Electronic Notes in Theoretical Computer Science 48 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume48.html pp. 1 – 14

Modeling Concurrent systems specified in a Temporal Concurrent Constraint language-I

Moreno Falaschi [1] Alberto Policriti [2] Alicia Villanueva [3]

Dipartimento di Matematica e Informatica
Università di Udine
Udine, Italy

Abstract

In this paper we present an approach to model concurrent systems specified in a temporal concurrent constraint language. Our goal is to construct a framework in which it is possible to apply the *Model Checking* technique to programs specified in such language.

This work is the first step to the framework construction. We present a formalism to transform a specification into a `tcc` Structure. This structure is a graph representation of the program behavior.

Our basic tool is the *Timed Concurrent Constraint Programming* (`tcc`) framework defined by Saraswat *et al.* to describe reactive systems. With this language we take advantage of both the natural properties of the declarative paradigm and of the fact that the notion of time is built into the semantics of the programming language. In fact, on this ground 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 Model Checking algorithms to be applicable. The graph construction is a completely automatic process that takes as input the `tcc` specification.

Key words: Timed Concurrent Constraint programming, Reactive systems, Model checking

1 Introduction

Timed Concurrent Constraint Programming (see [6]) is an extension of the Concurrent Constraint Programming paradigm obtained adding the time con-

[1] Email: falaschi@dimi.uniud.it
[2] Email: policrit@dimi.uniud.it
[3] Email: alicia@dimi.uniud.it

cept to the CCP model. The fundamental contribution of the tcc model is to augment the ability of constraint programming to detect *positive information* with the ability to detect *negative information*. Such a negative information is crucial to model reactive and real-time computations [4]. tcc, in addition, incorporates the idea that once a negative information is detected it is too late to change the past. If some information has not been stored in that instant it will not be stored.

The method of Model Checking is an extremely successful approach to formal verification 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 [1,3,8,5].

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 final goal consists in describing an environment in which is possible to apply Model Checking to a system specified in the tcc language. To apply Model Checking with our approach it is necessary to perform four 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 calculate the variable domains in the time interval where the Model Checking will be executed (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,
- to adapt a standard automatic verification mechanism based on Model Checking to the output of our previous steps. This may involve human assistance for example to evaluate the results or localize the errors.

In this paper we discuss the first two 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 tcc 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) ccp programs, while the evolution of graphs corresponding to different time instants is forced to be finite by the restrictions we impose on variable domains and time intervals.

In Section 2 we introduce the programming language `tcc` including a short description of its semantics and its main features. In Section 3 we introduce the definition of `tcc` Structure and present the basic definitions that allow us to formalize the method. In Section 3.3 we describe how to perform the first step in modeling the system producing a (finite) graph representation of an (a collection of) infinite state model(s). Then, in Section 4 we (briefly) discuss the introduction of the restrictions based on time parameters and values provided by the user. Finally, we draw our conclusions and comment on the steps which are necessary to complete the definition of our framework.

2 Timed Concurrent Constraint Programming

Timed Concurrent Constraint Programming was developed as a simple model for determinate, timed, and reactive systems. Using this model 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.

The language supports hierarchical and modular construction of specifications (programs). Because it is a determinate language, construction and analysis of programs and specifications is easier.

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

(Agents)	A ::= c	– Tell
	| `now` c `then` A	– Timed Positive Ask
	| `now` c `else` A	– Timed Negative Ask
	| `next` A	– Unit Delay
	| `abort`	– Abort
	| `skip`	– Skip
	| $A \mid\mid A$	– Parallel composition
	| $X \hat{} A$	– Hiding
	| g	– Procedure call
(Procedure Calls)	g ::= $p(t_1, \ldots, t_n)$	
(Declarations)	D ::= $g :: A$	– Definition
	| $D.D$	– Conjunction
(Programs)	P ::= $\{D.A\}$	

Fig. 1. Syntax for `tcc` programs

There are two kind of constructs in this language: `CCP` (*Concurrent Constraint Programming*) constructs and *Timing* constructs. *Tell, Parallel Composition* and *Timed Positive Ask* are the `CCP` constructs inherited from `CCP`. These operators do not cause "extension over time". In particular, Parallel Composition is the explicit representation of concurrence in the language and Timed Positive Ask and Tell allow the processes synchronization and state

evolution.

The other class of constructs are Timing constructs: *Timed Negative Ask, Unit Delay,* and *Abortion* that cause extension over time. Unit delay forces a process to start in the next time instant. Timed Negative Ask is a conditional version of Unit Delay, based on detection of negative information: it causes a process to be started in the next time instant if on the quiescence of the current time instant, the store was not strong enough to entail some information.

The above operators do not allow us to pass information from one time instant to the following one. The *Parametric Ask* operator is introduced for this purpose. The syntax of this operator is defined as

$$X\$(c \rightarrow A)$$

where X is a set of variables, c is a condition and A an agent. Intuitively, this operator works by searching a substitution with domain X that makes syntactically identical some constraint a in the store and c. If there exists such substitution (say γ), then the agent $A[\gamma]$ [4] is executed. An additional condition on the parametric ask is that all variables that appear in the guard c and that are not in X must be instantiated. Otherwise the agent cannot be executed.

We define the operational semantics of this operator as

$$\frac{\exists\gamma/\exists a{\in}\sigma(\Gamma) \wedge c[\gamma]{\equiv}a}{(\Gamma,X\$(c{\rightarrow}B)){\longrightarrow}(\Gamma,B[\gamma])}$$

where Γ is a multiset of agents and $\sigma(\Gamma)$ is the store contained in the configuration Γ. We have defined this rule following the notation introduced in [7].

In [6,9] the definition of other possible operators with respect to the basic ones can be found. For example the `always` A, `now` c `then` A `else` B, *multiple prioritized waits*, `whenever` c `do` P, `do` P `watching` c, etc. are defined using the operators showed in Figure 1. We could always transform a complex program into a simpler one that uses only basic operators. This is an important remark because, in principle, we only have to construct the mechanism of modelization for the basic operators.

As a matter of fact, we will not proceed exactly in this way but we will leave some operator (e.g. the `always` A) non translated. This is done in order to obtain a finite representation of an infinite state model after the first modelization step.

For example, in Figure 2 we can see a program written in `tcc` with the "extra" operator `always` A and the parametric ask agent.

The above program calculates a serie of integers starting from $Init$. In the first time instant it produces the $Init$ value, in the next time instant it obtains the following one, and so on.

The first step necessary to replace the `always` A by basic operators can

[4] By $\sigma[\gamma]$ we denote the application of the substitution γ to the syntactic object σ

counter(Init,I) ::
    ({I:Init} ||
    always (X,X1)$((I:X, X1 is X + 1) ⟶ **next** {I:X1}))

Fig. 2. Example: Counter Program.

be seen in Figure 3. We show only the first transformation step because the complete transformation would produce an infinite specification.

counter(Init,I) ::
    ({I:Init} ||
        (X,X1)$((I:X, X1 is X + 1) → **next** {I:X1}) ||
        always (X,X1)$((I:X, X1 is X + 1) → **next** {I:X1}))

Fig. 3. Example: Counter Program transformed.

In the following we assume to use the `tcc` language enriched with the Parametric Ask and `always` operators. The operational semantics of the constructs in Figure 1 is defined in [7]. For giving the semantics for the `always` A operator we add to the transition system in [7] the following rule:

$$(\Gamma, \texttt{always}\ A) \xrightarrow{\texttt{always}A} (\Gamma,\ A,\ \texttt{next always}\ A)$$

Again we have followed the notation used in [7] in order to define this rule.

3 Modeling

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. Reactive systems cannot be modeled 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 `tcc` formalism introduced above and will eventually contain an explicit notion of time.

3.1 Basics

In order to start with our modeling task we use the classical notion of *Kripke Structure*. This kind of structures are able to capture the behavior of reactive systems and, as we will show below, it is possible to construct automatically one structure that represents the behavior of the system from the `tcc` specification of the program.

Variables. The set $\mathcal{V}ar$ represents the universal set of variables. $V \subseteq \mathcal{V}ar$ is the set of variables that appears 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. [5,2]). The only difference in our case, is the fact that now our first-order formula representing the transition ($\mathcal{R}(V, V', T)$) has an extra parameter T expressing whether the transition corresponds to a passage to the next time instant or not.

A state of our graph (structure) is a set of constraints, that define the value of 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 operation are disabled while there exists a *normal* label active in the whole system (perhaps in another state).

Definition 3.1 Let C be the set of constraints in the `tcc` 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 3.2 Let AP be a set of atomic propositions. A `tcc` *Structure* M over AP is a 5-tuple $M = (S, S_0, T, R, L)$, where

(i) S is a finite set of states.

(ii) $S_0 \subseteq S$ is the set of initial states.

(iii) $T = \{t, n\}$ is the set of possible *type* of transitions. t denotes a temporal transition while n denotes a normal transition.

(iv) $R \subseteq S \times S \times T$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s', t)$ or $R(s, s', n)$.

(v) $L : S \to 2^{AP}$ 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 $R(s_i, s_{i+1}, X)$ holds for all $i \geq 0$.

Now we show how to derive a `tcc` structure $M = (S, S_0, T, R, L)$ from the `tcc` program specification of the system.

We cannot apply directly the method of [2] to model the system behavior because they assume that the domains for the variables are finite and hence the number of possible valuations for the variables is finite. We define the set of states as the set of all combinations of restrictions 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 the following we define a construction that will return a graph representation of a `tcc` structure associate to a given `tcc` specification. Such a graph can simply be seen as a pictorial counterpart of a `tcc` structure, with nodes representing the states and arcs representing the transition. To render the fact that we have two kind of transitions (depending upon the value of the third parameter in R) we will use two kind of arcs. The quiescence points can be identified in the graph because are those that preceded a temporal transition.

3.2 Labeling

The notion of label allows to represent the point of execution in a state. We introduce this information by translating the original specification into a labeled version. This transformation consists in introducing a different label associated to each instance of a constructor in the specification. In Figure 4 we show the labeled version of a program that calculates a serie of integers. Our approach in labeling the program is similar to the approach used in [5,2]: we only adapt classical approaches to our specific operators. A simple algorithm translates the original program in the labeled one.

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

- if $P = c$ then $P_l = \{\mathrm{l}\}\ c$.
- if $P =$ `now` c `then` A then $P_l = \{\mathrm{l}\}$ `now` c `then` A_l.
- if $P =$ `now` c `else` A then $P_l = \{\mathrm{l}\}$ `now` c `else` A_l.
- if $P =$ `next` A then $P_l = \{\mathrm{l}\}$ `next` A_l.
- if $P =$ `abort` (`skip`) then $P_l = \{\mathrm{l}\}$ `abort` (`skip`)
- if $P = A \mid\mid B$ then $P_l = \{\mathrm{l}\}\ (A_l \mid\mid B_l)$.
- if $P = X\hat{}A$ then $P_l = \{\mathrm{l}\}\ X\hat{}A_l$.
- if $P = g$ and g is a procedure call then $P_l = \{l\}g$.
- if $P =$ `always` A then $P_l = \{\mathrm{l}\}$ `always` A_l.
- if $P = X\$(c \rightarrow A)$ then $P_l = \{\mathrm{l}\}\ X\$(c \rightarrow A_l)$.

The result of applying this transformation to the Counter example is showed in Figure 4.

{l0} counter(Init,I) ::
  {l1} ({l2} {I:Init} ||
  {l3} **always** {l4} (X,X1)$((I:X, X1 is X + 1) → {l5} **next** {l6} {I:X1})).

Fig. 4. Example: Counter Program labeled

3.3 Graph construction

In this section we explain how we construct the `tcc` structure in an automatic way. A program is composed of a set of clauses and a goal. We start the construction from an initial node labeled with the label representing the goal of the program. Then we repeat a sequence of steps since the construction is completed. We define a series of actions associated with each operator defined in the language syntax that will produce the graph while analyzing the specification.

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).

It is important to notice that a temporal operation (`next`, `now` c `else` P, etc.) cannot be executed before all the "normal" operations are executed. This is motivated by the fact that we have to arrive to the quiescence state (where no more information can be produced in the present time instant) before moving to the subsequent instant of time. Only in quiescence we will be sure that we have all the information necessary in the next time instant.

Roughly, our procedure consists of the repetition of the following steps (while there exists an active label):

- to localize the agent A that will be executed (let l be the label associated to it),
- to perform the actions associated with such agent,
- to determine the set of agents (and its labels) that follows A in the specification,
- to add the new labels obtained in the previous step to the set of labels and remove the label l,
- to revise the set of labels defined in the previous step setting each one as active or disabled depending on the store

When we reach a state s where there is no active label, then we are in a quiescent point. Then we have to pass to the next time instant. In order to make this passage we have to analyze the labels in the labeling function of the quiescent state. First of all we analyze the Next agents, then the Negative Asks and finally the Always operators. In the graph construction it means that we create a new node t related with s by a temporal transition. In such node we introduce the labels associated to the body of the temporal agents present in the quiescence point. The store is set to the empty store and the execution proceeds as explained in the sketch of the procedure. Note the special case of the temporal labels associated to the always agents. For such labels we have

to make an extra verification. We search in the graph a previous quiescent node equal to t and, if we find it, then we relate t to such node and finish the construction. If there is no equivalent node, then we introduce in the set of labels of the node t the label associated to the always operator.

The existence of this cyclic behavior of the system is not guaranteed, it depends on the system itself. Thus, in order to ensure the finiteness of the graph construction we use the concept of (finite) time interval. This is the interval over which the verification will be executed, then we take an upper bound of such interval and, if we reaching such time instant whereas the construction is being performed, then we stop and obtain a model that is valid only for this specific verification. Otherwise, if we finish the construction before reach such time instant, we construct a generic model valid for whatever verification of the system. We describe now the construction of the graph for each syntactic construct.

- **Tell**. `tell` c adds some information to the store in the current time instant. In our graph construction it is translated as a new node s' related with the node s from which it is executed. We define the labeling function $L(s')$ as follows: the part representing the store is defined as the store of s plus the constraint c; the part representing the new point of execution is calculated removing the label representing the tell operator and obtaining the labels for the following agents in the specification. Finally we revise the active and disabled labels.
- **Positive Ask**. `if` c `then` P verifies if the guard is satisfied in the present 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 part representing the store of the labeling function is defined as the union of the store in the previous state and the constraint specified in the guard of the agent. Note that it is necessary to introduce the information of the guard because the variables present in the condition could be parameters of a procedure. Then, the part of the labeling function representing the new execution point is calculated as in the previous case.
- **Skip**. `skip` does no operation, which means that in the graph representation we will create a new node s and will define the labeling function $L(s)$ as the labeling function of the predecessor by removing the label representing the skip agent and performing the revision of active and disabled labels.
- **Parallel**. $(A \mid\mid B)$ represents a concurrent agent where A and B are executed in parallel. It will generate different branches of execution. In our graph construction we introduce a new node related with the previous one that inherits from it the part of the labeling function representing the store. The part of the labeling function representing the new execution point will be calculated as it was explained in the previous cases. It is important to remark that for one node there will be as many direct descendants as activated labels contained in it.

- **Hiding**. This operator represents the concealment of the variable to the rest of the system. It will be useful in the subprogram associated with the Hiding operator. In the graph representation it is expressed as a new node where the variables in the hiding operator are renamed apart. We apply the usual method to calculate the part of the labeling function that represents the point of execution.
- **Procedure Call**. This operator refers to another procedure in the specification. All procedures will have different labels and variables. In our graph construction it is created a new node where the substitution affecting the parameters of the procedure p is applied and the label of the first agent of p is introduced. Finally we revise the active and disabled labels.
- **Parametric Ask**. $X\$(c \rightarrow A)$ allows to pass some information from one time instant to the following one. If all variables in the condition c that are not included in X are instantiated, then it is calculated the substitution γ that unifies c and some constraint in the store and then the agent $A[\gamma]$ is executed. In our graph construction, a label representing an agent of this kind will be active only when the condition about the instantiation of variables is satisfied. Then we will introduce a new node related with the previous one with a normal transition and whose labeling function is calculated as follows: first of all the substitution Γ is calculated; then a new agent is created as the result of the application of the substitution Γ to the body A of the Parametric Ask. New labels are associated to such agent and introduced in the labeling function of the node. The labeling function is completed with the same items of the labeling function of the previous node except for the label representing the parametric ask executed. As in previous cases it is checked if labels are active or disabled.
- **Always**. `always` A allows us to model a cycle in the system behavior. In the graph construction we proceed by creating a new node s where the part of its labeling function representing the store is inherited from the ancestor node and the part of the execution point is calculated using our two procedures `follow` and `revision`. These procedures will produce a temporal label associated to the always operator (again) and the label associated to the body of the `always` A agent.

Note that we do not introduce any action for the temporal agents because all actions that those agents will cause will be executed when we reach a quiescent point. The labels representing its body will be introduced in the first node of the following time instant.

The following theorem proves the correctness of our graph construction.

Theorem 3.3 *Let T be the* `tcc` *structure constructed by the above method from the* `tcc` *specification S. Then the construction T is correct since*

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

where δ is the function that represents the traces given by the sequences, star-

ting from the root, of program stores in each `tcc` *node in a path of* T *and* $[\![S]\!]$ *represents the operational semantic of the* `tcc` *specification* S *in [7] enriched by the rules discussed in Section 1.*

3.4 An illustrative example

To illustrate the actions explained in the previous section, we show a `tcc` structure and graph construction for a programming example. The system specified computes a sequence of integers (see Figure 4). We add the goal $\{lg\}$ `counter`$(1, \texttt{Sal})$ to this specification and we impose the time interval $[1, 3]$.

In Figure 5 rectangular boxes show the piece of code which is generated by each execution of the parametric ask, which is labeled by $l4$.

4 Graph Restrictions

Up to this point we have introduced a method to construct a `tcc` structure starting from the specification of the system. We have obtained a finite graph but that cannot be handled by classical algorithms of Model Checking: it is in fact an abstract representation for an infinite collection of models.

The reason for which this structure cannot be used with classical approaches is that we have not restricted the variable domains to be finite. Moreover, it is necessary to have all variable values in each state, whereas in our graph representation there are variables that do not have a specific assigned value (for example, because it depends on the value of other variables). The solution to this problem is to introduce some kind of restriction to the representation forcing the variable domains to be finite.

The main idea is that we do this transformation taking into consideration information from the user that will be required to specify in which interval of time she wants to do the verification and what are the initial values of the variables. Then we analyze the program and give an interval of values on the variable domain over which this variable can be in that time interval. Those values are a limit for the possible values that are calculated by considering the structure of the program clauses. We consider all variable modifications and taking into consideration all increment and decrement actions and data dependencies we obtain the maximum and minimum values. Of course, perhaps those values are not necessarily reached. Finite domains are treated in a different manner because it is not necessary to restrict them.

Definition 4.1 [Domain Restriction]

Let $V \in Var$ the set of variables of the program P and $V_{inf} \in V$ the set of variables with an infinite domain. Let $t_I, t_F \in N$, $t_I, t_F \geq 0$ an initial and final time instant $t_I \leq t_F$, then we define:

- $val_t(V)$ as the set of variable values at the instant of time t.

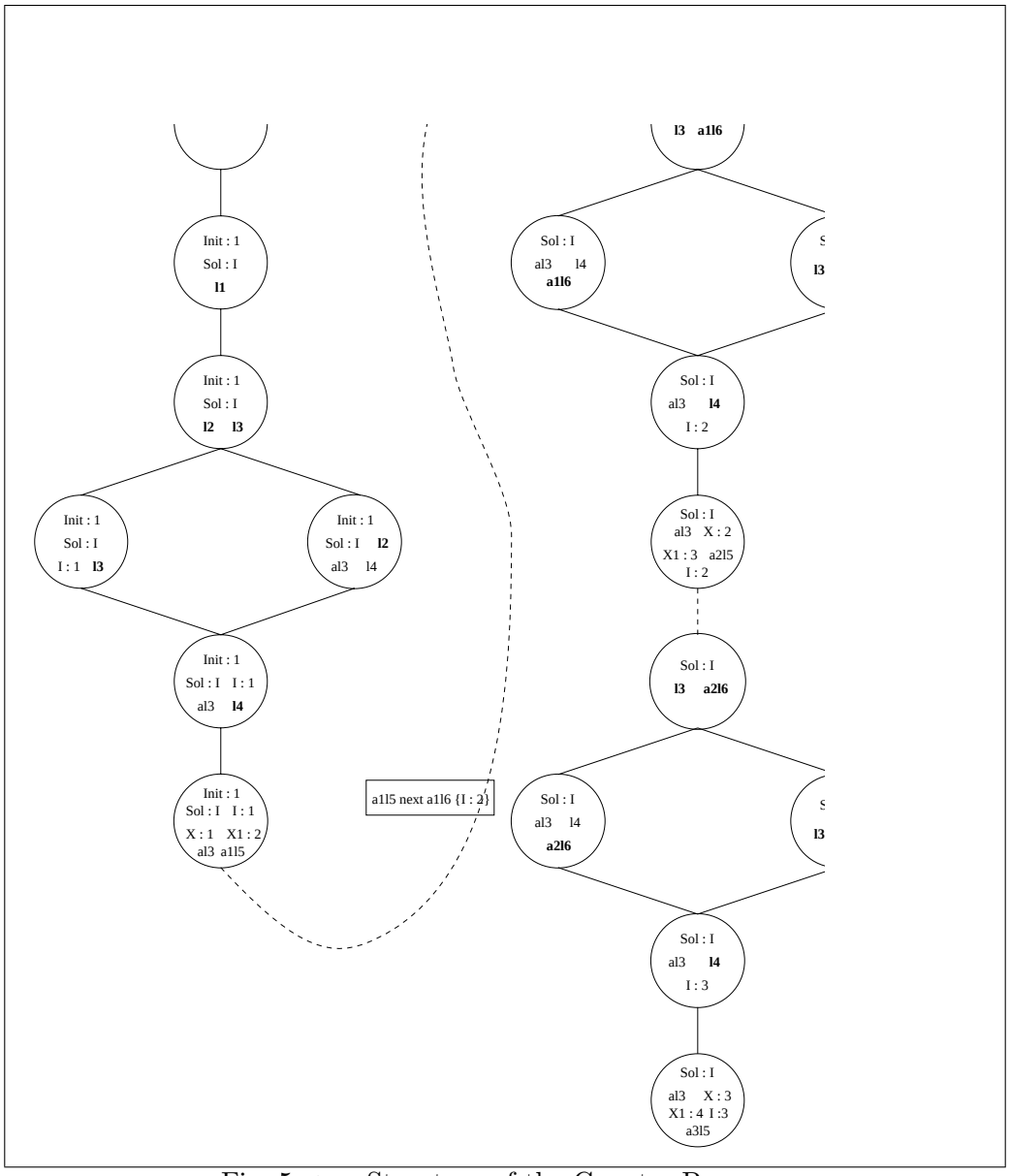

Fig. 5. `tcc` Structure of the Counter Program

- $val_{t_I}(V)$ as the set of initial variable values defined by the user.
- $val_{max}(V)$ as the set of the maximum values that variables can reach.
- $val_{min}(V)$ as the set of the minimum values that variables can reach.
- $analyze(S, val_{t_I}(V), t_I, t_F, val_{min}, val_{max})$ as the function that calculates the $val_{max}(V)$ and $val_{min}(V)$ in such a time interval where S is the `tcc` Structure obtained from the specification P

With these new values we will transform the graph representation obtaining a finite representation of the system behavior during the time interval introduced by the user and starting by the initial values of the variables defined by her.

The main difference between the classical approach and our method is that in classical approaches the restriction over the variable domains is introduced in the definition of the system while in our framework we can specify a system with infinite variable domains and construct the corresponding graph structure. Then we have a *basic* structure and we only have to make the graph restriction every time the user wants to make an execution of the Model Checker and introduces the necessary information.

5 Conclusions

We have defined an automatic transformation which takes a `tcc` program as input and returns a representation of the system behavior as a graph structure.

Classical Model Checking approaches cannot be applied to the graph structure constructed in this work because there exist some differences between the Kripke Structure constructed in classical works and our `tcc` Structure. The main difference is that we introduce two kinds of state transitions. In classical approaches transitions between states represent the increase of time while in our framework *normal* transitions change state in the same time instant. There is an increment of time only when a *timed* transition is executed. This introduces a synchronization primitive because a process that would execute a timed operation has to wait until all other processes have finished their normal operations. Classical approaches are asynchronous because they assume that all parallel processes are independent from each other, thus they can be executed without waiting for any information calculated by the other processes.

We ensure the finiteness of our construction by the following facts. In each time instant our language is similar to (terminating) determinate CCP, so each graph is finite. Moreover, for different time instants, we ensure finiteness by imposing a restriction on the variable domains, by considering time intervals. Note a similarity with the work of Clarke *et al.* [2] in the fact that they introduce some synchronization when they impose that all processes of a parallel operation may finish at the same time instant.

The main contribution of this work is the introduction of a method to verify properties of reactive systems specified in `tcc`. This language is a declarative language but has the notion of time built-in its semantics and this is the reason why we can define a Model Checker for a given time interval provided by the user. The resulting methodology is quite powerful since it exploits the fact of using a constraint language for programming, and hence ensuring finiteness, does not require to impose severe limitations on the expressivity of the language. The second part of this project will deal with representing the property that the user wants to verify. We plan to carry out the formalization at the ground of this step in a way similar to Clark *et al.* [2].

Subsequently our goal will be to process structures such as the ones described in this paper, in order to produce standard (equivalent) finite Kripke

Structures to be given as input to Model Checkers. The main problems to be solved to this end are the (finite) unfolding of cycles and the reduction of timed transitions to untimed (standard) ones.

References

[1] Clarke, E., O. Grumberg and D. E. Long, *Model checking*, Springer-Verlag Nato ASI Series F **152** (1996).

[2] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[3] Clarke, E. M., E. A. Emerson and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, toplas **8** (1986), pp. 244–263.

[4] Henzinger, T., Z. Manna and A. Pnueli, *Timed transition systems*, in: J. de Bakker, K. Huizing, W.-P. de Roever and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science **600**, Springer-Verlag, 1992 pp. 226–251.

[5] Manna, Z. and A. Pnueli, "Temporal Verification of Reactive Systems. Safety," Springer Verlag, 1995.

[6] Saraswat, V. A., R. Jagadeesan and V. Gupta, *Foundations of timed concurrent constraint programming*, in: S. Abramsky, editor, *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science* (1994), pp. 71–80.

[7] Saraswat, V. A., R. Jagadeesan and V. Gupta, *Programming in timed concurrent constraint programming*, in: B. Mayoh, E. Tyugu and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia, Springer Verlag*, 1994, pp. 361–410.

[8] Sipma, H. B., T. E. Uribe and Z. Manna, *Deductive model checking*, in: R. Alur and T. Henzinger, editors, *Computer-Aided Verification, 8th International Conference (CAV'96)*, Lecture Notes in Computer Science **1102**, Springer-Verlag, 1996 pp. 209–219.

[9] V.A.Saraswat, R.Jagadeesan and V.Gupta, *Timed default concurrent constraint programming*, Journal of Symbolic Computation **22** (1996), pp. 475–520, extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.