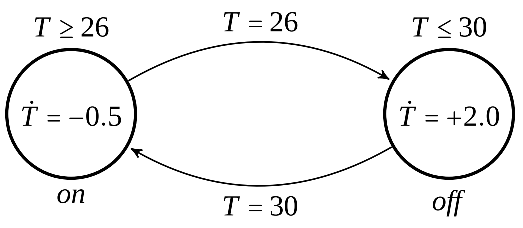

Figure 2: Hybrid automaton for the cooler system

- *Function Jump assigns to each discrete transition $t \in T$ a* guard *that must be satisfied in order to allow the transition to take place, and a* reset *predicate which updates the value and/or the flow of a continuous variables.*

EXAMPLE 2.3 Figure 2 shows a hybrid automaton modeling a cooler system. The automaton has two locations *on* and *off* and a continuous variable T storing the room temperature. When the automaton is at location *on* (the cooler is turned on) the temperature decreases at rate -0.5. When the location is *off* (the cooler is turned off) the temperature increases at rate $+2.0$. Transitions between locations represent the turning on or off of the cooler. These transitions are guarded with conditions. For instance, transition *on-off* takes place when the temperature is 26, while transition *off-on* takes place when the temperature is 30.

■

A hybrid automaton behaves like a *timed transition system* (TTS), where each step is labelled either with a positive real value τ (continuous transition of duration τ) or with σ (discrete transition). Let $[X \to \mathbb{R}]$ be the set of maps from X to $\mathbb{R}$. An automaton state, called hybrid state from now on, is a pair $(loc, v) \in (Loc \times [X \to \mathbb{R}])$, where $loc \in Loc$ is a location of the automaton, and $v \in [X \to \mathbb{R}]$ maps each continuous variable to its current value.

Let p be a predicate over $X \cup \dot{X}$ or $X \cup X'$, then $[\![p]\!]$ denotes all functions $v \in [X \to \mathbb{R}]$ that satisfy p.

DEFINITION 2.4 (TRAJECTORIES) *Let $H = \langle Loc, T, \Sigma, X, Init, Inv, Flow, Jump\rangle$ be a hybrid automaton. We consider two types of transitions:*

Discrete transitions *Let $(loc, loc') \in T$, $(loc, v) \to_{\sigma} (loc', v')$, iff $v, v' \in [X \to \mathbb{R}]$, and $(v, v') \in [\![Jump(t)]\!]$.*

Continuous transitions *For each $\tau \in \mathbb{R}^+$, we have $(loc, v) \to_{\tau} (loc, v')$ iff there exists a differentiable function $f : [0, \tau] \to \mathbb{R}^m$, $\dot{f} : [0, \tau] \to \mathbb{R}^m$ being its first derivative, such that:*

- $f(0) = v$
- $f(\tau) = v'$
- $\forall \tau' \in [0, \tau], f(\tau') \in [\![Inv(loc)]\!]$
- $(f(\tau'), \dot{f}(\tau')) \in [\![Flow(loc)]\!]$

A trajectory *is a (possible infinite) sequence of hybrid states such as $(loc_0, v_0) \to_{\lambda_1} (loc_1, v_1) \to_{\lambda_2} \ldots \to_{\lambda_n} (loc_n, v_n) \to_{\lambda_n} \ldots$, where for all $i \geq 0$, $v_i \in [\![Inv(loc_i)]\!]$ and $\lambda_i \in \mathbb{R} \cup \{\sigma\}$.*

It is worth noting that the system is free to select non-deterministically at each moment any enabled transition, either discrete or continuous.

EXAMPLE 2.5 Considering the hybrid system in Figure 2, the following trajectory represents a possible evolution of the automaton starting at hybrid state $(on, 27)$: $(on, 27) \to_{1} (on, 26.5) \to_{1} (on, 26) \to_{\sigma} (off, 26) \to_{0.5} (off, 27) \to_{1.5} (off, 30) \to_{\sigma} (on, 30) \ldots$ ■

3 Hy-*tccp*: an extension of *tccp* over continuous time

In this section, we present the language Hy-*tccp*, which subsumes *tccp* and includes new agents in order to model the continuous behavior typical of hybrid systems in the style of hybrid automata. In contrast to *tccp*, in Hy-*tccp* we consider a notion of *continuous* time by means of a global continuous clock.

Hy-*tccp* uses a *tccp* monotonic store (called *discrete store*) to model the information about the current location and the associated invariants of a hybrid automaton. Discrete transitions are modeled as instantaneous transitions in Hy-*tccp* and they are used to synchronize parallel agents/automata. In summary, the features offered by *tccp* are used to model the discrete behavior of hybrid automata. However, hybrid automata are characterized by the use of continuous variables whose values change following some ODEs. For this reason, the *tccp* store is extended by adding a component called *continuous store*. The continuous store is not monotonic, instead it records the dynamical evolution of the continuous variables.

We distinguish the set of discrete variables *Var*, whose information is accumulated monotonically, and the set of continuous variables $\widetilde{Var}$, whose values change continuously over time ($Var \cap \widetilde{Var} = \varnothing$). Constraints in $\mathcal{C}$ are now defined over $Var \cup \widetilde{Var}$.

A *continuous store* is a function that associates a continuous variable with two real numbers: its value and its flow, which indicates how its value changes over time. In this work, we consider only ODEs of the form $\dot{x} = n$ with $n \in \mathbb{R}$. In the future, we intend to also consider ODEs of the form $\dot{x} \in [n_1, n_2]$ with $n_1, n_2 \in \mathbb{R}$ in order to model rectangular hybrid systems.

We denote as $\tilde{\mathcal{C}} = [\widetilde{Var} \hookrightarrow (\mathbb{R} \times \mathbb{R})]$ the set of all possible continuous stores, and as $\widetilde{true}$ and $\widetilde{false}$ the empty and the inconsistent continuous store, respectively. We denote with $\mathrm{dom}(\tilde{c}) \subseteq \widetilde{Var}$ the domain of $\tilde{c}$. Given $\tilde{c} \in \tilde{\mathcal{C}}$ and $x \in \mathrm{dom}(\tilde{c})$, $\tilde{c}(x) = \langle v, f \rangle$ means that x has value v (denoted as $\tilde{c}(x).v$) and flow f (denoted as $\tilde{c}(x).f$). The binary operator $\tilde{\wedge} : \tilde{\mathcal{C}} \times \tilde{\mathcal{C}} \to \tilde{\mathcal{C}}$ merges the information from two continuous stores. In the case the same variable appears in both stores with different values or flows, their merge is inconsistent. Given $\tilde{c}, \tilde{d} \in \tilde{\mathcal{C}}$:

$$\tilde{c} \,\tilde{\wedge}\, \widetilde{true} = \tilde{c} \qquad \tilde{c} \,\tilde{\wedge}\, \widetilde{false} = \widetilde{false}$$

$$\tilde{c} \,\tilde{\wedge}\, \tilde{d} = \widetilde{false} \quad \text{if } \exists x \in \mathrm{dom}(\tilde{c}) \cap \mathrm{dom}(\tilde{d}).\ \tilde{c}(x) \neq \tilde{d}(x)$$

$$\tilde{c} \,\tilde{\wedge}\, \tilde{d} = \lambda y. \begin{cases} \tilde{c}(y) & \text{if } y \in \mathrm{dom}(\tilde{c}) \\ \tilde{d}(y) & \text{if } y \in \mathrm{dom}(\tilde{d}) \end{cases} \quad \text{if } \forall x \in \mathrm{dom}(\tilde{c}) \cap \mathrm{dom}(\tilde{d}).\ \tilde{c}(x) = \tilde{d}(x)$$

We define the operator $\tilde{\exists} : Var \times \tilde{\mathcal{C}} \to \tilde{\mathcal{C}}$ such that, given $\tilde{c} \in \tilde{\mathcal{C}}$ and $x \in \widetilde{Var}$, $\tilde{\exists}_x \tilde{c}$ deletes the information about x in $\tilde{c}$.

Given $\tilde{c} \in \tilde{\mathcal{C}}$, $x \in dom(\tilde{c})$ and $v \in \mathbb{R}$, we denote as $\tilde{c}[v/x]$ the continuous store that is equal to $\tilde{c}$ except for the value of x that becomes v.

$$\tilde{c}[v/x] = \lambda y. \begin{cases} \tilde{c}(y) & \text{if } y \in \mathrm{dom}(\tilde{c}), y \neq x \\ (v, \tilde{c}(x).f) & \text{if } y = x \end{cases}$$

A Hy-*tccp* store is a pair $\langle c, \tilde{c} \rangle$ where $c \in \mathcal{C}$ (discrete store) is a monotonic constraint store as in *tccp* and $\tilde{c} \in \tilde{\mathcal{C}}$ (continuous store) is such that $c \wedge \bigwedge_{x \in \mathrm{dom}(\tilde{c})} (x = \tilde{c}(x).v) \neq false$, i.e., discrete and continuous store are consistent[1]. We denote as Γ the set of all possible Hy-*tccp* stores. We define the extension of the entailment relation $\vdash$ over Hy-*tccp* stores as $\tilde{\vdash} : \Gamma \times \mathcal{C}$ such that given $\langle c, \tilde{c} \rangle \in \Gamma$ and $d \in \mathcal{C}$, $\langle c, \tilde{c} \rangle \tilde{\vdash} d$ if $c \wedge \bigwedge_{x \in \mathrm{dom}(\tilde{c})} (x = \tilde{c}(x).v) \vdash d$. In other words, a store $\langle c, \tilde{c} \rangle$ entails a constraint d if the discrete store c

[1]We assume that our underlying constraint system handles equality constraints.

merged with the projection of the current values of the continuous variables entails d in the underlying constraint system.

Given $\tau \in \mathbb{R}^+$ we denote as $\langle c, \tilde{c}_\tau \rangle$ the continuous projection of the store $\langle c, \tilde{c} \rangle$ at time τ: the values of the continuous variables are updated at time τ, while the flows are unchanged: $\tilde{c}_\tau = \lambda y. \tilde{c}[n_y/y]$ where $y \in \mathrm{dom}(\tilde{c})$ and $n_y = \tilde{c}(y).v + (\tilde{c}(y).f * \tau)$. For instance consider the store $\langle x > 10, y \mapsto (2,5) \rangle$, its projection at time 3 is the store $\langle x > 10, y \mapsto (17,5) \rangle$. We say that $\langle c, \tilde{c}_\tau \rangle$ is a continuous projection of $\langle c, \tilde{c} \rangle$ at time τ that satisfies d (denoted as $\langle c, \tilde{c} \rangle \rightsquigarrow^d_\tau \langle c, \tilde{c}_\tau \rangle$) if for all $\tau' \in [0, \tau]$ $\langle c, \tilde{c}_{\tau'} \rangle \tilde{\vdash} d$. For instance, the above projection satisfies $y > 0$: $\langle x > 10, y \mapsto (2,5) \rangle \rightsquigarrow^{y>0}_3 \langle x > 10, y \mapsto (17,5) \rangle$.

The update operator $\lhd$, given $\tilde{c}, \tilde{d} \in \tilde{\mathcal{C}}$ such that $\mathrm{dom}(\tilde{c}) \cap \mathrm{dom}(\tilde{d}) = \{x_1, \ldots, x_n\}$, updates $\tilde{c}$ with the information of $\tilde{d}$ as follows: $\tilde{c} \lhd \tilde{d} := (\tilde{\exists}_{x_1,\ldots,x_n} \tilde{c}) \tilde{\wedge} \tilde{d}$. Note that it is impossible to obtain an inconsistent continuous store since the common variables are hidden from $\tilde{c}$ and replaced by the new values and flows from $\tilde{d}$.

In order to model the typical behaviors of hybrid systems we introduce two new constructs w.r.t. the syntax of *tccp*: change and $\widetilde{\mathsf{ask}}$.

The agent change updates the current continuous store with a new value and/or flow for a given continuous variable. It roughly corresponds to the *reset* predicate of hybrid automata.

Continuous transitions are modeled by the new construct $\widetilde{\mathsf{ask}}(inv)$ that makes continuous variables evolve over *continuous* time while the invariant *inv* is satisfied. The *tccp* choice agent is extended by allowing the non-deterministic choice between discrete and continuous transitions in the following way:

$$\textstyle\sum_{i=1}^n \mathsf{ask}(c_i) \to A + \sum_{j=1}^m \widetilde{\mathsf{ask}}(inv_j).$$

Here, the $\widetilde{\mathsf{ask}}$ branches can be non-deterministically selected when the corresponding invariant inv_j is entailed in the current store. The continuous variables evolve over time while inv_j holds and until another ask branch is selected.

The syntax of Hy-*tccp* agents is given by the following grammar:

$$\begin{gathered} A ::= \mathsf{stop} \mid \mathsf{tell}(c) \mid A \parallel A \mid \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ A \mid \exists x A \mid p(\vec{x}) \mid \\ \mathsf{change}(y, v, f) \mid \textstyle\sum_{i=1}^n \mathsf{ask}(c_i) \to A + \sum_{j=1}^m \widetilde{\mathsf{ask}}(inv_j) \end{gathered}$$

where c, c_i and inv_j are finite constraints in $\mathcal{C}$, y is a continuous variable in $\widetilde{Var}$, v, $f \in \mathbb{R}$, p is a process symbol, $x \in Var \cup \widetilde{Var}$, $\vec{x} \in (Var \cup \widetilde{Var}) \times \cdots \times (Var \cup \widetilde{Var})$, $n \geq 0$ and $m \geq 0$.

The *operational semantics* of Hy-*tccp* is described by a transition system $T = (\widetilde{Conf}, \to_\sigma, \to_\tau)$. Configurations in $\widetilde{Conf}$ are triples $\langle A, c, \tilde{c} \rangle$ representing the agent A to be executed in the current extended store $\langle c, \tilde{c} \rangle$. In contrast to the *tccp* approach, the discrete transition relation $\to_\sigma \subseteq \widetilde{Conf} \times \widetilde{Conf}$ does not represent the passage of one time unit. Instead, it models a computational step which does not consume time but it is needed to synchronize the agents in parallel. The continuous passage of time is modeled by the transition relation $\to_\tau \subseteq \widetilde{Conf} \times \widetilde{Conf}$ where $\tau \in \mathbb{R}^+$ is a (strictly) positive real number that indicates the duration of the transition. In Figure 3, we formally describe the operational semantics of Hy-*tccp*. Wherever possible we will use the subindex $\lambda \in \mathbb{R}^+ \cup \{\sigma\}$ to represent both kinds of transitions (discrete and continuous).

Rule **R1** shows the effects of adding a constraint $c \in \mathcal{C}$ to the current discrete store. In Rule **R1'**, the agent change updates the continuous store $\tilde{d}$ with a new initial value v and a new flow f for the variable y by using the update operator $\lhd$.

Rules **R2** and **R2'** describe the non-deterministic choice behavior. Rule **R2** represents the discrete transition that is performed when one of the ask guards is entailed in the current store. In this case the corresponding agent is executed in the next step. Rule **R2'** models the continuous evolution of the system while one of the $\widetilde{\mathsf{ask}}$ invariants holds in the store. After a continuous transition of duration τ, the values of the variables in the continuous store $\tilde{d}$ are updated while the discrete store is unchanged. At the end of that transition the non-deterministic choice is executed again allowing another discrete or continuous branch to be selected. In the case no guard or invariant holds this agent suspends.

Rules **R3**, **R3'**, **R4** and **R4'** describe the behavior of agent now. This agent behaves as A if c is entailed by the constraint store, otherwise it behaves as B.

Rule **R5** represents the parallel execution of two discrete transitions in terms of maximal parallelism, i.e., all the enabled agents of A and B are executed at the same time. Rule **R6** represents the parallel execution of two continuous transitions, note that their duration must coincide. Rule **R7** expresses the parallel composition of a discrete and a continuous transition. In this case, the discrete transition is executed before the continuous one. Rule **R8** states that when an agent is blocked, the other one performs its transition (discrete or continuous).

In Rule **R9**, the agent $\exists^{\langle l,\tilde{l}\rangle} xA$ makes variable x local to A. It behaves like A with x considered local, i.e., the information on x provided by the external environment is hidden from A by using the $\tilde{\exists}$ operator, and, in the same way, the information on x produced by A is hidden from the global environment. The store $\langle l,\tilde{l}\rangle$ in the agent $\exists^{\langle l,\tilde{l}\rangle} xA$ represents the store local to A. This auxiliary operator is linked to the hiding construct by setting the initial local store to $\langle true, \widetilde{true}\rangle$, thus $\exists xA := \exists^{\langle true,\widetilde{true}\rangle} xA$.

Finally, in Rule **R10**, the agent $p(\vec{x})$ takes from D a declaration of the form $p(\vec{x}) :- A$ and executes A.

Let us formalize the notion of behavior of a Hy-*tccp* program P in terms of the transition system described in Figure 3. The small-step operational behavior of Hy-*tccp* collects all the small-step computations associated with P (in terms of sequences of Hy-*tccp* stores closed by prefix) for each possible initial store. We assume that subsequent continuous transitions are considered as a unique (maximal) one whose length is equal to the sum of all the subsequent transition lengths. For instance, a sequence of continuous transitions of the form $\langle A_0, c_0, \tilde{c}_0\rangle \rightarrow_{\tau_1} \ldots \rightarrow_{\tau_n} \langle A_n, c_n, \tilde{c}_n\rangle$ is considered as the unique transition $\langle A_0, c_0, \tilde{c}_0\rangle \rightarrow_{\tau} \langle A_n, c_n, \tilde{c}_n\rangle$ where $\tau = \sum_{i=1}^{n} \tau_i$[2].

Definition 3.1 *Let $P = D.A$ be a* Hy-*tccp program. The* small-step (observable) behavior *of P is defined as:*

$$\mathcal{B}^{ss}[\![D.A]\!] := \bigcup_{\langle c_0,\tilde{c}_0\rangle\in\Gamma} \Big\{\langle c_0,\tilde{c}_0\rangle\cdot\langle c_1,\tilde{c}_1\rangle\cdot\ldots\cdot\langle c_n,\tilde{c}_n\rangle \mid \langle A,c_0,\tilde{c}_0\rangle \rightarrow_{\lambda_1} \langle A_1,c_1,\tilde{c}_1\rangle \\ \rightarrow_{\lambda_2} \ldots \rightarrow_{\lambda_n} \langle A_n,c_n,\tilde{c}_n\rangle,\ \forall 1\le i\le n.\ \lambda_n \in \mathbb{R}^+ \cup \{\sigma\}\Big\} \cup \{\varepsilon\}$$

4 Examples

In order to show the expressivity of Hy-*tccp*, we present some examples of hybrid systems described in this language. For each case, we present the Hy-*tccp* code and the corresponding hybrid automaton.

4.1 Cooler system

In Figure 4 we model in Hy-*tccp* the cooler system introduced in Example 2.5. The initial state of the cooler is set to *off* and the temperature T initially has value 29 and changes with a rate of +2.0. The temperature value increases continuously over time (first $\widetilde{\mathsf{ask}}$) until the temperature is lower than or equal

[2]We assume that our system does not exhibit Zeno behaviors.

$$\frac{}{\langle \mathsf{tell}(c), d, \tilde{d}\rangle \rightarrow_\sigma \langle \mathsf{stop}, c \wedge d, \tilde{d}\rangle} \qquad \textbf{(R1)}$$

$$\frac{}{\langle \mathsf{change}(y, v, f), d, \tilde{d}\rangle \rightarrow_\sigma \langle \mathsf{stop}, d, \tilde{d} \triangleleft (y \mapsto (v, f))\rangle} \qquad \textbf{(R1')}$$

$$\frac{\exists 1 \leq k \leq n\,.\,\langle d, \tilde{d}\rangle \tilde{\vdash} c_k}{\langle \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i + \sum_{j=1}^{m} \widetilde{\mathsf{ask}}(inv_j), d, \tilde{d}\rangle \rightarrow_\sigma \langle A_k, d, \tilde{d}\rangle} \qquad \textbf{(R2)}$$

$$\frac{\exists 1 \leq k \leq m,\, \tau \in \mathbb{R}^+ . \langle d, \tilde{d}\rangle \leadsto_{\tau}^{inv_k} \langle d, \tilde{d}_\tau\rangle}{\langle \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i + \sum_{j=1}^{m} \widetilde{\mathsf{ask}}(inv_j), d, \tilde{d}\rangle \rightarrow_\tau \langle \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i + \sum_{j=1}^{m} \widetilde{\mathsf{ask}}(inv_j), d, \tilde{d}_\tau\rangle} \qquad \textbf{(R2')}$$

$$\frac{\langle A, d, \tilde{d}\rangle \rightarrow_\lambda \langle A', d', \tilde{d}'\rangle \quad \lambda \in \mathbb{R}^+ \cup \{\sigma\} \quad \langle d, \tilde{d}\rangle \tilde{\vdash} c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d, \tilde{d}\rangle \rightarrow_\lambda \langle A', d', \tilde{d}'\rangle} \qquad \textbf{(R3)}$$

$$\frac{\langle A, d, \tilde{d}\rangle \not\rightarrow_\lambda \quad \lambda \in \mathbb{R}^+ \cup \{\sigma\} \quad \langle d, \tilde{d}\rangle \tilde{\vdash} c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d, \tilde{d}\rangle \rightarrow_\sigma \langle A, d, \tilde{d}\rangle} \qquad \textbf{(R3')}$$

$$\frac{\langle B, d, \tilde{d}\rangle \rightarrow_\lambda \langle B', d', \tilde{d}'\rangle \quad \lambda \in \mathbb{R} \cup \{\sigma\} \quad \langle d, \tilde{d}\rangle \tilde{\nvdash} c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d, \tilde{d}\rangle \rightarrow_\lambda \langle B', d', \tilde{d}'\rangle} \qquad \textbf{(R4)}$$

$$\frac{\langle B, d, \tilde{d}\rangle \not\rightarrow_\lambda \quad \lambda \in \mathbb{R} \cup \{\sigma\} \quad \langle d, \tilde{d}\rangle \tilde{\nvdash} c}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B, d, \tilde{d}\rangle \rightarrow_\sigma \langle B, d, \tilde{d}\rangle} \qquad \textbf{(R4')}$$

$$\frac{\langle A, d, \tilde{d}\rangle \rightarrow_\sigma \langle A', d', \tilde{d}'\rangle \quad \langle B, d, \tilde{d}\rangle \rightarrow_\sigma \langle B', d'', \tilde{d}''\rangle}{\langle A \parallel B, d, \tilde{d}\rangle \rightarrow_\sigma \langle A' \parallel B', d' \wedge d'', \tilde{d}' \tilde{\wedge} \tilde{d}''\rangle} \qquad \textbf{(R5)}$$

$$\frac{\langle A, d, \tilde{d}\rangle \rightarrow_\tau \langle A, d, \tilde{d}'\rangle \quad \langle B, d, \tilde{d}\rangle \rightarrow_\tau \langle B, d, \tilde{d}'\rangle \quad \tau \in \mathbb{R}^+}{\langle A \parallel B, d, \tilde{d}\rangle \rightarrow_\tau \langle A \parallel B, d, \tilde{d}'\rangle} \qquad \textbf{(R6)}$$

$$\frac{\langle A, d, \tilde{d}\rangle \rightarrow_\sigma \langle A', d', \tilde{d}'\rangle \quad \langle B, d, \tilde{d}\rangle \rightarrow_\tau \langle B, d, \tilde{d}''\rangle \quad \tau \in \mathbb{R}^+}{\langle A \parallel B, d, \tilde{d}\rangle \rightarrow_\sigma \langle A' \parallel B, d', \tilde{d}'\rangle} \qquad \textbf{(R7)}$$

$$\frac{\langle A, d, \tilde{d}\rangle \rightarrow_\lambda \langle A', d', \tilde{d}'\rangle \quad \langle B, d, \tilde{d}\rangle \not\rightarrow_{\lambda'} \quad \lambda, \lambda' \in \mathbb{R}^+ \cup \{\sigma\}}{\begin{array}{c}\langle A \parallel B, d, \tilde{d}\rangle \rightarrow_\lambda \langle A' \parallel B, d', \tilde{d}'\rangle \\ \langle B \parallel A, d, \tilde{d}\rangle \rightarrow_\lambda \langle B \parallel A', d', \tilde{d}'\rangle\end{array}} \qquad \textbf{(R8)}$$

$$\frac{\langle A, l \wedge \exists_x d, \tilde{l} \tilde{\wedge} \tilde{\exists}_x \tilde{d}\rangle \rightarrow_\lambda \langle B, l', \tilde{l}'\rangle \quad \lambda \in \mathbb{R}^+ \cup \{\sigma\}}{\langle \exists^{\langle l, \tilde{l}\rangle} x A, d, \tilde{d}\rangle \rightarrow_\lambda \langle \exists^{\langle l', \tilde{l}'\rangle} x B, d \wedge \exists_x l', \tilde{d} \tilde{\wedge} \tilde{\exists}_x \tilde{l}'\rangle} \qquad \textbf{(R9)}$$

$$\frac{p(\vec{x}) :- A \in D}{\langle p(\vec{x}), d, \tilde{d}\rangle \rightarrow_\sigma \langle A, d, \tilde{d}\rangle} \qquad \textbf{(R10)}$$

Figure 3: The transition system for Hy-*tccp*.

$$
\begin{aligned}
&\texttt{init} :- \exists\, St, T\ \big(\mathsf{tell}(St = [\mathit{off} \mid \_]) \parallel \mathsf{change}(T, 29, +2.0) \parallel \mathsf{tell}(T \geq 26 \wedge T \leq 30) \parallel \texttt{cooler}(St, T)\big)\\
&\texttt{cooler}(St, T) :- \exists\, St' \Big(\ \widetilde{\mathsf{ask}}(St = [\mathit{off} \mid \_] \wedge T \leq 30)\\
&\qquad + \mathsf{ask}(St = [\mathit{off} \mid \_] \wedge T = 30) \rightarrow \big(\mathsf{tell}(St = [\mathit{off} \mid St']) \parallel \mathsf{tell}(St' = [\mathit{on} \mid \_]) \parallel \mathsf{change}(T, 30, -0.5) \parallel \texttt{cooler}(St', T)\big)\\
&\qquad + \widetilde{\mathsf{ask}}(St = [\mathit{on} \mid \_] \wedge T \geq 26)\\
&\qquad + \mathsf{ask}(St = [\mathit{on} | \_] \wedge T = 26) \rightarrow \big(\mathsf{tell}(St = [\mathit{on} \mid St']) \parallel \mathsf{tell}(St' = [\mathit{off} \mid \_]) \parallel \mathsf{change}(T, 26, +2.0) \parallel \texttt{cooler}(St', T)\big)\Big)
\end{aligned}
$$

Figure 4: Hy-*tccp* model for the cooler system

to the value of 30. When the temperature reaches this limit, the cooler is turned on and the flow of the temperature changes from +2.0 to −0.5 (first ask). At this point, the temperature starts decreasing until it reaches the value of 26 (second $\widetilde{\mathsf{ask}}$). When this happens, the cooler is turned off and the flow of the temperature is changed again to +2.0 (second ask).

It is worth noting that, due to the monotonicity of the discrete constraint store, streams (written in a list-fashion way) are used to model *imperative-style* variables [2]. A stream is a list of the form $St = [on \mid St']$ where the head *on* represents the current value of *St*, and the tail *St'* is a free variable that will be instantiated with the future values of *St*. Observe that we use the global constraint $T \geq 26 \wedge T \leq 30$ to add a global invariant of the cooler system ensuring that the temperature always stays in the interval $[26, 30]$.

The following partial trace represents the small-step behavior (see Definition 3.1) of $\texttt{cooler}(St, T)$ starting from the initial store $\langle St = [\mathit{off} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (29, +2.0)\rangle$. This means that, initially, the cooler is turned off and the temperature has a value of 29 and a flow of +2.0. Moreover, the temperature is constrained to be between the values 26 and 30. Observe how the values *on* and *off* are accumulated in the stream *St* in order to model the evolution of the state. The current state corresponds to the last value added to the stream. We use _ to indicate that the tail of the stream *St* is a free variable that can be instantiated with future values. The continuous variables evolve over time until another discrete transition is executed. The repeated equal stores occurring in the trace correspond to the discrete computational steps taken in Hy-*tccp* (as well as in *tccp*) to evaluate one of the ask guards or to perform a procedure call. These steps are necessary to synchronize parallel agents. For sake of clarity, we explicitly indicate the kind of transition occurring between two states (we write σ for discrete transitions and the duration $\tau \in \mathbb{R}^+$ for continuous ones).

$$
\begin{aligned}
&\langle St = [\mathit{off} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (29, +2.0)\rangle \cdot_{0.5} \langle St = [\mathit{off} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (30, +2.0)\rangle \cdot_{\sigma}\\
&\langle St = [\mathit{off} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (30, +2.0)\rangle \cdot_{\sigma} \langle St = [\mathit{off}, \mathit{on} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (30, -0.5)\rangle \cdot_{\sigma}\\
&\langle St = [\mathit{off}, \mathit{on} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (30, -0.5)\rangle \cdot_{8} \langle St = [\mathit{off}, \mathit{on} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (26, -0.5)\rangle \cdot_{\sigma}\\
&\langle St = [\mathit{off}, \mathit{on} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (26, -0.5)\rangle \cdot_{\sigma} \langle St = [\mathit{off}, \mathit{on}, \mathit{off} \mid \_] \wedge T \geq 26 \wedge T \leq 30,\ T \mapsto (26, +2.0)\rangle \ldots
\end{aligned}
$$

4.2 Cat and mouse race

We consider the cat and mouse problem proposed in [7] (see Figure 5 for the corresponding hybrid automaton). The Hy-*tccp* code of this model is shown in Figure 6. The positions of the cat and the mouse are modeled by two continuous variables, called *C* and *M* respectively. A mouse starts running from the point of origin at a speed of 10 meters/second ($\mathsf{change}(M, 0, 10.0)$) towards a hole that is 100 meters

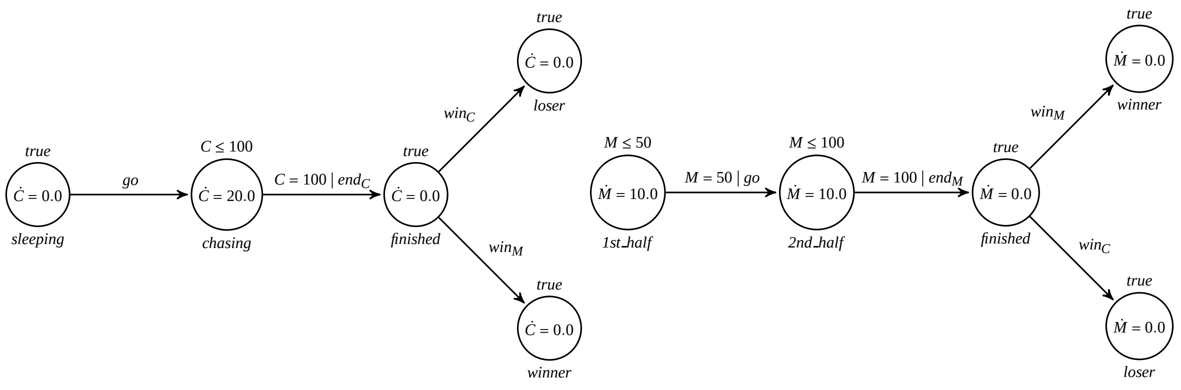

Figure 5: Hybrid automata for the cat and mouse problem

```
init :− mouse ∥ cat ∥ controller
mouse :− $\exists M\Big(\mathsf{change}(M,0,10.0) \parallel$
      $\big(\widetilde{\mathsf{ask}}(M \leq 50)$
      $+\,\mathsf{ask}(M = 50) \rightarrow (\mathsf{tell}(go) \parallel (\widetilde{\mathsf{ask}}(M \leq 100)$
                    $+\,\mathsf{ask}(M = 100) \rightarrow (\mathsf{tell}(end_m) \parallel \mathsf{ask}(win_m) \rightarrow claimPrize(\ldots)$
                                        $+\,\mathsf{ask}(win_c) \rightarrow \mathsf{stop})))\big)\Big)$
cat :− $\exists C\Big(\mathsf{ask}(go) \rightarrow \big(\mathsf{change}(C,0,20.0) \parallel$
                 $(\widetilde{\mathsf{ask}}(C \leq 100)$
                  $+\,\mathsf{ask}(C = 100) \rightarrow (\mathsf{tell}(end_c) \parallel \mathsf{ask}(win_c) \rightarrow claimPrize(\ldots)$
                                      $+\,\mathsf{ask}(win_m) \rightarrow \mathsf{stop}))\big)\Big)$
controller :− $\mathsf{ask}(end_m) \rightarrow \mathsf{tell}(win_m) + \mathsf{ask}(end_c) \rightarrow \mathsf{tell}(win_c)$
```

Figure 6: Hy-*tccp* model for the cat and mouse race

away. After it has run 50 meters it sends a signal to the cat ($\mathsf{tell}(go)$) and continues its run. When the cat receives the signal *go*, it starts chasing the mouse from the point of origin at a speed of 20 meters/second ($\mathsf{change}(C,0,20.0)$). The cat wins if it catches the mouse before it reaches the hole, otherwise it loses. At the end of their run, the mouse and the cat send a message to the `controller` (end_M and end_C, respectively), which decides non-deterministically the winner and informs of it through a signal (win_m or win_c). The winner, at this point, can claim his prize.

4.3 Gear shift system

The hybrid automaton in Figure 7 represents a car gear shift system. Each location models a gear (1, 2 or 3) and the fact that the speed is either increasing or decreasing (↑ or ↓ respectively). When the speed increases (respectively decreases) over time and it reaches a given threshold, the current gear is changed to the upper (respectively lower) one. When a signal of danger (*dng*) is received, the system changes the current gear to the lower one and the speed starts decreasing. At this point, when a signal of safe situation (*safe*) is received, the system is allowed to stay in the current location as well as to increase the

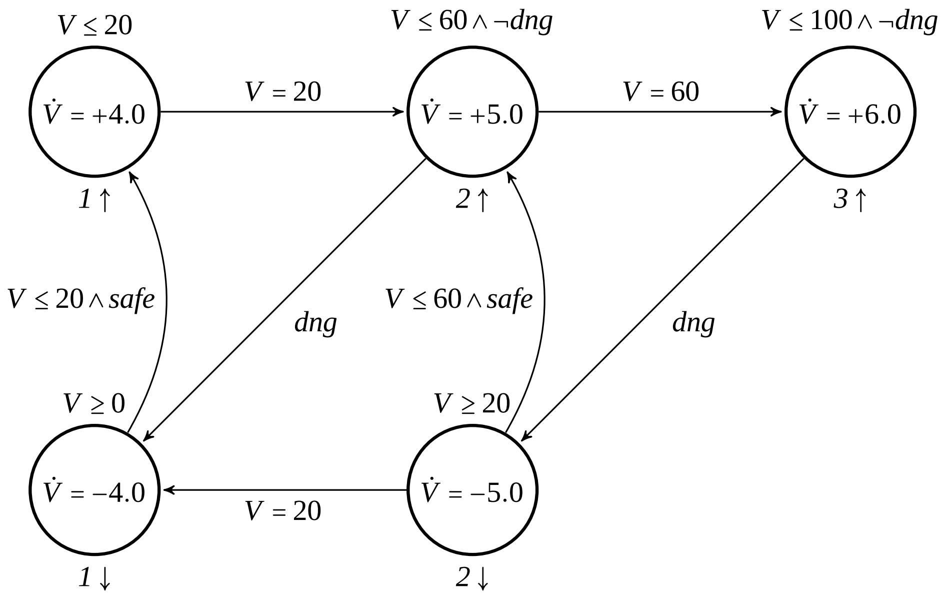

Figure 7: Hybrid automaton for the gear shift system

speed. The latter case is modeled by the transitions from location $1\downarrow$ to location $1\uparrow$, and from $2\downarrow$ to $2\uparrow$.

The Hy-*tccp* program modeling this system is shown in Figure 8. The stream G stores the evolution of the gear state. The $\widetilde{\texttt{ask}}$ statements model the five locations of the automaton of Figure 7, i.e., the possible cases in which a continuous transition is performed. It is worth noting that the invariant of each location is modeled by the guard of the corresponding $\widetilde{\texttt{ask}}$ statement. The first three ask statements model the gearbox shifting automatically into a higher (respectively lower) gear if the speed V reaches the upper (respectively lower) threshold of the current gear. The watcher informs to the gearbox about the current external situation (danger or safe), through channel WG. When gearbox receives a danger signal *dng* and the speed is growing (fourth and fifth ask branches), it moves to a lower gear, and changes the speed flow from positive to negative by means of a change agent. Otherwise, when it receives a safety signal *safe* and the speed is decreasing (sixth and seventh ask branches), it is allowed to change the speed flow from negative to positive.

5 Related Work

In [7], *hcc* was introduced as the first extension over continuous time of the concurrent constraint paradigm. Although both Hy-*tccp* and *hcc* are declarative languages with a logical nature, there are some important differences between them. First of all, Hy-*tccp* is a non-deterministic language, while *hcc* is deterministic. We believe that this is an essential feature for modeling hybrid systems, which are inherently non-deterministic. Hy-*tccp* has been defined as a modeling language for hybrid systems in the style of hybrid automata. This means that we aim to obtain programs with a structure similar to that of hybrid automata, but described in a more abstract way. The non-deterministic choice is a powerful construct that allows the set of all possible transitions of an hybrid automata to be expressed as a list of ask and $\widetilde{\texttt{ask}}$ branches. Furthermore, in *hcc*, the information on the value and flow of continuous variables is modeled as a constraint of the underlying continuous constraint system. On the contrary, in Hy-*tccp*,

$$
\begin{array}{l}
\mathtt{init} :\!-\ \exists\, V, G, WG\big(\,\mathsf{tell}(G = [1\uparrow\!|\,\_]) \parallel \mathsf{change}(V, 0, +4.0) \parallel \mathsf{tell}(V \geq 0 \wedge V \leq 100) \parallel \mathtt{gearbox}(G, WG, V) \parallel \mathtt{watcher}(WG)\big) \\
\mathtt{gearbox}(G, WG, V) :\!-\ \exists\, G', WG'\big(\\
\quad \widetilde{\mathsf{ask}}(G = [1\uparrow\!|\,\_] \wedge V \leq 20) + \widetilde{\mathsf{ask}}(G = [2\uparrow\!|\,\_] \wedge V \leq 60 \wedge WG \neq [dng \mid \_]) + \widetilde{\mathsf{ask}}(G = [3\uparrow\!|\,\_] \wedge V \leq 100 \wedge WG \neq [dng \mid \_]) \\
+\, \widetilde{\mathsf{ask}}(G = [1\downarrow\!|\,\_] \wedge V \geq 0) + \widetilde{\mathsf{ask}}(G = [2\downarrow\!|\,\_] \wedge V \geq 20) \\
+\, \mathsf{ask}(G = [1\uparrow\!|\,\_] \wedge V = 20) \rightarrow \big(\,\mathsf{tell}(G = [1\uparrow\!|\, G']) \parallel \mathsf{tell}(G' = [2\uparrow\!|\,\_]) \parallel \mathsf{change}(V, \_, +5.0) \parallel \mathtt{gearbox}(G', WG, V)\big) \\
+\, \mathsf{ask}(G = [2\uparrow\!|\,\_] \wedge V = 60) \rightarrow \big(\,\mathsf{tell}(G = [2\uparrow\!|\, G']) \parallel \mathsf{tell}(G' = [3\uparrow\!|\,\_]) \parallel \mathsf{change}(V, \_, +6.0) \parallel \mathtt{gearbox}(G', WG, V)\big) \\
+\, \mathsf{ask}(G = [2\downarrow\!|\,\_] \wedge V = 20) \rightarrow \big(\,\mathsf{tell}(G = [2\downarrow\!|\, G']) \parallel \mathsf{tell}(G' = [1\downarrow\!|\,\_]) \parallel \mathsf{change}(V, \_, -4.0) \parallel \mathtt{gearbox}(G', WG, V)\big) \\
+\, \mathsf{ask}(G = [2\uparrow\!|\,\_] \wedge WG = [dng \mid \_]) \rightarrow \big(\,\mathsf{tell}(G = [2\uparrow\!|\, G']) \parallel \mathsf{tell}(G' = [1\downarrow\!|\,\_]) \parallel \mathsf{tell}(WG = [dng \mid WG']) \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{change}(V, \_, -4.0) \parallel \mathtt{gearbox}(G', WG', V)\big) \\
+\, \mathsf{ask}(G = [3\uparrow\!|\,\_] \wedge WG = [dng \mid \_]) \rightarrow \big(\,\mathsf{tell}(G = [3\uparrow\!|\, G']) \parallel \mathsf{tell}(G' = [2\downarrow\!|\,\_]) \parallel \mathsf{tell}(WG = [dng \mid WG']) \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{change}(V, \_, -5.0) \parallel \mathtt{gearbox}(G', WG', V)\big) \\
+\, \mathsf{ask}(G = [1\downarrow\!|\,\_] \wedge WG = [safe \mid \_] \wedge V \leq 20) \rightarrow \big(\,\mathsf{tell}(G = [1\downarrow\!|\, G']) \parallel \mathsf{tell}(G' = [1\uparrow\!|\,\_]) \parallel \mathsf{tell}(WG = [safe \mid WG']) \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{change}(V, \_, +4.0) \parallel \mathtt{gearbox}(G', WG', V)\big) \\
+\, \mathsf{ask}(G = [2\downarrow\!|\,\_] \wedge WG = [safe \mid \_] \wedge V \leq 60) \rightarrow \big(\,\mathsf{tell}(G = [2\downarrow\!|\, G']) \parallel \mathsf{tell}(G' = [2\uparrow\!|\,\_]) \parallel \mathsf{tell}(WG = [safe \mid WG']) \parallel \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{change}(V, \_, +5.0) \parallel \mathtt{gearbox}(G', WG', V)\big)\big) \\
\mathtt{watcher}(WG) :\!-\ \exists\, WG'\big(\quad \widetilde{\mathsf{ask}}(true) \\
\qquad\qquad\qquad\qquad\qquad +\, \mathsf{ask}(true) \rightarrow \big(\,\mathsf{tell}(WG{=}[safe \mid WG']) \parallel \mathtt{watcher}(WG')\big) \\
\qquad\qquad\qquad\qquad\qquad +\, \mathsf{ask}(true) \rightarrow \big(\,\mathsf{tell}(WG{=}[dng \mid WG']) \parallel \mathtt{watcher}(WG')\big)\big)
\end{array}
$$

Figure 8: Hy-*tccp* model for a gear shift system

there is a clear distinction between discrete and continuous variables. In *hcc* the positive information in the store must be transferred by using the agent hence. In contrast, in Hy-*tccp* the positive information in the discrete store is transferred automatically from one step to the next.

In [1] and [4], two process algebras for hybrid systems have been defined: *Hybrid Chi* and *HyPa*, respectively. The process algebra *Hybrid Chi* [1] shares with Hy-*tccp* the separation between discrete and continuous variables, the synchronous nature and the concept of delayable guard (corresponding to the suspension of the non-deterministic choice). *HyPa* [4] was introduced as an extension of the process algebra *ACP*. It differs from *Hybrid Chi* mainly in the way time-determinism is treated, and in the modeling of time passing.

6 Conclusions

In this paper we have presented Hy-*tccp*, an extension of *tccp* over continuous time with the aim of modeling hybrid systems in a declarative and logical way by abstracting away from all the implementation details. Hy-*tccp* has been introduced as a synchronous and non-deterministic language defining computations similar to that of hybrid automata.

Hy-*tccp* has several advantages that make it suitable for modeling hybrid systems. Its declarative nature facilitates a high level description close to that of hybrid automata. In addition, the logical nature of Hy-*tccp* eases the development of formal methods techniques for the static analysis and verification of hybrid systems. Furthermore, since Hy-*tccp* is a conservative extension of *tccp*, it is possible to describe with the same syntax concurrent, reactive and hybrid systems.

In the future, we plan to develop a framework for the description and simulation of Hy-*tccp* programs. In this way, we will be able to model complex hybrid systems in Hy-*tccp*. Given the affinity of the two formalisms, we are interested in defining a translation rules system from Hy-*tccp* to hybrid automata and viceversa, in order to transfer verification and analysis results from one formalism to the other. Furthermore, we plan to use model checking and abstract interpretation techniques to verify temporal properties of hybrid systems written in Hy-*tccp* (as done in [6] for SPIN and in [3] for *tccp*). Another feature we would like to explore is the adjustment of the language to make it compatible with rectangular hybrid automata [9].

References

[1] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda & R. R. H. Schiffelers (2006): *Syntax and consistent equation semantics of hybrid Chi*. *Journal of Logic and Algebraic Programming* 68(1-2), pp. 129–210, doi:10.1016/j.jlap.2005.10.005.

[2] F. S. de Boer, M. Gabbrielli & M. C. Meo (2000): *A Timed Concurrent Constraint Language*. *Information and Computation* 161(1), pp. 45–83, doi:10.1006/inco.1999.2879.

[3] M. Comini, L. Titolo & A. Villanueva (2014): *Abstract Diagnosis for* tccp *using a Linear Temporal Logic*. *Theory and Practice of Logic Prog.* 14(4-5), pp. 787–801, doi:10.1017/S1471068414000349.

[4] P. J. L. Cuijpers & M. A. Reniers (2005): *Hybrid process algebra*. *Journal of Logic and Algebraic Programming* 62(2), pp. 191–245, doi:10.1016/j.jlap.2004.02.001.

[5] C. Daws & S. Yovine (1995): *Two Examples of Verification of Multirate Timed Automata with Kronos*. In: *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, IEEE Computer Society, Washington, DC, USA, pp. 66–75, doi:10.1109/REAL.1995.495197.

[6] M. M. Gallardo & L. Panizo (2013): *Extending Model Checkers for Hybrid System Verification: the case study of SPIN*. *Software Testing, Verification and Reliability*, doi:10.1002/stvr.1505.

[7] V. Gupta, R. Jagadeesan, V. A. Saraswat & D. G. Bobrow (1994): *Programming in Hybrid Constraint Languages*. In P.J. Antsaklis, W. Kohn, A. Nerode & S. Sastry, editors: *Hybrid Systems II, Lecture Notes in Computer Science* 999, Springer, pp. 226–251, doi:10.1007/3-540-60472-3_12.

[8] T. A. Henzinger (1996): *The theory of hybrid automata*. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, IEEE Computer Society, Washington, DC, USA, pp. 278–292.

[9] P. W. Kopke (1996): *The Theory of Rectangular Hybrid Automata*. Technical Report, Ithaca, NY, USA.

[10] V. A. Saraswat (1989): *Concurrent Constraint Programming Languages*. Ph.D. thesis, Pittsburgh, PA, USA.