Time Limited Model Checking

M. Falaschi, A. Policriti, A. Villanueva

Dipartimento di Matematica e Informatica
Università di Udine
Via delle Scienze 206
33100 Udine, Italy
{falaschi,policrit,alicia}@dimi.uniud.it

Abstract

In this paper we present an approach to model concurrent systems specified in a temporal concurrent constraint language, which is able to model Hybrid Systems.

We construct a framework in which it is possible to apply the *Model Checking* technique to programs specified in such language.

We present a formalism to transform correctly specification into a Hybrid cc Structure. This structure represents the program behavior by a graph.

Our basic tool is the *Hybrid Concurrent Constraint Programming* (`hybrid cc`) framework defined by Saraswat *et al.* to describe hybrid systems which have a continuous behavior over time but with a discrete control. With this language we take advantage of both the natural properties of the declarative paradigm and of the fact that the notion of continuous time is built into the semantics of the programming language. Following this approach it becomes reasonable to introduce the idea of applying the technique of Model Checking to a *finite* time interval (introduced by the user). With this restriction we naturally force the space representing the behavior of the program to be finite and hence efficient Model Checking algorithms to be applicable. More specifically, we present an automatic transformation from *Hybrid cc Structures* to *linear hybrid automata*, and thus we can use standard model checkers working on timed automata, such as HyTech, in order to verify properties of hybrid systems.

Keywords: Hybrid Concurrent Constraint programming, Hybrid systems, Model Checking

1 Introduction

The main goal of our research is the exploitation of logic programming languages and model checking techniques to support the phases of specification, verification and validation as well as the phases of design and implementation of (hybrid) reactive and real-time systems. Plants or weapon control devices, “fly by wire” aircraft, time critical information systems, embedded applications are only some examples of this important family of systems.

The declarative nature of logic programming languages allows us to reduce the distance between the specification and implementation languages, thus simplifying the process of specification, verification, validation, and synthesis of systems. Furthermore there is a direct mechanism to transform a hybrid automaton into an equivalent `hybrid cc` program which is explained in [4]. Then, if we have to model a system which is intuitively so easy to see as an automaton, it is straightforward to write the `hybrid cc` program that models it.

Here we define a framework that allows us to apply the technique of Model Checking to programs defined in the `hybrid cc` language defined by Saraswat *et al.* in [10, 12]. This language is an extension of the *Default cc* paradigm defined in [18] over (continuous) real time.

Default cc is an extension of the Timed Concurrent Constraint language [18], which refers to a notion of discrete time, which is useful to model reactive systems, that are those which react with their environment at a rate controlled by the environment. However, for hybrid systems it is necessary to consider a notion of time which is 'dense' because these systems evolve continuously over time by keeping a discrete control.

The method of Model Checking is an extremely successful approach to formal verification which has been developed in the last two decades. With this method it is possible to verify a determined behavioral property of a reactive system over a model. It is an algorithmic method that makes an exhaustive enumeration of all the states reachable by the system and analyzes all possible behaviors.

Model Checking has two major advantages: it is fully automatic and its application requires no user supervision or expertise in mathematical disciplines (as opposed to completely deductive techniques) and when the design fails to satisfy a desired property it produces a *counterexample*.

Our goal consists in describing an environment in which is possible to apply Model Checking to a system specified in the `hybrid cc` language. To apply Model Checking with our approach it is necessary to perform three basic tasks:

- to convert the specification of the system into a formalism *almost* accepted by a Model Checking tool,
- to use the concept of time and the initial value of variables to transform the graph structure into a structure which can be handled by some standard Model Checker (we assume initial values and time interval provided by the user),
- to represent the properties that the design must satisfy in an appropriate (logical) formalism.

In this paper we discuss all these tasks with particular attention to the formalism for the representation of the model and to the restrictions for guaranteeing its finiteness. In fact, one of the main features of our approach consists in the fact that, working with a (declarative) constraint based language, we can build abstract versions of collection of models *directly* from the specification. Moreover, we also use the (explicit) notion of time present in `hybrid cc` to bound variables and obtain models in which Model Checking can be performed classically. Indeed, the graph which we build is finite because in a given time instant we consider programs which are essentially determinate (terminating) `hybrid cc` programs, while the evolution of graphs corresponding to different time instants is also finite because of the restrictions we impose on time intervals.

After introducing the definition of hybrid automata, we show a transformation from a Hybrid cc Structure to a linear hybrid automaton ([2]) and we show that the properties of the program behavior can be verified by giving the hybrid automata as input to some standard model checkers, such as HyTech [14].

In some previous work [9] we have considered a language (timed concurrent constraint programming) with a discrete notion of time. In that paper we worked essentially on the first two basic tasks. In fact, in [9] we defined a similar, but simpler, transformation to generate a representation of the system behavior by a graph, but we did not show in that paper any automatic technique to transform the representation into an input for a model checker.

In Section 1.1 we review the most important notions of the `hybrid cc` language, then in Section 1.5 we define an automatic transformation of a `hybrid cc` specification into a graph structure Hybrid cc Structure. In Section 2 we introduce the definition of hybrid automaton and in Section 3 we define some restrictions over our structure and show a transformation from a Hybrid cc Structure to a linear hybrid automaton. In Section 4 we present some conclusions and discuss future work.

1.1 Hybrid Concurrent Constraint programming

Hybrid Concurrent Constraint programming [11, 12] was developed as a paradigm for the modelling, programming and analysis of hybrid systems. Using this methodology the typical advantages of the declarative paradigm are gained. For example, programming in this model is more intuitive and reasoning with the derived languages is easier, than with imperative languages. Another important property is that the specification is executable, that is *what you prove is what you execute*.

In Figure 1 we can see the original syntax of `hybrid cc`.

The Tell operator simply tells us that a holds now. The Positive Ask says that if a holds now, then A holds now whereas the Negative Ask says that if a does not hold now, then A is called (now). The Hiding operator can be seen as an existential quantification over the variable X in the agent A. A, B is the Parallel Composition and imposes that both A and B are called (now). Finally, the `hence` operator tells us that A holds at *every instant after the current one*.

Looking at the definitions, we can see that the notion of negative information needed to model, for example, situations of timeouts is represented by the agent `if` a `else` A. With this language it is possible to model *strong preemptions*, because negative information and execution of the defined actions occur in the

(Agents)	A ::=	a	– Tell
	$\mid$	`if` a `then` A	– Positive Ask
	$\mid$	`if` a `else` A	– Negative Ask
	$\mid$	`new` X `in` A	– Hiding
	$\mid$	A, B	– Parallel Composition
	$\mid$	`hence` A	– Hence

Figure 1: Syntax for `hybrid cc` programs

same time instant [11]. The notion of time is controlled by the `hence` A agent: such agent says that we will execute the agent A at each time instant after the current one. Combining this operator with the positive and negative ask agents it is possible to define all sorts of behaviors over time. For example, a typical watchdog can be modelled with the specification

$$\texttt{new } A \texttt{ in } (\texttt{hence } (\texttt{if } X \texttt{ else } A, \texttt{ if } a \texttt{ then always } X))$$

defining `always` $A = (A,$ `hence` $A)$.

In order to make this operator implementable, in [10, 12] some assumptions are made. The basic intuition behind such assumptions is that most physical systems change "slowly", with points of discontinuous change followed by periods of continuous evolution. With this in mind the *stability* condition is introduced for continuous constraint systems, that guarantees that for every pair of constraints a and b there is a neighborhood around 0 in which a entails b either everywhere or nowhere. This condition implies that the store of the program cannot change an infinite number of times during a finite time interval.

The hiding operator can be problematic as well. In fact, in [10, 12] another restriction is introduced in order to avoid situations in which an infinite number of copies of a single variable would be needed. The authors restrict the variables on which existential quantification can be performed to those variables for which one copy suffices for a continuous evolution. The intuition is that only quantifications over variables for which we can permute the agent for existential quantification and the temporal agent without changing the semantics of the system are allowed.

The constraint system used in `cc` is extended to a continuous constraint system in `hybrid cc` in order to model the notion of a constraint which is true in a period of time.

Execution of a `hybrid cc` program evolves as a sequence of point and interval phases alternately. In a point phase, the Default cc rules are applied in order to obtain a quiescence point. In an interval phase, a new store is calculated using the Default cc rules at the same time. The length of the interval phase is determined as the longest interval for which the status of the asks in the program does not change. Once we have obtained the result we move to another interval point phase.

As in tcc languages, the store of the program is removed when passing from one interval phase to a point phase (in tcc paradigm this occurs when a time step is made) [17]. This problem is solved during the implementation of the interpreter and we consider it when we construct the graph structure passing the information to the point phase.

The reader can find an example of hybrid cc program in Figure 2. This program models a system that has one continuous variable "X" whose derivative is 1 and when it reaches the value 1, then it is set to 0 again.

$$
\begin{array}{ll}
(& X = 0, \\
 & \text{hence (if } prev(X = 1) \text{ then } X = 0), \\
 & \text{hence (if } prev(X = 1) \text{ else } dot(X) = 1))
\end{array}
$$

Figure 2: Example: hybrid cc program

The predicate *prev* asks for the value of constraints in the point where the phase starts (the limit value of the previous phase) whereas the predicate *dot* asks for the value of derivatives of the variable passed as argument. Both are defined in the constraint system ([12]).

1.2 Modelling

One of the first activities in verifying properties of a system is to construct a formal model for the system. This model should capture those properties that will be verified. Both reactive and hybrid systems cannot be modelled by their input-output behavior: it is necessary to capture the *state* of the system, i.e. a description of the system that contains the values of the variables in a specific time instant. We have to model how the state of the system changes when an action occurs (*transition* of the system). In our case the *state transition graph* must be built starting from the hybrid cc formalism introduced above and will eventually contain an explicit notion of time.

1.3 Basics

In order to start with our modelling task we will define a graph structure, inspired by the classical notion of *Kripke Structure*, that allows us to capture the behavior of hybrid systems. We will show how we can obtain automatically such a representation of the system behavior from the system specifications written in hybrid cc.

The set $\mathcal{V}ar$ represents the universal set of variables. $V \subseteq \mathcal{V}ar$ is the set of variables that appear in the program clauses specifying the system properties and describe the *state* of the program in each time instant. Variables in $\mathcal{V}ar$ are typed, where the type of a variable, such as *boolean*, *integer*, etc., indicates the domain D over which the variable ranges.

We can describe *sets* of states and transitions by first-order formulas essentially as usual (cf. [15, 6]). The only difference in our case, is the fact that now our first-order formula representing the transition ($\mathcal{R}(V, V', T, A)$) has two extra parameters: T expresses whether the transition corresponds to a passage, either from point to interval phases, or from interval to point phases, or is a normal transition. A is a proposition that tells us which agent is being executed in this transition. A will be used when we will describe the transformation into linear hybrid automaton.

The main idea of this modelling is that a state of our graph (structure) is a set of constraints that define the value of variables and its evolution over time if are real-valued variables, and a set of *labels* that represent the point of execution of the system. The labels are introduced in the original program and can be active or disabled, depending upon the store of the system at each time instant. If the store allows to execute the operation associated with a specific label, then this label is active, otherwise it is disabled. All labels representing a temporal or preemption operation are disabled while there exists a *normal* label[1] active in the whole system (perhaps in another state). We take advantage of the declarative nature of the language in order to construct a finite-state representation.

Definition 1.1 *Let C be the set of constraints in the* `hybrid cc` *syntax and L be the set of all possible labels generated to label the original specification of the system. We define the set of states as $S \subseteq 2^{C \cup L}$*

Now we can define formally our graph (structure) capable of representing the system behavior.

Definition 1.2 (Hybrid cc Structure) *Let AP be a set of atomic propositions. An* Hybrid cc Structure *M over AP is a 6-tuple $M = (S, S_0, T, A, R, L)$, where*

1. *S is a finite set of states.*
2. *$S_0 \subseteq S$ is the set of initial states.*
3. *$T = \{n, p, i\}$ is the set of possible* type *of transitions. n denotes a normal transition while p denotes a transition from a point to the interval phase and i denotes a transition from an interval to a point phase.*
4. *A is a set of propositions representing each kind of agent.*
5. *$R \subseteq S \times S \times T \times A$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ and an agent A such that $R(s, s', n, A)$ or $R(s, s', p, A)$ or $R(s, s', i, A)$.*
6. *$L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

A *path* in M from the state s is defined as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1}, X, Y)$ holds for all $i \geq 0$. A transition where the third component is n (respectively p and i) is called *n-arc* (respectively *p-arc* and *i-arc*).

[1] We call normal labels those labels that do not cause extension over time

1.4 Labelling

The notion of label allows us to represent the point of execution in a state. We introduce this information by translating the original specification into a labelled version. This transformation consists in introducing a different label associated to each occurrence of a constructor in the specification.

Below we show the details of this transformation. Let P be a statement, the labelled version P_l of it is defined as follows:

- if $P = a$ then $P_l = \{1\}\ a$.
- if $P =$ `if` a `then` A then $P_l = \{1\}$ `if` a `then` A_l.
- if $P =$ `if` a `else` A then $P_l = \{1\}$ `if` a `else` A_l.
- if $P = (A, B)$ then $P_l = \{l\}(A_l, B_l)$.
- if $P =$ `new` X `in` A then $P_l = \{1\}$ `new` X `in` A_l.
- if $P =$ `hence` A then $P_l = \{1\}$ `hence` A_l.

In Figure 3 we show the labelled version of the program showed in Figure 2.

$$
\begin{array}{ll}
l_0(& l_1\ X = 0, \\
 & l_2 \text{ hence } l_4(\text{if } prev(X = 1) \text{ then } l_5\ X = 0), \\
 & l_3 \text{ hence } l_6(\text{if } prev(X = 1) \text{ else } l_7\ dot(X) = 1))
\end{array}
$$

Figure 3: Example: `hybrid cc` program

1.5 Graph construction

Here we describe how to derive an Hybrid cc Structure $M = (S, S_0, T, A, R, L)$ from the `hybrid cc` program specification of the system.

We define the set of states as the set of all combinations of constraints that appear in the system and we construct them while analyzing the specification. In each state a collection of constraints that represent the possible values of the variables in that time instant will appear.

In this section we show the construction that will return a graph representation of a Hybrid cc Structure associated to a given `hybrid cc` specification. Such a graph can simply be seen as a pictorial counterpart of a Hybrid cc Structure, with nodes representing the states and arcs representing the transition. To render the fact that we have three kind of transitions (depending upon the value of the third parameter in R) we will use three kind of arcs.

We start from an initial node constructed by assigning to its labelling function the set of labels corresponding to those instructions that can be executed in the first time instant. We define a series of actions associated with each operator defined in the language syntax that will produce the graph while analyzing the specification. We have a function `revision` that obtains the set of active labels for a given store.

In each state we represent the point of execution of the program by providing the labels of the instructions that can be executed in the next step. Each label corresponds to a different parallel process and can be active (which means that the conditions to execute the operator associated to this label are satisfied) or disabled (which means that the operator represented by this label cannot be executed in that moment because the store does not entail the necessary conditions).

A `hybrid cc` computation is a sequence of alternated point and interval phases. Intuitively, in each point phase, a Default cc program is executed and correspondingly we calculate the store of the system in that specific time instant. In such instantaneous computation, all agents which do not contain a preemption operator (negative ask) must be executed. Then, all preemption agents are executed, until a quiescent point is reached, and finally the temporal agents which are in the store are executed.

The execution of the temporal operators at the end of the point phase corresponds to the passage from the point to the interval phase. In this phase, the store is updated executing the current Default cc program and, at the same time, it is computed the minimum time interval in which the guard conditions of ask agents that we have to execute do not change. Once we have analyzed all these agents we obtain a time interval and then we pass to the next point phase by following each possible path consistent with the value of the discrete variables (which can change at any time instant, since they may be modified by other processes). In the new nodes generated we update the time and set the value of variables in that instant as the limit value of such variables in the previous interval phase.

Now, we use the time interval specified by the user in order to impose a constraint on the global execution time of our program. This constraint will make the graph construction finite.

We distinguish two cases.

First, if for the current instant we find a quiescence point at the end of the point phase that had already appeared in a previous time instant, and which is represented in the graph by another node n, we must connect the current node to the same nodes to which n is connected and the construction terminates. Note that in this case we get quite a strong result. In fact, since we did not cut the construction of the graph because we reached the time limit specified by the user then our graph is complete, i.e. it represents any run of the system.

Otherwise, if the current quiescent point had not appeared already in the graph we must continue the construction and we must build a new node which will be connected by a p-arc to the current one and which will contains the store and the execution point relative to the initial state of the next interval phase.

In the case that we reach the time limit provided by the user, the construction is terminated and the model obtained is valid only for the actual verification. It cannot be reused for different verifications.

It is easy, given a specification and an actual agent, to decide which is the following agent in the specification. We use the function `follow`[2] for this purpose. Moreover, the function `revision`[2] is defined in order to determine which

[2]The referee can find the formal definition in the appendix

labels are active and which ones are disabled.

Next we show the actions performed when we analyze an occurrence of each operator. We call *source* node the node from which a specific agent is executed.

- **Tell**. This operator adds some information to the store in the current time instant. In our graph construction it is translated as a new node related with the node from which it is executed. In this new node we add to the information stored in the previous node, the new constraint expressed with the tell operator. Then we execute `follow` and `revision`.
- **Positive Ask**. This operator verifies if the guard is satisfied in the current store and if it is satisfied, then the program execution can continue with the body specification. In our graph representation we construct a new node where the previous constraints over variables continue to hold and the condition of the positive ask is added. Note that in the case that by adding such constraint to the store it becomes inconsistent then the execution of the program will terminates. An edge that loops in the source node is added to the graph in order to model the case when the store does not entail the condition of the Positive Ask. Finally the body labels are added by executing the two functions `follow` and `revision`.
- **Negative Ask**. This operator (`if` a `else` A) verifies that the condition specified in the guard a is not satisfied in the present store. In that case it executes the actions specified in the body. We have two possible situations and we have to model both in the graph. In the first case, we have that if a is entailed by the store in s, then in the graph construction the only performed action is to construct a new node, add the condition a to the label of such node and remove from the set of agents to be executed the Negative Ask operation. The second case represents the case in which the store and default information allows the execution of the body of the agent. Then another new node is constructed and the label representing the negative ask's body is added to the set of labels in the state by the function `follow`. Finally, we apply the `revision` procedure.
- **Hiding**. This operator represents the existential quantification of the variable. In the graph representation it is expressed as a new node where the store does not change but in the labelling function the labels are updated. The information of the renaming of variables is introduced in an auxiliar structure where we put the agents with the quantified variables renamed apart. Then the functions `follow` and `revision` are applied.
- **Parallel**. The Parallel operator represents the concurrent behavior of the language. It generates different branches of execution. In our graph construction we introduce a new node in which the labelling function has all labels of those execution branches defined by the parallel agent. It is important to note that for one node there will be as many direct descendants as activated labels contained in it because of the notion of *interleaving* which corresponds to the language semantics. Then the functions `follow` and `revision` are applied.

- **Hence**. This operator executes at each time instant after now the agent specified in the body. The label associated to this operator will be disabled until we reach the quiescence point. At this point we construct a new node which is connected to the current one by a p-arc and whose store and label function are computed according to the operational semantics of `hybrid cc`. Finally, `follow` and `revision` are applied.

The i-arcs are introduced when we make the passage from an interval to a point phase. When we finish the analysis, we obtain a graph structured as a sequence of point and interval phases.

The following theorem proves the correctness of our graph construction.

Theorem 1.3 *Let T be the Hybrid cc Structure constructed by the above method from the* `hybrid cc` *specification S. Then the construction T is correct since*

$$\delta(T) \subseteq [\![S]\!]$$

where δ is the function that represents the traces in T given by the sequences, starting from the root, of program stores in each `hybrid cc` *node in a path and $[\![S]\!]$ represents the operational semantic of the* `hybrid cc` *specification S in [12].*

In Figure 4, the reader can find the graph construction relative to the program in figure 3 (cf. Section 1.4). It is possible transform a Hybrid cc Structure into a linear hybrid automaton as we show in next section.

2 Hybrid automata

There exist already several Model Checkers which implement verification techniques for hybrid systems (see for example in [13, 14] the Model Checker called HyTech). Here we review the definition of hybrid system and we define a transformation from an Hybrid cc Structure to a linear hybrid automaton.

An hybrid system is a discrete program within an analog environment that can be modelled by hybrid automata that are generalized finite-state machines. In addition, hybrid systems allow us to represent that the global state of a system changes continuously with the time.

Works on verification of hybrid systems are usually limited to *linear* hybrid automata (see [2]). In such automata, for each variable the rate of change with time is constant and the terms involved in the invariants, guards and assignments are required to be linear.

Now we will show the definition of a linear hybrid automaton (see [14] for details). A *linear expression* over a set X of real-valued variables is a linear combination of variables from X with rational coefficients. A *linear inequality* over X is an inequality between linear expressions over X. A *convex predicate* over X is a conjunction of linear inequalities over X; and a *linear predicate* is a disjunction of convex predicates.

Definition 2.1 *[14]*
A linear hybrid automaton *A consists of the following components.*

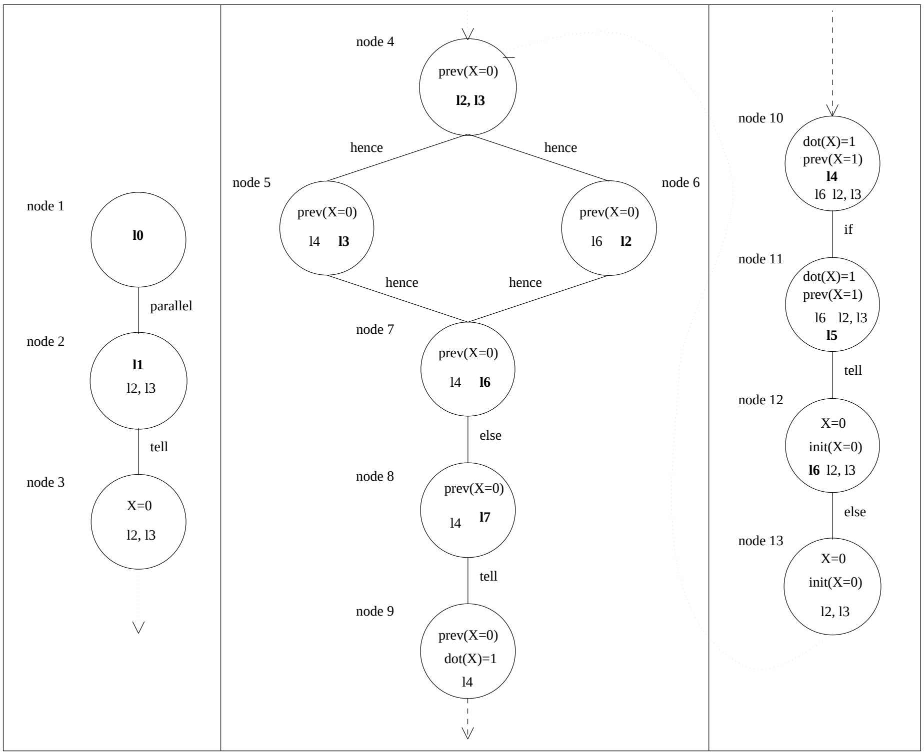

Figure 4: Example: Hybrid cc Structure

- **Variables**. *A finite ordered set* $X = \{x_1, \ldots, x_n\}$ *of real-valued* variables.
- **Locations**. *A finite set* V *of vertices called* locations, *used to model control modes.*
- **Initial condition** *A state predicate* ϕ^0 *called the* initial condition.
- **Location invariants**. *A labelling function* inv *that assigns to each location* $v \in V$ *a convex predicate* $inv(v)$ *over* X, *the* invariant *of* v.
- **Transitions**. *A finite multiset* E *of edges called* transitions, *used to model discrete events. Each transition* (v, v') *identifies a source location* $v \in V$ *and a target location* $v' \in V$.
- **Instantaneous actions**. *A labelling function jump that assigns an* update set *and a* jump condition *to each transition* $e \in E$. *The update set* $upd(e)$ *is a subset of* X. *The jump condition* $jump(e)$ *is a convex predicate over* $X \cup Y'$, *where* $Y = \{y_1, \ldots, y_k\} = upd(e)$, *and* $Y' = \{y'_1, \ldots, y'_k\}$. *The primed variable represents the variable value after the transition.*
- **Urgency flags**. *a partial labelling function asap that assigns the* urgency flag *ASAP to some transitions in* E.

- **Continuous activities**. *A labelling function rate that assigns a* rate condition *to each location $v \in V$. The rate condition $rate(v)$ is a convex predicate over $\dot{X} = \{\dot{x}_1, \dot{x}_2, \ldots, \dot{x}_n\}$. The variable $\dot{x}_i$ denotes the rate of change (the first derivative) of x_i.*
- **Synchronization labels**. *A finite set L of* synchronization labels*, and a labelling function syn that assigns a synchronization label from $L \cup \{\tau_A\}$ to each transition in E. The* internal label τ_A *is specific to the automaton A, and does not occur in the label set of any other automaton.*

A *state* (v, s) of the automaton A consists of a location $v \in V$ and a *valuation* of variables $s \in R^n$. The state (v, s) is *admissible* if $s \in inv(v)$. Control of A may reside in location v only while $inv(v)$ is satisfied. Only variables in $upd(e)$ are updated by a transition e.

3 Transformation

Being able to model the behavior of a concurrent system specified by an `hybrid cc` program via an Hybrid cc Structure and being possible to apply model checking to a linear hybrid automaton, we now outline an algorithm transforming a (limited) Hybrid cc Structure into a linear hybrid automaton. Note that in this work we want to exploit a model checker (HyTech) which has been previously defined ([14]) which accepts only linear hybrid automata, therefore we must consider only linear hybrid systems as input.

Our first task is to limit the (linear) Hybrid cc Structure using the time interval provided by the user. This is done by assigning values to variables and time intervals, and unfolding possible cycles in the phase sequence. With this operation we obtain a representation with a finite number of interval and point phases.

The main idea in this transformation is to identify the interval phases of Hybrid cc Structure with locations in the linear hybrid automaton, and point phases with arcs connecting locations. Obviously, an arc associated with the point phase s_p will join the locations that are associated with the previous and next interval phases, respectively.

Let us now give some detail on how we define the components of the linear hybrid automaton. In each location we define the set of variables and derivatives of variables, according to the conditions obtained in the relative interval phase defining them. The invariant will be the negation of the guard conditions of the ask agents defined in such interval phase. In addition we add a constraint that imposes the maximum time interval that the program can satisfy in such location. This limit is defined by the interval calculated in the interval phase.

Each arc in the linear hybrid automaton corresponds to a point phase in the Hybrid cc Structure. As we said, an arc joins two locations that are those representing the previous and next interval phases of the respective point phase. The set of variables that will be updated and their new values are obtained from the store calculated in the point phase. In addition, the label associated to the arc is the label associated to the arc that goes from the point phase to the interval phase. The set of urgent labels is null, because this notion is not used in `hybrid cc`. Finally, we do not impose restrictions in the synchronization labels and assign a different one to each arc.

In Figure 5, the reader finds the linear hybrid automata obtained from the structure in figure 4 associated to the `hybrid cc` program in figure 3.

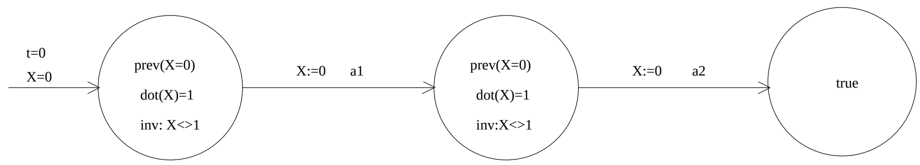

Figure 5: Example: linear hybrid automaton

4 Conclusions

In this work, building on the possibility represented (see [10, 12]) by `hybrid cc` language of using the Default cc paradigm together with a notion of continuous time, we extended to the realm of continuous computations the circle of ideas proposed in [9]. In particular, we defined an automatic transformation which takes an `hybrid cc` program as input and returns a representation of the system behavior as a graph structure. Then, an algorithm is outlined whose purpose is that of transforming such a graph structure into a linear hybrid automata using data from in order to limit the time interval in which the verification process will take place. The construction is based on the so-called *stability* assumption on the semantics of `hybrid cc` programs, guaranteeing a computation as a sequence of point and interval phases.

Previous work on this subject was presented in [11] where the authors do propose a translation technique turning an `hybrid cc` program into an hybrid automata with the aim of capturing executions of the program as traces of the automata. The main difference between the above mentioned approach and the one presented here, is the fact that we do not impose any particular format to the `hybrid cc` program in input (required to be *normal* in [11]). The use of a limited interval of time in the analysis and the consequent limit obtained to the number of states of the system, allows us to guarantee the possibility of using genuine finite-state (conventional) model-checkers, further characterizing our approach.

We are currently studying the applicability of our method to variations (see [8]) of the semantics of `hybrid cc` presented here. As a matter of fact, such variations—essentially based on the choices made on parameter passing methods in procedure calls—is considered for `tcc` as well. We plan to study a model checking algorithm for `hybrid cc` programs without the limitation to linear hybrid automaton. Note that in such case we could not use the HyTech model checker.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3-34, 1995.

[2] R. Alur, C. Courcoubetis, T.A. Henzinger, P.-H. Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. *Lecture Notes in Computer Science 736*, Springer-Verlag, 209-229, 1993.

[3] E. M. Clarke, E.A. Emerson, A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2),1986.

[4] B. Carlson, V. Gupta. The `hcc` Programmer's Manual. 1996, Available in http://www.parc.xerox.com/spl/projects/mbc/.

[5] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking. *Springer-Verlag Nato ASI Series F*, Vol 152, 1996.

[6] E.M. Clarke, O. Grumberg, D. Peled. Model Checking. *MIT Press*, 1999.

[7] E. M. Clarke, K. McMillan, S. Campos, V. Hartonans-GarmHausen Symbolic Model Checking. In Alur and Henziger editors. *Computer Aided Verification 96* volume 11(02), pages 419-422 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, July 1996. Springer Verlag.

[8] F. S. de Boer, M. Gabbrielli, and M. C. Meo. Semantics and expressive power of a Timed Concurrent Constraint Language. *Proceedings of Third International Conference on Principles and Practice of Constraint Programming*, CP 97., 1997

[9] M. Falaschi, A. Policriti, and A. Villanueva. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language Proc. of AGP'2000, 2000.

[10] V. Gupta, R. Jagadeesan, V.A. Saraswat, and D. Bobrow. Programming in hybrid constraint languages. In Panos Antsaklis, Wolf Kohn, Anil Nerode, and Sankar Sastry editors, *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[11] V. Gupta, R. Jagadeesan, and V.A. Saraswat. Hybrid cc, hybrid automata and program verification. In Alur, Henziger and Sontag editors. *Hybrid Systems III*, *Lecture Notes in Computer Science*, Springer Verlag, 1996.

[12] V. Gupta, R. Jagadeesan, and V Saraswat. Computing with continuous change. *Science of Computer Programming*, vol 30(1-2), pages 3-49, 1998.

[13] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel. Hybrid systems. In Grossman, Nerode, Ravn, and Rischel editors. *Lecture Notes in computer Science* 736, Springer Verlag, 1993.

[14] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: the next generation. in *Proceedings of the 16th Annual Real-time Systems Symposium*, pages 56-65. IEEE Computer Society Press, 1995.

[15] Z. Manna, A. Pnueli. Temporal Verification of Reactive Systems. Safety, *Springer-Verlag*, 1985.

[16] V. A. Saraswat, R. Jagadeesan, V. Gupta. Foundations of Timed Concurrent Constraint Programming. In Abramsky, S., editor, *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71-80. IEEE Computer Press, July 1994.

[17] V. A. Saraswat, R. Jagadeesan, V. Gupta. Programming in Timed Concurrent Constraint Programming. In Mayoh B. and Tyugu E. and Penjaam J. editor, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, pages 361-410. Springer Verlag, 1994.

[18] V. A. Saraswat, R. Jagadeesan, V. Gupta. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475-520, 1996

[19] H. B. Sipma, T. E. Uribe and Z. Manna. Deductive Model Checking. in *8th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 1102, pages 209-219, Springer-Verlag, 1996.

A Appendix

```
revision(S: set of agents, Store: set of constraints, S_act: set of agents, S_dis:
set of agents);
        e: agent;
        T: set of agents;
        while E ≠ φ do
            select(E,e);
            E = E\e;
            if ((e[2] != 'hence') then
               if entail(Store,e.label) then
                   S_act = S_act ∪ e
               else
                   S_dis = S_dis ∪ e ;
            else
               T = T ∪ e ;
        if empty(S_act) then
            S_act = T;
            S_dis = φ
        else
            S_dis = S_dis ∪ T;
```

Figure 6: Algorithm `revision`

```
struct agent:
        label: t_label;
        type: {tell,positive,negative,hiding,parallel,hence,null};
        B1: agent;
        B2: agent;

follow(S: agent, e: label, labs: set of labels )
        S1 :agent;
        b :boolean;
        localize(S, e, S1, b);
        case S1.type of
            tell : labs = {};
            positive : labs = {S.B1.label};
            negative : labs = {S.B1.label};
            hiding : labs = {S.B1.label};
            parallel : labs = {S.B1.label, S.B2.label};
            hence : labs = {S.label, S.B1.label};
        end case;

localize(S: agent, e: label, S': agent, found: boolean)
        S'1, S'2 :agent;
        found1, found2 :boolean;
       while S.label <> e ∧ S.type <> 'null' do
        if S.type <> 'parallel' then;
            localize(S.B1, e, S', found)
        else
            localize(S.B1, e, S'1, found1);
            if found1 = true then
               found = found1;
               S' = S'1;
            else
               localize(S.B2, e, S'2, found2);
               if found2 = true then
                   found = found2;
                   S' = S'2;
```

Figure 7: Algorithm `follow`