Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 246 (2009) 131–145

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Tool for Generating a Symbolic Representation of tccp Executions[1]

Alexei Lescaylle[2] Alicia Villanueva[3]

DSIC, Universidad Politécnica de Valencia
Camino de Vera s/n,
46022 Valencia, Spain

Abstract

The *Timed Concurrent Constraint language* (tccp) was defined by F. de Boer *et al.* as an extension of the *Concurrent Constraint Paradigm* (Saraswat, 1993) for specifying reactive and embedded systems. In this paper, we describe the StructGenerator system which, given the specification of a tccp program, constructs a symbolic representation (a tccp Structure) modeling the behavior of such tccp program. The resulting structure allows one to verify the program by using a *model-checking* algorithm. It is similar to a Kripke Structure but, due to the nature of the ccp model, it differs from the classical approach in some important points that will be described along the paper. The StructGenerator system, implemented in C++, takes as input a file containing the specification of a tccp program and generates the associated tccp Structure. Along the paper, we cover the design and implementation of StructGenerator. We also demonstrate its functionality carrying out the execution of two practical examples.

Keywords: Timed concurrent constraint programming, symbolic representation, tool demonstration.

1 Introduction

The *Concurrent Constraint Paradigm* (ccp in short) [10] is a simple but powerful model based on the notion of *store-as-constraint* instead of the classical notion of *store-as-valuation.* Therefore, the computational model is based on states which are composed of a conjunction of constraints instead of being defined as a valuation of variables. The *Timed Concurrent Constraint language* (tccp in short) was introduced in [4] as an extension of ccp for specifying reactive systems in an intuitive way. The authors introduced a notion of time within the language semantics and new agents to handle negative information, i.e., information that is not present in

[1] This work has been supported by the Spanish MICINN under grant TIN2007-68093-C02-02, HA2006-0007, and by the Universidad Politécnica de Valencia Valencia, program PAID-06-07 TACPAS.

[2] Email: alescaylle@dsic.upv.es

[3] Email: villanue@dsic.upv.es

doi:10.1016/j.entcs.2009.07.019

the system. Negative information is needed for modeling behaviors such as timeouts or preemption in reactive systems.

During the last years, verification techniques for concurrent and reactive systems have been widely developed. The *model-checking* technique [5] is a formal technique that allows one to verify whether a property is satisfied by a model. This technique suffers the so-called *state explosion problem* which motivates the development of many optimization approaches to mitigate it. We can find in the literature many optimizations based on abstract interpretation, partial order, symbolic representations, etc. for different modeling and (fragments of) programming languages. The ccp framework in general, and the tccp language in particular, thanks to the store-as-constraint approach, symbolically represents sets of classical states as conjunctions of constraints, thus achieving a natural compression of the search space. Note that this fact does not prevent us from applying other optimization techniques such as [1,2].

A model-checking algorithm for tccp was proposed in [6] which given a tccp program transforms it into a symbolic representation (the tccp Structure) which is the input of the verification phase. The proposal in [6] is similar to the classical one, with the tccp Structure playing the role of the Kripke Structure. However, as we will show later, the tccp Structure differs from the Kripke Structure in some important points, so the verification algorithm had to be reformulated and adapted to the ccp model.

In this paper we present StructGenerator, a system that automatically constructs the tccp Structure given the specification of a tccp program. The construction algorithm differs in some aspects w.r.t. the one proposed in [6], incorporating some notions from [3] that make the tccp framework more flexible.

The paper is organized as follows. Section 2 briefly presents the tccp language. Section 3 describes the notion of tccp Structure presenting the main difficulties from dealing with the ccp model. In that section we also show how we implemented the construction of the tccp Structure. Section 4 shows how to use the tool by developing two running examples and, finally, Section 5 concludes.

2 The tccp language

The Timed Concurrent Constraint Language, tccp, is a concurrent declarative language defined in [4] as an extension of the Concurrent Constraint Programming paradigm (ccp, [10]). In the ccp model, the notion of store-as-valuation is replaced by the notion of store-as-constraint. The computational model is based on (1) a global store where constraints are accumulated, and (2) a set of agents that may interact with others via the store. The ccp model is non-deterministic and there is no notion of time defined. Intuitively, the execution of a ccp program evolves by asking and telling information to the store. It is a simple but powerful model in which partial information can easily be handled. The temporal extension of tccp introduces a notion of time within the semantics, i.e., no special agent for time passing is defined. However, a new agent able to handle negative information (the *conditional agent*) is introduced. We can find other approaches to the temporal extension of ccp in the literature that make different design choices [11,7,9]. In tccp, the execution of a program evolves, again, by asking and telling information to the store, but this time these actions consume time. In practice this means that both, consults and updates to the store, take time. Finally, we remark that it is not possible to remove information from the store, thus differently from other extensions, the store behaves monotonically.

The model is parametric w.r.t. a cylindric constraint system $\mathcal{C}$ defined in [4] as a structure $\langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists \rangle$ where $\sqcup$ is a lub operation, *true* and *false* the least and greater elements of $\mathcal{C}$, *Var* a denumerable set of variables and $\exists$ a cylindric operation on variables. Details on constraint systems can be found in [4,10].

As we have said before, the tccp language was defined to model reactive or embedded systems. In these systems, the absence of information can cause the execution of an action, what motivates the introduction of the new conditional agent w.r.t. the ccp model. Let us briefly recall the syntax of tccp [4]:

$$A ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A \mid A || A \mid \exists x\, A \mid \mathsf{p}(\overline{x})$$

where c, c_i are *finite constraints* (i.e., atomic propositions) of $\mathcal{C}$. A tccp program P is an object of the form $D.A$, where D is a set of procedure declarations of the form $\mathsf{p}(\overline{x})$:$-A$, and A is an agent. Intuitively, the skip agent finishes the execution of a process by doing nothing; $\mathsf{tell}(c)$ adds the constraint c to the store; The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ non-deterministically executes (in the following time instant) one of the agents A_i provided its guard c_i is satisfied. In case no condition c_i is entailed, the choice agent *suspends*; The conditional agent (now c then $A1$ else $A2$) executes $A1$ if the store satisfies c, otherwise executes $A2$; $A1||A2$ executes the two agents $A1$ and $A2$ in parallel (the concurrent model used is *maximal parallelism*); The $\exists x\ A$ agent is used to define the variable x local to the process A; Finally, $\mathsf{p}(\overline{x})$ is the procedure call agent where $\overline{x}$ denotes the set of parameters of the process p.

Agents are synchronized by means of a global clock. In the semantics of tccp, the only agents that consume time are the tell, the choice, and the procedure call agents. The store in the original tccp execution model [4] can be seen as a blackboard where information is continuously written and never canceled. The store grows monotonically, thus it is not possible to change the value of a given variable. We can use *streams* to model the evolution of variable values along the time. This allows one to handle imperative-style variables. In order to retrieve in a simpler way information regarding the order in which the information was added to the store, [3] proposes a new computational model in which a *structured store* substitutes the classical notion of store. The StructGenerator system considers this new computational model. Intuitively, a *structured store* [3] consists of a timed sequence of stores st_i where each store contains the information added at the *ith* time instant. Notions such as entailment had to be modified to obtain a model that respects the original semantics of tccp.

3 A symbolic representation of tccp executions

A graph structure (tccp Structure) for modeling tccp traces was proposed in [6] in the context of the definition of a model-checking algorithm. This structure can be seen as a variant of a Kripke Structure where, following the ccp model, the notion of state as valuation of variables is replaced by the notion of state as a conjunction of constraints. This means that a node in the tccp Structure can represent a set of classical nodes of a Kripke Structure. The StructGenerator system implements the construction of such structure. As we will show, the main difficulties for this constructions arise from the monotonic nature of the tccp store. Moreover, both in the construction of the tccp Structure phase and later in the verification phase, we have to take into consideration that, differently from the classical approach, the absence of some information in the store does not (necessary) mean that the negation of that information holds.

Let us briefly describe the symbolic representation. A state of the tccp Structure contains a set of atomic propositions; more specifically, it consists of a set of atomic constraints from the underlying constraint system in tccp. Each state of the tccp Structure also contains a set of labels representing the current execution step. These labels are uniquely associated to each occurrence of an agent in the tccp program. Note that a pre-process for labeling the program is needed. Labels allow us to set those agents that must be executed in the following time instant.

Since during the execution of a program the store grows monotonically, by definition, and differently from the classical approach, there cannot be two states syntactically equal. This means that in order to define a finite representation of executions, a notion of equivalence between states (which eventually would determine the cycles in the graph structure) was needed. This notion is also necessary for having a finite construction algorithm.

Informally, we say that two states are equivalent if (1) the set of labels in both states coincide, and (2) the set of constraints in one state is equivalent to the set of constraints in the second state modulo renaming [6]. In order to deal with streams equivalence, we use the notion of *current* value (i.e., the more recent added value [3]) of the stream to set whether the stream is *equal* in two states. Note that the number of constraints defining a stream is always increasing (thus we never have two states syntactically equal) but, following the imperative-style notion of variable, at some specific execution point, we are interested just on the current value of such stream. The definition of state equivalence allows us to overcome the problem of termination of the algorithm caused by the monotonic behavior of the tccp store. Remember that tccp is used to model reactive systems. In case of being modeling arithmetic functionality, termination cannot be generally ensured.

3.1 The implementation

StructGenerator has been implemented in C++. It consists of approximately 3100 lines of code divided in 10 classes. Each class handles one of the entities of tccp (agents, constraints, declarations, stores, states, . . .). We decided to implement the system in C++ since we needed to connect it to some constraint solvers, and we found some interfaces to ease these connections had been defined for that language. Moreover, we had to deal with complex data structures, thus C++ provided us the capability of defining classes and structuring them.

As we have said above, the StructGenerator system constructs in an automatic way the already described tccp Structure, so it aims to representing in a symbolic way the behavior of a tccp program. An algorithm for this construction was proposed in [6]. We have followed the main ideas in that algorithm, but, as mentioned above, we decided to refine some of the definitions by using the more flexible computational model in [3] which allowed us to deal more easily with streams.

Intuitively, the implementation moves through the following phases. First of all, given the specification of a tccp program in a text file, the system parses the program where each occurrence of an agent is labeled with a different tag. Thereafter, the initial states are built and, from each initial state, we construct the states reached from such state following the semantics of the language. States are generated but are not introduced into the structure until confirmation. Confirmation is achieved when it is checked whether there exists no equivalent node in the structure. Let us describe the generation process from each node. This generation process is iteratively performed for each new state (initially for the initial states):

1 to generate the possible successors of a node following the semantics of the language,

2 to check whether there exist an equivalent node for each possible successors,

3 to confirm the introduction in the structure of the new node:
   (a) in case that an equivalent node does not exist, then the successor is added to the tccp structure
   (b) in case that an equivalent node exists, the calculated *renaming* that makes the two states equivalent is associated to the edge connecting the original node to the equivalent one. The successor node is discarded.

The iterative process ends when all the nodes have been processed for successors generation. The key point for the construction is the second and third phases, where we look for equivalent nodes. As we have said before, this step is crucial to (partially) ensure the termination of the algorithm.

As shown in Fig. 1, the StructGenerator system has been structured in 2 modules, Program Parser and Construction Process. The Program Parser module takes as input the file that contains the program specification. Thereafter, by using the auxiliary procedures Declaration Parser and Agent Parser, generates the data structure Declarations which stores all the program information: agents, constraints, labeling, etc. The generated information Declarations is the input of the Construction Process module, which generates as output the graph structure. To this end, it uses a Node Creation process corresponding to the first phase in the iterative process above. In brief, it constructs the states of the structure by using three auxiliary functions:

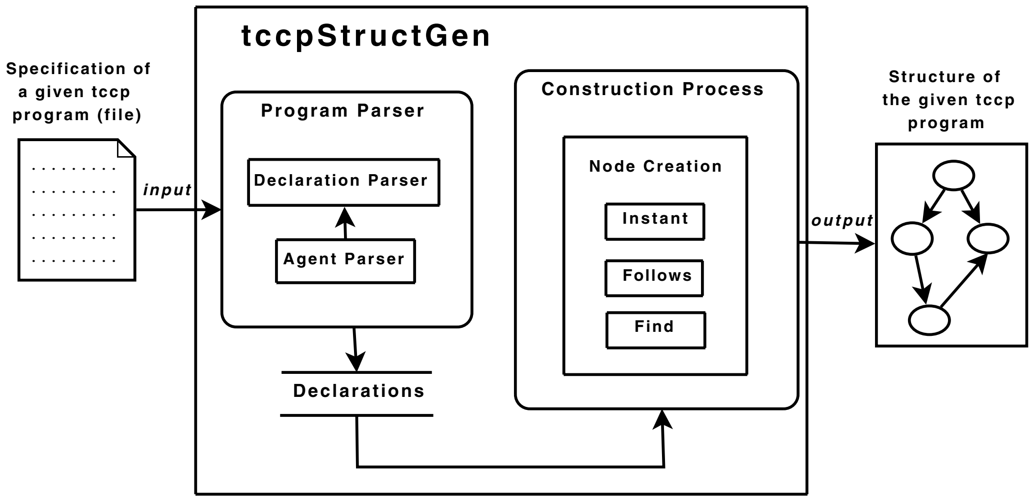

Fig. 1. StructGenerator framework

instant, follows and find[4]. Intuitively, instant and follows calculate, following the tccp semantics, (1) the constraints that an agent can add in a time instant and (2) the agents that must be executed in the following time instant, respectively. In other words, they calculate the contents of new nodes (labels and constraints). find is the function that looks for equivalent nodes in the structure as described above.

The interface of our tool is console guided. To run the tool, we have to enter the command StructGenerator. Then, the system asks the user for the filename where the specification of the tccp program is written. Fig. 2 shows a simple execution.

```
user@pcname:~$ StructGenerator
Enter the filename of the tccp program(*.tccp or *.txt):
  ...microwave_error.txt
-------------------------------------------------------------

.....Creating graph of the declaration:  microwave_error

      - creating children of the state:  1
      - creating children of the state:  2
      - creating children of the state:  3
      - creating children of the state:  4
⋮
```

Fig. 2. StructGenerator run

[4] The interested reader can find in [6] the definitions of the original instant and follows functions.

```
{ld} microwave_error(Door,Button,Error) :-
{le0}∃ D,B,E ({lp1}({lt2}tell(Error=[_|E]) ||
              {lp3}({lt4}tell(Door=[_|D]) ||
              {lp5}({lt6}tell(Button=[_|B]) ||
              {lp7}({ln8}now(Door=[open|D] ∧ Button=[on|B])
                         then
                         {lp9}({le10}∃E1({lt11}tell(E=[yes|E1])) ||
                              {le12}∃B1({lt13}tell(B=[off|B1]))) ||
                         else
                           {le14}∃E1({lt15}tell(E=[no|E1])) ||
                           {lc16}microwave_error(D,B,E))))).
```

Fig. 3. The microwave_error declaration in tccp.

4 Practical examples

In this section, we present two practical examples. The first one is the (partial) specification of a microwave oven controller that we have borrowed from [6]. The second one is the specification of the scheduler example used in [1] to motive the symbolic version of the model checker. We first provide the tccp specification of the concurrent system and then we provide the tccp Structure generated by StructGenerator. We also show graphically the connection between nodes.

4.1 *The* microwave_error *System*

In Fig. 3 we show the tccp specification of the system. To make clearer the relation between the specification and the generated graph, we show the labeled version of the program. Labels appear within braces { }.

The procedure declaration microwave_error models the process of detecting when the door of a microwave is open at the same time that the system is turned-on. This situation is controlled by the conditional agent ln8. In case the condition holds, the process forces (with the tell agent lt13) the microwave to be turned-off in the following time instant. Note that the tell agent is executed at the same time instant when the error is detected, but the told constraint is only available to others in the following time instant. Moreover, an error signal must be emitted (agent lt11). If the condition does not hold, then the program emits (via another tell agent lt15) a signal of *no error* that will be available in the global store at the following time instant. These signals may be captured by other processes, thus it can be seen that the store allows the synchronization of processes.

Bellow we show the symbolic representation obtained when executing StructGenerator with input the (non-labeled version of) microwave_error. Currently, the output is shown in a textual way, listing the set of states in the graph structure together with the relation between states. For the considered system, 7 states are generated. Each state contains an identifier, for example state 1. In this case, state 1 contains:

- a set of `constraints: yes(Door=[d(open)|D] and Button=[b(on)|B])`,
- a set of `labels:lt2-microwave_error, lt4-microwave_error, lt6-microwave_error, ...`,
- a set of defined `variables: Door, Button, Error, E1, B1, D, B, E` and,
- finally, looking at the `children` part, we can say that `state 1` is linked with `state 3` and `state 4`, with no renaming defined on the edge.

```
   Graph(s) successfully created.....
-------------------------------------------------------------
-- tree: 0 -> called: microwave_error --

-- state 0 --
constraints: (true),
labels: ld-microwave_error,
variables: Door,Button,Error,
children:
in tree: 0 -> state: 1 - renaming:
in tree: 0 -> state: 2 - renaming:

-- state 1 --
constraints: yes(Door=[d(open)|D] and Button=[b(on)|B]),
labels: lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
lt11-microwave_error,lt13-microwave_error,lc16-microwave_error,
variables: Door,Button,Error,E1,B1,D,B,E,
children:
in tree: 0 -> state: 3 - renaming:
in tree: 0 -> state: 4 - renaming:

-- state 2 --
constraints: not(Door=[d(open)|D] and Button=[b(on)|B]),
labels: lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
lt15-microwave_error,lc16-microwave_error,
variables: Door,Button,Error,E1,D,B,E,
children:
in tree: 0 -> state: 5 - renaming:
in tree: 0 -> state: 6 - renaming:

-- state 3 --
constraints: (Error=[e(_)|E]),(Door=[d(_)|D]),(Button=[b(_)|B]),
(E=[e(yes)|E1]),(B=[b(off)|B1]),yes(D=[d(open)|D'] and B=[b(on)|B']),
labels: lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
lt11-microwave_error,lt13-microwave_error,lc16-microwave_error,
variables: Door,Button,Error,E1,B1,D,B,E,B1',E1',D',B',E',
children:
```

```
in tree:  0 -> state:  3 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,B1'/B1,D''/D',B''/B',
in tree:  0 -> state:  4 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,B1'/B1,D''/D',B''/B',

   -- state 4 --
constraints:  (Error=[e(_)|E]),(Door=[d(_)|D]),(Button=[b(_)|B]),
(E=[e(yes)|E1]),(B=[b(off)|B1]),not(D=[d(open)|D'] and B=[b(on)|B']),
labels:  lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
lt15-microwave_error,lc16-microwave_error,
variables:  Door,Button,Error,E1,B1,D,B,E,B1',E1',D',B',E',
children:
in tree:  0 -> state:  5 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',
in tree:  0 -> state:  6 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',

-- state 5 --
constraints:  (Error=[e(_)|E]),(Door=[d(_)|D]),(Button=[b(_)|B]),
(E=[e(no)|E1]),yes(D=[d(open)|D'] and B=[b(on)|B']),
labels:  lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
lt11-microwave_error,lt13-microwave_error,lc16-microwave_error,
variables:  Door,Button,Error,E1,D,B,E,E1',B1,D',B',E',
children:
in tree:  0 -> state:  3 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',
in tree:  0 -> state:  4 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',

-- state 6 --
constraints:  (Error=[e(_)|E]),(Door=[d(_)|D]),(Button=[b(_)|B]),
(E=[e(no)|E1]),not(D=[d(open)|D'] and B=[b(on)|B']),
labels:  lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
lt15-microwave_error,lc16-microwave_error,
variables:  Door,Button,Error,E1,D,B,E,E1',D',B',E',
children:
in tree:  0 -> state:  5 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',
in tree:  0 -> state:  6 - renaming:  E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',
```

In this example, just one *tree* is generated since there is just one procedure declaration. We can see this fact since all nodes are `in tree: 0`. The system generates as many trees as declarations in the program. Note that some relations, for example those of `state 0` to `state 1` and `state 2`, have no renaming. Renaming is associated to the relation only when cycles are established. We can graphically observe the relation between nodes in the generated structure in Fig. 4 [5].

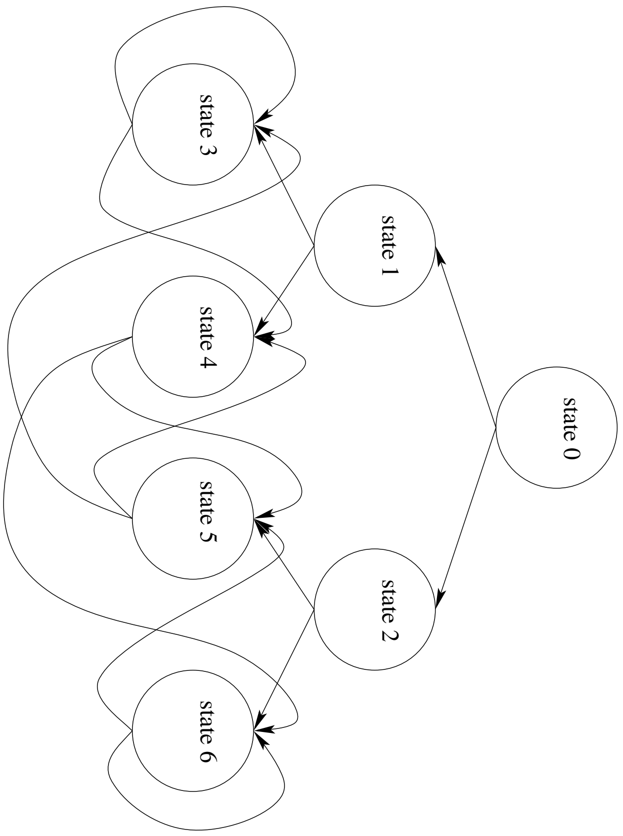

Fig. 4. The relation between nodes in the tccp Structure generated

Intuitively, `state 1` models the case when the condition in the program holds (thus an error occurs), whereas `state 2` represents when the condition does not hold. Consider now `state 2`. This state is related to states `5` and `6`. These states are related to already existing nodes: `state 5` travels to states `3` and `4`. This means that the successors of state `5` are equivalent (following the definition of state equivalence informally given above) to these states. In these cases, the renaming that make the states equivalent is provided.

4.2 The `build` *System*

In order to illustrate how one tree is generated for each declaration, we show in Fig. 5 the labeled version of a `tccp` program example borrowed from [1]. It consists of two predicates. The first one, `build`, models the duration of three different tasks of the process of building a house. It gets the value of variables `D1`, `T1` and `E1` –representing the duration of the tasks that must be scheduled– by calling the

[5] The figure does not show the renaming associated to each cyclic edge.

predicate get_constraints. In parallel, an ask agent checks whether the value of the three variables are instantiated to integer numbers and, in that case, several constraints representing the scheduling restrictions are added to the store. Finally, a recursive call to the building process is made, allowing us to recalculate the planning schedule. The predicate get_constraints is simpler since it just adds to the store the values of the different duration of tasks.

```
{ld} build([PD|PD'],[PT|PT'],[PE|PE'],[PA|PA']) :-
{le0}∃ D1,T1,E1 ({lp1}({lx2} get_constraints(D1,T1,E1) ||
                {la3}ask(atom(D1) and atom(T1) and atom(E1))→
                       {lp4}{lt5}(tell(PD+D1=<PT) ||
                       {lp6}{lt7}(tell(PT+T1=<PE) ||
                       {lp8}{lt9}(tell(PE+E1=<PA) ||
                       {lc10} build(PD',PT',PE',PA'))))).


{ld} get_constraints(W1,I1,P1) :-
{lp0}{lt1} (tell(W1) ||
{lp2}{lt3} (tell(I1) ||
{lt4} tell(P1))).
```

Fig. 5. The build system program in tccp

The resulting structure from the execution of StructGenerator is shown below. Note that in state 1, the set of labels contains the label to the procedure call agent lx2. state 1 is linked to nodes state 2 and state 3, that, as one can observe, have labels from agents in the get_constraint declaration and in the build declaration.

```
  Graph(s) successfully created.....
-------------------------------------------------------------
-- tree:  0 -> called:  build --

-- state 0 --
constraints:  (true),
labels:  ld-build,
variables:  PD, PD', PT, PT', PE, PE', PA, PA',
children:
in tree:  0 -> state:  1 - renaming:

-- state 1 --
constraints:  (true),
labels:  lx2-build,la3-build,
variables:  PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree:  0 -> state:  2 - renaming:
```

```
in tree:  0 -> state:  3 - renaming:

-- state 2 --
constraints:  (true),yes(atom(D1) and atom(T1) and atom(E1)),
labels:  lt1-get_constraints,lt3-get_constraints,lt4-get_constraints,
lt5-build,lt7-build,lt9-build,lc10-build
variables:  PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree:  0 -> state:  4 - renaming:

-- state 3 --
constraints:  (true),
not(atom(D1) and atom(T1) and atom(E1)),
labels:  lt1-get_constraints,lt3-get_constraints,lt4-get_constraints,
la3-build,
variables:  PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree:  0 -> state:  5 - renaming:
in tree:  0 -> state:  6 - renaming:

-- state 4 --
constraints:  (D1),(T1),(E1),(PD+D1=<PT),(PT+T1=<PE),(PE+E1=<PA),
labels:  lx2-build,la3-build,
variables:  PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1, D1', T1',
E1',
children:
in tree:  0 -> state:  2 - renaming:  D1'/D1,T1'/T1,E1'/E1,
in tree:  0 -> state:  3 - renaming:  D1'/D1,T1'/T1,E1'/E1,

-- state 5 --
constraints:  (D1),(T1),(E1),yes(atom(D1) and atom(T1) and atom(E1)),
labels:  lt5-build,lt7-build,lt9-build,lc10-build,
variables:  PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree:  0 -> state:  4 - renaming:  PT/PD,T1/D1,PE/PT,E1/T1,PA/PE,

-- state 6 --
constraints:  (D1),(T1),(E1),not(atom(D1) and atom(T1) and atom(E1)),
labels:  la3-build,
variables:  PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree:  0 -> state:  5 - renaming:
in tree:  0 -> state:  6 - renaming:

-----------------------------------------------------------
```

```
-- tree:  1 -> called:  get_constraints --

-- state 0 --
constraints:  (true),
labels:  ld-get_constraints,
variables:  W1, I1, P1,
children:
in tree:  1 -> state:  1 - renaming:

-- state 1 --
constraints:  (true),
labels:  lt1-get_constraints,lt3-get_constraints,lt4-get_constraints,
variables:  W1, I1, P1,
children:
in tree:  1 -> state:  2 - renaming:

-- state 2 --
constraints:  (W1),(I1),(P1),
labels:
variables:  W1, I1, P1,
children:
```

In Fig. 6 we graphically show the relation between states of the system. `state 2` represents the case when the condition of the ask agent does hold whereas `state 3` represents when the condition does not hold. `state 2` is related to `state 4` which is related again to `state 2` and also to `state 3`. This is due to the execution of the procedure call agent `lc10-build` which models the recursive call to the building process. The second tree represents the predicate get_constraints composed by three states which describe the process of adding the corresponding variables to the store.

5 Conclusions and Future Work

In this paper we have presented the StructGenerator system that, given the specification of a tccp program, constructs a symbolic representation of the set of all possible executions of the program. The system is publicly available at the url `http://www.dsic.upv.es/~villanue/tccp-StructGenerator/`. With respect to the symbolic representation, the tccp Structure can be seen as a variant of a Kripke Structure where the notion of node is adapted to the ccp framework and, differently from the classical approach, a renaming may be associated to some edges. Due to the tccp model, the construction of such symbolic representation becomes non trivial, since due to the monotonic behavior of the store, we have to deal with infinite sets of states. To avoid this problem, a notion of equivalence among states is used, which, combined with the use of the *current value* of streams, allowed us

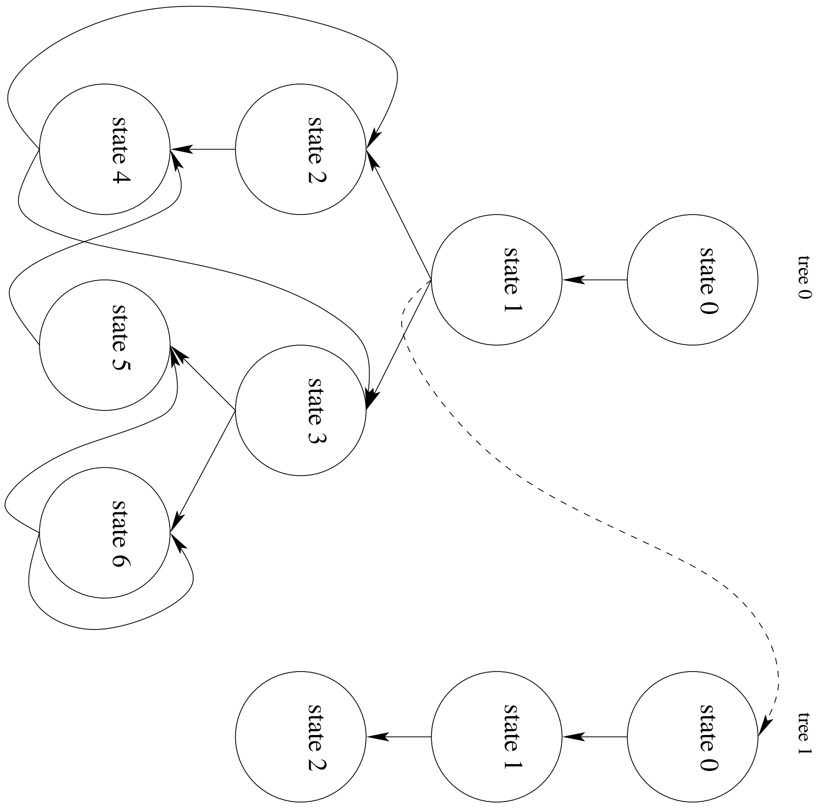

Fig. 6. The relation between nodes in the `build` system

to implement a finite process for the generation.

As future work, we plan to consider in this tool the new agents recently presented in [8]. We also plan to adapt the model-checking algorithm to this new framework (including the new agents and formulated within the structured computation model of [3]). Note that the tccp Structure generated by StructGenerator has enough information to be the input of such model-checking tool. Finally, we would like to improve the interface of our tool, by automatically showing the graphical version of the relation between nodes.

References

[1] M. Alpuente, M. Falaschi, and A. Villanueva. A Symbolic Model checker for tccp Programs. In *Proceedings of the International Workshop on Rapid Integration of Software Engineering techniques (RISE'04)*, volume 3475 of *Lecture Notes in Computer Science*, pages 45–56. Springer Verlag, 2005.

[2] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Semantic Framework for the Abstract Model Checking of `tccp` programs. *Theoretical Computer Science*, 346:58–95, 2005.

[3] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Real-Time Logic for tccp verification. *Journal of Universal Computer Science*, 12(11):1551–1573, November 2006.

[4] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[6] M. Falaschi and A. Villanueva. Automatic Verification of Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 6(3):265–300, 2006.

[7] V. Gupta, R. Jagadeesan, V. A. Saraswat, and D. G. Bobrow. Programming in hybrid constraint languages. In *Hybrid Systems II, volume 999 of LNCS*, pages 226–251. Springer Verlag, 1995.

[8] A. Lescaylle and A. Villanueva. Using tccp for the Specification and Verification of Communication Protocols. In *Proceedings of the 16th International Workshop on Funcional and (Constraint) Logic Programming (WFLP'07)*, pages 169–183, Paris (France), July 2007.

[9] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: denotation, logic and applications, 2002.

[10] V. A. Saraswat. *Concurrent Constraint Programming Languages*. The MIT Press, Cambridge, MA, 1993.

[11] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Press, 1994.