

# Modeling Concurrent systems specified in a Temporal Concurrent Constraint language

*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. 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.

**Keywords:** Timed Concurrent Constraint programming, Reactive systems, Model checking

# 1 Introduction

Timed Concurrent Constraint Programming (see [9]) is an extension of the Concurrent Constraint Programming paradigm obtained adding the time concept 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. `tcc`, in addition, incorporates the idea that once a negative information is detected it is too late to change the pass. 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.

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 also finite because of 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`. For technical reasons we have modified some minor details.

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 concurrency 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.

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

Figure 1: Syntax for tcc programs

In [9, 11] the definition of other possible operators with respect to the basic ones can be found. For example the **always**, **now  $c$  then  $A$  else  $B$** , *multiple prioritized waits*, **whenever**, **do  $P$  watching** 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**) 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**.

```
fib(Val) ::= Val:0 || next Val:1 ||
           always X^(now Val:X then
                    next (Y,Z)^(now Val:Y then skip ||
                               (Z = X + Y) || next Val:Z)).
```

Figure 2: Example: Fibonacci Program.

The above program calculates the Fibonacci series. In the first time instant it produces the first value, in the next time instant it obtains the following one in the series, and so on. The first step necessary to replace the **always** by basic operators can be seen in Figure 3. We show only the first transformation step because the complete transformation would produce an infinite specification.

In the following we assume to use the tcc language enriched with the **always**

```

fib(Val) :: Val:0 || next Val:1 ||
           X^(now Val:X then
              next(Y,Z)^(now Val:Y then skip ||
                          (Z = X + Y) || next Val:Z)) ||
           next always (X^(now Val:X then
                          next (Y,Z)^(now Val:Y then skip ||
                                    (Z = X + Y) ||
                                    next Val:Z))).

```

Figure 3: Example: Fibonacci Program transformed.

operator. The operational semantics of the constructs in Figure 1 is defined in [10]. For giving the semantics for the `always` operator we add to the transition system in [10] the following rule:

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

We have defined this rule following the notation used in [10]

### 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  $\mathcal{Var}$  are typed, where the type of a variable, such as *boolean*, *integer*, etc., indicates the domain  $D$  over which the variable ranges. Following Manna and Pnueli (see [7]), in order to model the temporal behavior we have to introduce the set of *primed versions* of each variable. In particular, we assume that, for each  $x \in \mathcal{Var}$ , its primed version  $x'$  is also in  $\mathcal{Var}$ . Those primed versions represent the *value of the unprimed variable in the next time instant*.

We can describe *sets* of states and transitions by first-order formulas essentially as usual (cf. [7, 3]). 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  $\tau cc$  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 3.2 ( $\tau cc$  Structure)** *Let  $AP$  be a set of atomic propositions. A  $\tau cc$  Structure  $M$  over  $AP$  is a 5-tuple  $M = (S, S_0, T, R, L)$ , where*

1.  $S$  is a finite set of states.
2.  $S_0 \subseteq S$  is the set of initial states.
3.  $T = \{t, n\}$  is the set of possible type of transitions.  $t$  denotes a temporal transition while  $n$  denotes a normal transition.
4.  $R \subseteq S \times S \times T$  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  $R(s, s', n)$ .
5.  $L : S \rightarrow 2^{AP}$  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  $\pi = s_0 s_1 s_2 \dots$  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 [3] 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. Moreover, we will have special nodes (that is, those associated to the `always` operator) that correspond to points where an infinite unfolding would be necessary if we translated every operator in the pure basic language.

## 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 the maximum number between two values given by the user. Our approach in labeling the program is similar to the approach used in [7, 3]: 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  $P_l = \{1\} c$ .
- if  $P = \text{now } c \text{ then } A$  then  $P_l = \text{now } c \text{ then } \{1\} A_l$ .
- if  $P = \text{now } c \text{ else } A$  then  $P_l = \text{now } c \text{ else } \{1\} A_l$ .
- if  $P = \text{next } A$  then  $P_l = \text{next } A_l$ .

- if  $P = \text{abort}(\text{skip})$  then  $P_l = P$
- if  $P = A \parallel B$  then  $P_l = A_l \parallel B_l$ .
- if  $P = X \hat{A}$  then  $P_l = X \hat{A}_l$ .
- if  $P = g$  and  $g$  is a procedure call then  $P_l = g$ .
- if  $P = \text{always } A$  then  $P_l = \{1\} \text{ always } A_l$ .

The result of applying this transformation to the “maximum” example is showed in Figure 4.

$$\{10\} \text{ max}(X, Y, Z) :: \{11\} \text{ now } (X > Y) \text{ then } \{12\} Z : X \parallel \\ \{13\} \text{ now } (X > Y) \text{ else } \{14\} Z : Y.$$

Figure 4: Example: Maximum number program labeled

### 3.3 Graph construction

In this section we explain how we construct the `tcc` structure in an automatic way. We start from an initial node constructed by assigning to its labeling function all constraints referring to initial parameters of the program and 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.

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 disabled label can be activated in the same time instant but in a subsequent state, when the store entails all the necessary conditions.

It is important to notice that a temporal operation (`next`, `now-else`, 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.

- **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 been executed. In this new node we add to the information stored in the previous node, the new constraint expressed with the tell operator. We then analyze the next operator in the specification and introduce in the node labeling its label or set of labels. When we introduce a label in a node, we have to analyze it to decide if it must be active or disabled.

- **Positive Ask.** This operator 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 previous constraints over variables continue to hold. In addition it adds the body's labels with the analysis explained above.
- **Negative Ask.** This operator verifies that the condition specified in the guard is not satisfied in the present state. In that case it executes in the *next* time instant the actions specified in the body. In building the graph representation we act as in the case of Positive Ask but the information added to the node labeling is the negation of the guard and the nodes associated to the body are related with the current one by a time transition. The fact that the body is executed in the next time instant is guaranteed by the fact that a label associated to a time operator is disabled if we can execute some normal action (and this is the case for labels associated to a Negative Ask's bodies). The introduction of the negation of the guard is performed because we need to update the meaning of the node to represent always the conditions that are satisfied in each state.
- **Next.** This operator represents the increment of one time unit, i.e., the pass from the actual time instant to the next one. In the graph it is represented by a particular transition. No time transition is taken if one of the *non temporal* actions can be executed. This causes that in a specific step of the construction all labels of all states correspond to *temporal* operators (Negative Ask, Next or Abortion). In such a situation the time instant is increased and the variables renamed to avoid confusion. In the graph construction the transition associated to the Next operation is represented with a special arc just to indicate that there has been a change of the transition interpretation and variables have been renamed. Such a special arc corresponds to the value  $t$  for the third parameter of the associated transition relation in the `tcc` structure.
- **Abort.** The meaning of the Abort action is the end of the execution. If this operator is executed, then the branch of the graph is closed: there will be no descendants from that node.

- **Skip.** Skip does no operation, which means that in the graph representation we will have only that the label associated to the Skip operator disappears. The process continues with the same analysis for labels to be introduced, as in the previous cases.
- **Parallel.** The Parallel operator represents the concurrence behavior of the language. It generates different branches of execution. In our graph construction we introduce in the labeling function all labels of those execution branches taking into consideration all possible restrictions for the active or disabled label state. It is important to note 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 to the Hiding operator. In the graph representation it is expressed as a new node with all previous constraints and this variable renamed. As for the labels, we do the typical action of addition of the Next operators.
- **Procedure Call.** This operator simply connects to another graph structure built using the previous rules.
- **Always.** The node associated to the Always operator is the special kind of node to which we referred commenting on the pictorial representation of `tcc` structures. Its main feature is the fact that we will have to introduce an (normal) arc from the subsequent state in which a quiescence point occurs.

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*

$$\delta(T) \subseteq \llbracket S \rrbracket$$

where  $\delta$  is the function that represents the traces in  $T$  given by the sequences, starting from the root, of program stores in each `tcc` node in a path and  $\llbracket S \rrbracket$  represents the operational semantic of the `tcc` specification  $S$  in [10] enriched by the rule 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 specified system calculates the maximum value between two (see Figure 4).

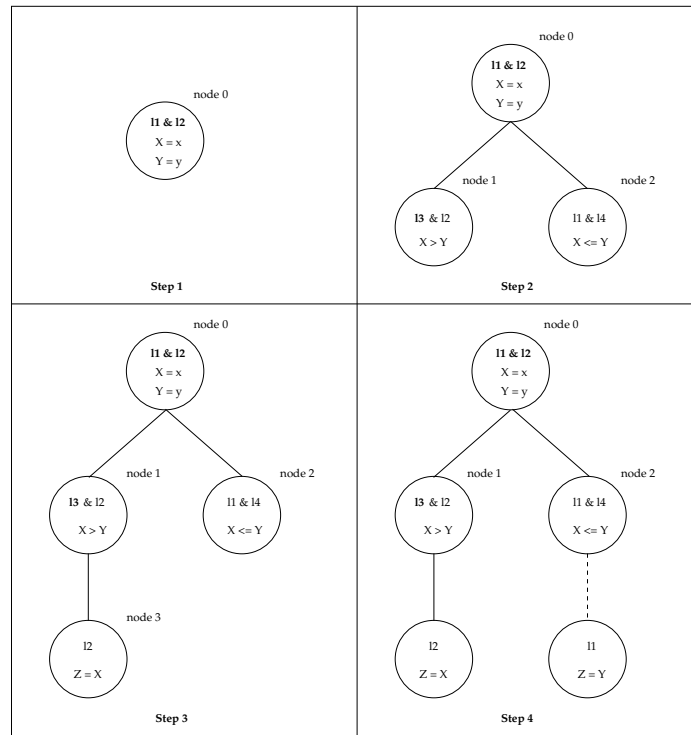


Figure 5: Sequence of steps

With this example we show the use of the two types of state transitions. To differentiate those two types of arcs in the graph we will draw the normal arcs as solid lines and time transitions as dotted ones. In the fourth step of Figure 5 it is showed the graph structure obtained when we apply our method to the “maximum” example (see Figure 4). Note that we have assumed that each node inherits all constraint from its predecessor but we will not show them explicitly, we only show the new information added to the store.

Step by step, the algorithm evolves as follows (in Figure 5 we show graphically the process). In the first step an initial node that represents the initial state is constructed. In the labeling of this node we have the two possible branches of execution represented by two labels and the real values of the program parameters. The two labels are active because their associated operators can be executed in the next step.

In the second step we have to develop all possible branches of execution, i.e. the branch of the Positive Ask and the one of the Negative Ask. The left one corresponds to the Positive Ask ( $l1$ ) and the right one to the Negative Ask ( $l2$ ). The *node 1* is labeled with the condition that imposes the guard of the **now-then** operator, the label of the Positive Ask body and the rest of the labels that have not been executed in this transition. In this case, the label  $l2$  appears disabled because the store does not allow the execution of the Negative Ask (the store

entails the guard conditions).

Considering now the right branch, in the *node 2* we have the Negative Ask execution. In the labeling of this node we have the *negation* of the condition that imposes the guard of the operator, the label of the Ask body and the rest of the labels (i.e. *l1* is disabled).

The third step takes the *node 1* and constructs the graph that models the body of the Positive Ask. It is added *node 3* that represents this and is labeled with the new variable values, the constraint accumulated from the previous nodes and the adequate set of labels. In this case *node 3* turns out to be a simple node but, in general, it could be a complex subgraph.

Finally, in the fourth and last step, the algorithm finds that all labels are disabled and then the execution cannot continue normally. It is necessary to increment the time instant. Variables are renamed and a temporal arc is introduced modeling the transition of the temporal operation that can be executed in that moment: the body of the Negative Ask (*l4* in the Figure). Hence, a new node is created (*node 4*) that contains the negated guard condition, the labels that have not been executed, and the new labels (in this case there are not new labels).

With this example we have seen the two different types of transitions that we can have. Now, to show the parallel behavior we show a more elaborated example. It consists on the calculation of the Fibonacci series. In Figure 6 we show the labeled version and the graph construction result can be seen in Figure 7.

```
{10} fib(Val) :: {11} Val:0 || {12}next {14} Val:1 ||
      {13} always {15} X^({16} now Val:X then
      {17} next {18} (Y,Z)^({19} now Val:Y then{112} skip ||
      {110} Z : X + Y || {111} next {113} Val:Z)).
```

Figure 6: Example: Fibonacci Program labeled.

We can see how the **always** operator introduces a cycle in the graph forcing the infinite behavior. This is natural if we think of the meaning of the represented program: it should calculate an infinite sequence of numbers. Temporal transitions are represented as discontinued arcs and note that a temporal transition cannot be made if some label is active. To understand the figure we have to think of having a set of nodes that can produce another set of nodes at each step of construction. The sequence of those sets of states would be the next one (as it is shown in Figure 7): {*node 0*}, {*node 1, node 2*},{*node 3, node 4, node 5*}, {*node 6, node 7*}, {*node 8*}, {*node 6, node 9, node 10*}, {*node 8, node 11, node 12*}, {*node 8, node 13*}, {*node 8, node 14, node 15*}, {*node 8, node 16, node 17*}, {*node 6, node 9, node 10, node 18*}, {*node 8, node 11, node 12*}... This last

set of nodes has already appeared in the series and then the sequence is repeated indefinitely.

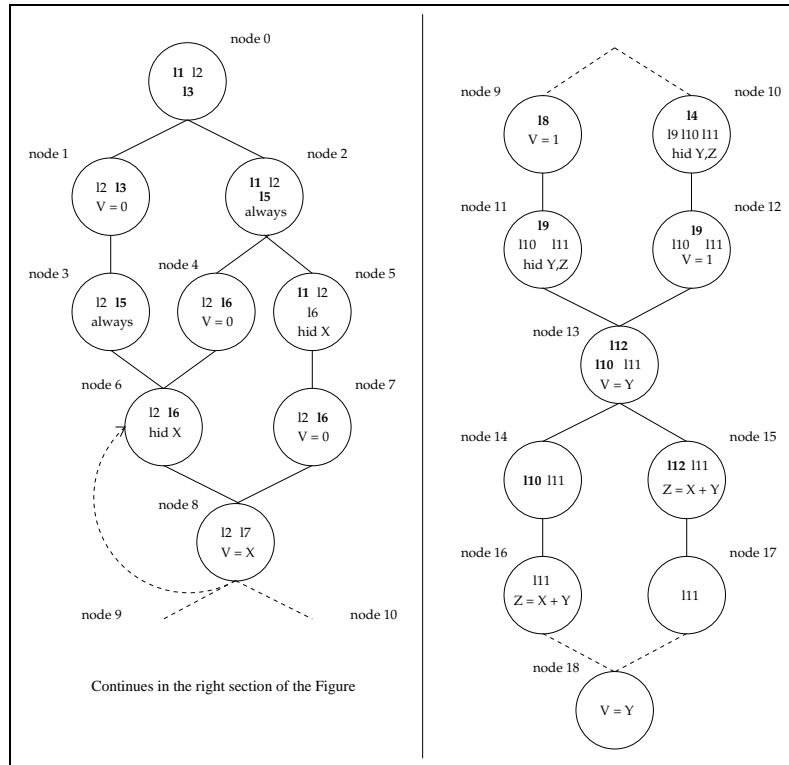


Figure 7: Graph behavior of the Fibonacci program

## 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  $t$ .
- $val_{t_I}(V)$  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 *time* transition is executed. This introduces a synchronization because a process that would execute a time operation has to wait until all other processes have finished their normal operations. Classical approaches are asynchronous because they consider that all parallel processes are independent of each others, 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.

We note a similarity to the work of Clarke *et al.* [3] in the fact that they introduce some synchrony 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 in its semantics and for this reason we can define a Model Checker for a given time interval. This time interval is determined 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 following step of our construction consists of representing the property that the user wants to verify. The formalization of this can be done in a way similar to Clark *et al.* [3].

## References

- [1] Berry, G. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11-17. Elsevier Science Publishers B. V. (North Holland), 1989.
- [2] Clarke, E. M., Emerson E. A., Sistla A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2),1986.
- [3] Clarke, E. M., Grumberg O., Peled D. Model Checking. *MIT Press*, 1999.
- [4] Halbwachs, N. *Synchronous programming of reactive systems*. The Kluwer international series in Engineering and Computer Science. Kluwer Academic Publishers, 1993.
- [5] Harel, D., Pnueli, A. Logics and Models of Concurrent Systems. volume 13, chapter On the development of reactive systems, pages 471-498, *NATO Advanced Study Institute*, 1985.

- [6] Henzinger T. A., Manna Z., Pnueli A. Timed Transition Systems. In REX workshop Real-Time: Theory in Practice, *Lecture Notes in Computer Science 600*, Springer-Verlag, pp. 226-251, 1992.
- [7] Manna, Z., Pnueli, A. Temporal Verification of Reactive Systems. Safety, *Springer-Verlag*, 1985.
- [8] Sipma H. B., Uribe T. E., Manna Z. Deductive Model Checking. In 8th International Conference on Computer-Aided Verification, LNCS vol. 1102, pp. 209-219, Springer-Verlag, July 1996.
- [9] Saraswat, V. A., Jagadeesan, R., Gupta, V. 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.
- [10] Saraswat, V. A., Jagadeesan, R., Gupta, V. 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.
- [11] Saraswat, V. A., Jagadeesan, R., Gupta, V. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475-520, 1996