A Symbolic Model Checker for tccp Programs*

M. Alpuente[1], M. Falaschi[2], and A. Villanueva[1]

[1] DSIC, Technical University of Valencia,
Camino de Vera s/n, E-46022 Valencia, Spain
[2] DIMI, University of Udine,
Via delle Scienze 206, I-33100 Udine, Italy

Abstract. In this paper, we develop a symbolic representation for *timed concurrent constraint* (tccp) programs, which can be used for defining a lightweight model–checking algorithm for reactive systems. Our approach is based on using streams to extend *Difference Decision Diagrams* (DDDs) which generalize the classical *Binary Decision Diagrams* (BDDs) with constraints. We use streams to model the values of system variables along the time, as occurs in many other (declarative) languages. Then, we define a symbolic (finite states) model checking algorithm for tccp which mitigates the state explosion problem that is common to more conventional model checking approaches. We show how the symbolic approach to model checking for tccp improves previous approaches based on the classical Linear Time Logic (LTL) model checking algorithm.

Keywords: Lightweight formal methods, Model Checking, Timed Concurrent Constraint Programs, DDDs.

1 Introduction

In the last decades, formal verification of industrial applications has become a hot topic of research. As the complexity of software systems increases, lightweight automatic verification tools which are able to guarantee, at little cost, the correct behavior of such systems are dramatically lacking. *Model checking* is a fully automatic formal verification technique which is able to demonstrate certain properties formalized as logical formulas which are automatically checked on a model of the system; otherwise, it provides a counterexample which helps the programmer to debug the wrong code. However, its potential for push–button verification is not easily realizable due to the well-known *state-space explosion* problem. Recent advances in model checking deal with huge state-spaces by using symbolic manipulation algorithms inside model checkers [7].

* This work has been partially supported by MCYT under grants TIC2001-2705-C03-01, HU2003-0003, by Generalitat Valenciana under grants GR03/025, GV04/389 and by ICT for EU-India Cross Cultural Dissemination Project under grant ALA/95/23/2003/077-054.

N. Guelfi (Ed.): RISE 2004, LNCS 3475, pp. 45–56, 2005.

The *concurrent constraint* paradigm (cc) was presented in [11] to model concurrent systems. A global store consisting of a set of constraints contains the information gathered during the computation. Constraints are dynamically added to the store which can also be consulted. The programming model was extended in [3] over a discrete notion of time in order to deal with reactive systems, that is, systems which continuously interact with their environment without producing a final result and execute infinitely along the time. The use of constraints, the inherent concurrency and the notion of time which lay in tccp permit to program reactive systems in a very natural way. Reactive systems are usually modeled as concurrent systems which are more difficult to be manually debugged, simulated or verified than sequential systems. In previous works ([8, 9]) we have defined an explicit model checking algorithm for tccp programs. Such method automatically constructs a model of the system which is similar to a Kripke Structure.

The main purpose of this work is to improve the exhaustive model checking algorithm defined in the last years to verify tccp programs. Starting from the graph representation of [9], in this paper we formalize a symbolic representation of reactive systems written in tccp. Such representation allows us to formulate a symbolic model checking algorithm which allows us to verify more complex reactive systems. To the best of our knowledge, this work defines the first symbolic model checking algorithm for tccp. In order to ensure the termination of our approach, we refer to finite state systems in this work. It would be possible to remove this assumption and consider infinite state systems by adapting to our context standard abstract interpretation techniques [2], or by requiring the user to indicate a finite time interval for limiting the duration of tccp computations.

The paper is organized as follows. In Section 2 we recall the tccp programming language and the tccp Structure which can be derived from the program specification and which is the reference point of this work. In Section 3 we introduce the verification method that we propose, and in Section 3.2 we define the algorithms that allow us to automatize the model construction process. In Section 5 we develop an example of property verification. Finally, Section 6 concludes. More details and missing definitions can be found in [1].

2 The tccp Framework

The cc paradigm has some nice features which can be exploited to improve the difficult process of verifying software: the declarative nature of the language ease the programming task of the user, and the use of constraints naturally reduces the state space of the specified system.

The *Timed Concurrent Constraint Language* (tccp) was developed in [3] by F. de Boer *et al.* as a framework for modeling reactive and real-time systems. It was defined by extending the concurrent computational model of the cc paradigm [11] with a notion of discrete time.

Basically, a cc program describes a system of agents that can add (*tell*) information into a store as well as check (*ask*) whether a constraint is entailed by such global store. The basic agents defined in tccp are those inherited from cc

plus a new conditional agent described below. Moreover, a *discrete global clock* is provided. Computation evolves in steps of one time unit by adding or asking (entailment test) some information to the store. It is assumed that *ask* and *tell* actions take one time unit, and the parallel operator is interpreted in terms of maximal parallelism. Moreover, it is assumed that constraint entailment tests take a constant time independently of the size of the store[1].

Let us first recall the notion of constraint system which underlies the tccp programming language[2]. A simple constraint system can be defined as a set D of tokens (or primitive constraints) together with an entailment relation $\vdash \subseteq 2^D \times D$. Concurrent constraint languages, actually use a notion of *cylindric* constraint system, which consists of a simple constraint system plus an existential quantification operator which is monotonic, conservative and supports renaming. This additional operator allows one to model local variables in a given agent. The formal definition of the notion of *cylindric constraint system* can be found in [3].

In this work, we consider a specific constraint system which allows us to verify a class of software systems. In particular, we consider a constraint system with two kind of tokens: the first one allows us to handle arithmetical constraints whereas the other constraints are used for representing streams. In tccp, streams are modeled as lists of terms. These lists represent the value of a given system variable along the time. Intuitively, in the current time instant, the head of the list represents the value of a variable and the tail of the list models the future. The entailment relation for lists is specified by Clark's Equality Theory. For example, following Prolog notation for lists, $[X|Z]{=}[a|Y]$ entails $X{=}a$ and $Z{=}Y$.

We use $\mathcal{V}$ to denote the set of variables ranging over $\mathbb{R}$ (or $\mathbb{Z}$), and $\mathcal{LV}$ is the set of lists of such variables. From now, we will use $\mathbb{D} \in \{\mathbb{R}, \mathbb{Z}\}$ to denote arbitrarily one of the two domains. Roughly speaking, we define the set of tokens of our constraint system as containing the set of difference constraints of the form $X - Y \leq c$ and $X - Y < c$, as well as the set of stream constraints of the form $V = [\,]$, $V = [X|W]$ and $V = [c|W]$, where X and Y belong to $\mathcal{V}$, V and W are in $\mathcal{LV}$, and the constant c belongs to $\mathbb{D}$.

We define the set AP of atomic propositions as the set of tokens of the (cylindric) constraint system above. In the rest of the paper, we identify the notion of (finite) constraint with atomic propositions.

Let us now recall the syntax of tccp, defined in [3] as follows[3]:

Definition 1 (tccp Language). *Let C be a (cylindric) constraint system. The syntax of agents of the language is given by the following grammar:*

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

[1] In practice, some syntactic restrictions are imposed in order to ensure that these hypotheses are reasonable (see [3] for details).

[2] A formal definition of the constraint system can be found in [1].

[3] The operational and denotational semantics of the language can be found in [3].

where c, c_i are finite constraints *of C. A* tccp process *P is an object of the form $D.A$, where D is a set of procedure declarations of the form* $\mathsf{p}(x) :- A$*, and A is an agent.*

The stop agent terminates the execution whereas the $\mathsf{tell}(c)$ agent adds the constraint c to the store. Nondeterminism is modeled by the choice agent (written $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$) that executes nondeterministically one of the choices whose guard is satisfied by the store. The agent $A \,||\, A$ represents the concurrent component of the language, and $\exists x\, A$ is the existential quantification, that makes the variable x local to the agent A. The agent for the procedure call is $\mathsf{p}(x)$.

Finally, the now c then A else B agent (called conditional agent) is the new agent (w.r.t. cc) which allows us to describe notions such as *timeout* or *preemption*. This agent executes A if the store entails c, otherwise it executes B.

2.1 The tccp Structure

In [9], we provided a model for tccp programs which essentially consists of a graph structure. The main difference w.r.t. a Kripke structure is in the definition of the states. A state in a Kripke Structure consists of a valuation of the system variables, whereas in a tccp Structure, states are represented by (conjunctions of) constraints which represent a set of possible valuations of systems variables. In other works, a state of a tccp Structure represents a set of states of a Kripke Structure. In [9], the interested reader can find a formal definition of tccp Structures and a method to automatically build it from a given tccp program.

2.2 The Scheduler Example

In Figure 1 we show an example of tccp program which consists of a predicate with three output variables. We use streams to simulate the values of the system variables along the time. Intuitively, the program gets the value of variables `D1`, `T1` and `E1` by calling the process `get_constraints`. These variables represent the duration of three different tasks of the process of building a house. In parallel, an ask agent simply checks if the values of the variables are isntanciated to integer numbers and, in that case, some constraints are added to the global store. Finally, a recursive call to the building process is made which allows us to recalculate the planning schedule.

The tccp Structure associated with this code is shown in Figure 2. The black circle indicates the initial state of the graph. We have simplified the structure by

```
build([PD|PD_],[PT|PT_],[PE|PE_]) ::=
     ∃ D1,T1,E1 (get_constraints(D1,T1,E1) ||
     ask(atom(D1),atom(T1),atom(E1)) → (tell(PD+D1 =< PT) ||
        tell(PT+T1 =< PE) ||tell(PE+E1 =< PA)) || build(PD_,PT_,PE_)).
```

Fig. 1. Example of a tccp program

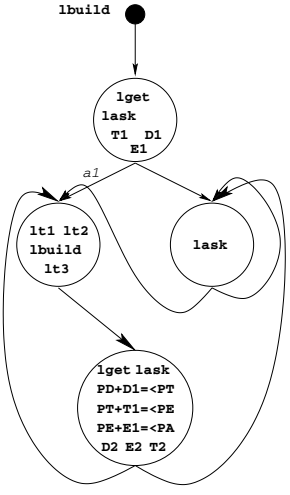

Fig. 2. tccp Structure of `build`

showing, in each state, only the new information added to the store. Informally, labels are used to identify the point of execution of the program. Each occurrence of every agent of a program is labelled, thus the set of labels in a given state represents the set of agents that must run in such execution point (see [9]).

The most important point of this example is the fact that we have added to the store only constraints of the form `V1+C=<V2` which can also be written as `V1-V2`$\leq$`C` being `C` an integer or real constant. This kind of constraints appears in applications where, for example, we compare two clocks of a system to control the timing between tasks, or in scheduling applications such as this example. In the following sections, we show how we can symbolically represent this kind of constraints in a similar way as *Binary Decision Diagrams* (BDDs, [5]) do in the basic symbolic model checking approach.

3 Symbolic Model Checking

The idea of symbolic model checking is to represent the graph structure (the model) as a boolean formula, and then transform it into the efficient structure of BDD [5]. In our approach, we aim to represent the tccp Structure as a formula with difference constraints and logical streams, and then transform it into a suitable extension of BDDs.

In [9], we already developed a preliminary model checker for tccp which uses constraints to achieve a compact representation of the system. Unfortunately, the expected state-explosion problem shows up when we combine the model with the property that we want to verify.

By considering the constraint system described in Section 2 for the tccp language, the tccp Structure which can be automatically obtained by following [9] only contains difference and stream constraints. Thus, in this work, our main idea is to represent that tccp Structure by means of a new symbolic structure (called DDD+LSs). Then, we extend to the new structure the existing efficient algorithms for manipulating DDDs [10] in order to verify tccp programs.

3.1 tccp *Structures* as Logic Formulas

A tccp *structure* can be translated into a formula of the logic underlying our constraint system similarly as it is done in the classical symbolic approach. The key idea is both to encode states by means of a logic formulae, and to represent the transition relation of the graph (i.e., the arcs of the graph) also with a logic formula which is defined from the labels of the nodes. Once we have the formula, we can construct a symbolic BDD-like structure corresponding to that formula, which represents an encoding of the system.

Let us explain how to obtain the formula by using the graph example shown in Figure 2. First, we can encode each arc as a conjunction of constraints. For example, the formula

$$\mathtt{lget} \wedge \mathtt{lask} \wedge \mathtt{T1} \wedge \mathtt{D1} \wedge \mathtt{E1} \wedge \mathtt{lt1}' \wedge \mathtt{lt2}' \wedge \mathtt{lbuild}' \wedge \mathtt{lt}' \qquad (1)$$

represents the arc labelled with $a1$. In the following, we call *arc-formula* the logic formula representing an arc of the tccp structure. Note that we have used primed versions of agent labels to express their value in the following time instant.

Each arc of the graph corresponds to an element in the transition relation R. Then it is easy to see that the R relation can be represented by a disjunction of arc-formulas. The resulting formula is the input for the next task, where we symbolically represent it by means of the new structure (similar to BDDs). We define the algorithms that automatically construct such model from the formula.

3.2 The Symbolic Structure

In order to correctly represent tccp structures, we cannot directly use simple boolean structures such as BDDs, but the more sophisticated Difference Decision Diagrams (DDDs, [10]). The main reason for this is that nodes in DDDs may contain constraints (as states of the tccp structure) which can encode some implicit information whereas nodes in BDDs can contain only boolean variables.

DDDs are an extension of the BDDs to symbolically represent *difference constraint expressions*. Difference constraint expressions are formulas of a logic extended with difference constraints. Difference constraints are inequalities of the form $x - y \leq c$ where x and y are integer or real-valued variables, and c is a constant. A difference constraint expression consists of difference constraints combined with boolean connectives.

DDDs and BDDs share some common features. Both BDDs and DDDs can be ordered and reduced, and the algorithms to handle them are quite similar. A drawback of DDDs is that maintaining them as a canonical data structure is

more expensive than for BDDs. Actually, if we reduce a DDD following the ideas of BDDs, then we do not obtain a canonical representation for the considered difference constraint expression, as opposed to the case of BDDs. However, it is still possible to obtain a semi-canonical[4] structure which can be used to decide satisfiability, validity, falsifiability and unsatisfiability of expressions.

Even though we can use DDDs to represent difference constraints, we need to model also constraints over streams (modeled as logical lists in tccp). Therefore, we need to extend the expressivity of DDDs and consistently redefine the algorithms which automatically construct the DDD Structure from a given formula.

Extending Difference Decision Diagrams with Logical Streams. Similarly to BDDs, *Difference Decision Diagrams + Logical Streams* (DDD+LSs) are directed acyclic graphs designed to handle the following logic:

$$\phi ::= x - y \leq c \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x.\phi \mid X = [x|Y] \mid X = [c|Y] \mid X = [\,]$$

where the constant c belongs to $\mathbb{D}$, and $X, Y \in \mathcal{V}$ denote variables. The grammar is extended as usually with the derived operators $x - y < c$, $\phi_1 \vee \phi_2$ and $\forall x.\phi$. Note that this logic is similar to the constraint system considered in this work. For the interested reader, a formal description of DDD+LSs is given in [1].

The key idea of this construction is that a node of a DDD+LS structure represents an expression which can be either a difference constraint or a stream constraint. Moreover, two arcs go out from each non-terminal node modeling the cases when the constraint represented by the node is satisfied or not. In Figure 3 (a), we show a DDD+LS structure representing the following formula.

$$\mathtt{PD} - \mathtt{PT} =< 4 \wedge \mathtt{PT} - \mathtt{PE} =< 7 \wedge \mathtt{P} = [\mathtt{PT} \mid \mathtt{PT\_}] \tag{2}$$

In order to obtain an ordered graph structure which considers the new attributes of DDD+LS, we extend in the natural way the standard total order on the vertices of the graph defined in [10]. Intuitively, nodes containing difference expressions will appear in the graph structure before nodes containing stream expressions. The resulting formal definition of *Ordered* DDD+LS (called ODDD+LS in short) can be found in [1].

Following [10], we can consider semi-canonical structures to verify some properties. To get them, we apply some local and path reductions for ODDD+LSs, which are convenient extensions of the reductions defined for Ordered DDDs.

Roughly speaking, we first apply the local reduction which replaces constraints of the form $x < y$ by $x \leq y - 1$. Then, nodes which can be considered identical are eliminated, thus each node of the resulting Locally Reduced DDD+LS (LRDDD+LS) is different from the others.

The next step towards a suitable semi-canonical representation of DDD+LSs is the formulation of path reduction. The LRDDD+LS structure which results

[4] A DDD is semi-canonical if (i) an expression ϕ is represented by **1** iff ϕ is valid, and (ii) an expression ϕ is represented by **0** iff ϕ is unsatisfiable.

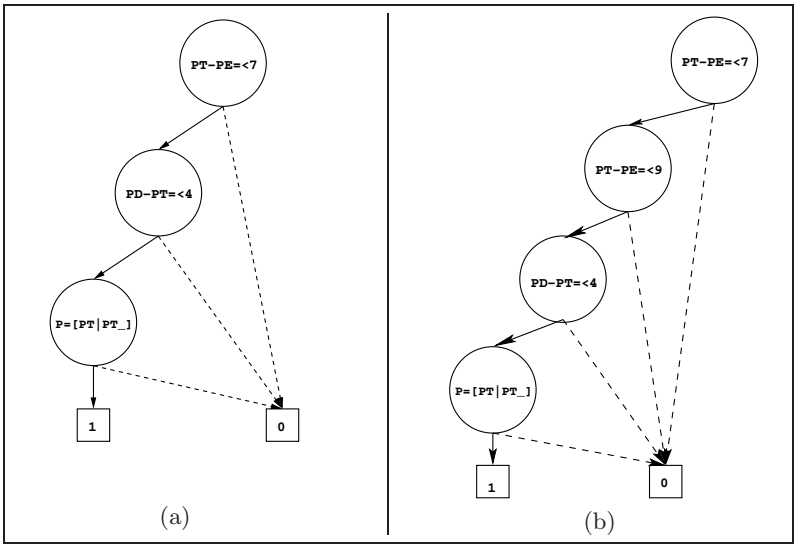

Fig. 3. Example: DDD+LS from the formula in (2)

from a path reduction step is called PRDDD+LS. Essentially, we can identify redundant arcs regarding difference constraint expressions by checking how expressions divide the domain. Each arc splits the domain into two disjoint subsets. If one of these subsets is empty, then we know that the arc is redundant.

In Figure 3 (b) we show a DDD+LS structure representing the formula in (2), which has a redundant node (the second one from the top). It is redundant since the part of the domain for which the constraint is not satisfied is empty. Thus we could eliminate it obtaining the DDD+LS shown in Figure 3 (a).

Theorem 1 below allows us to check properties in the PRDDD+LS in a safe way. We know that the expression represented by the node u is valid if and only if $u = \mathbf{1}$. If $u = \mathbf{0}$, then the expression is unsatisfiable. If u is a non terminal vertex, then we know that the expression is both satisfiable *and* falsifiable. The proof of this result can be found in [1].

Theorem 1 (semi-canonicity). *In a PRDDD+LS, the terminal vertex* ***1*** *is the only representation of valid expressions, and the terminal vertex* ***0*** *is the only representation of unsatisfiable expressions.*

4 Construction of DDD+LSs

In the present section, we provide the algorithms which automatically construct a LRDDD+LS structure from a given formula. From now, we assume that we are always considering LRDDD+LS. Vertices and arcs of the DDD+LS are stored in a graph data structure simply called *Graph*. Let G be a *Graph*. Initially, G

```
vertex MkD(G: graph, x ∈ V, y ∈ V, o: operator, c ∈ D, h: vertex, l: vertex)
  if D = Z ∧ o =′< then c := c − 1; o :=′ ≤′
  if member(G, (x, y, o, c, ⊥, ⊥, ⊥, h, l)) then
    return lookup(G, (x, y, o, c, ⊥, ⊥, ⊥, h, l))
  else if l = h then return l
       else if (x, y) = var(l) ∧ h = high(l) then return l
            else return insert(G, (x, y, o, c, ⊥, ⊥, ⊥, h, l))

vertex MkL(G: graph, x ∈ LV, y ∈ V, z ∈ LV, o: operator, h: vertex, l: vertex)
  if member(G, (⊥, ⊥, LIST, ⊥, x, y, z, h, l)) then
    return lookup(G, (⊥, ⊥, LIST, ⊥, x, y, z, h, l))
  else if l = h then return l
       else if (x, z) = var(l) ∧ h = high(l) then  return l
            else return insert(G, (⊥, ⊥, LIST, ⊥, x, y, z, h, l))
```

Fig. 4. Algorithms MkD and MkL that create vertices

contains only the two terminal vertices **0** and **1**. The set of arcs of G are implicitly stored via the attributes of its vertices.

Let us introduce some functions which allow us to access the information or modify the structure. First, the function $\mathsf{insert}(G, a)$ creates a new vertex v in G with attribute a, and returns v. The function $\mathsf{member}(G, a)$ returns true if there exists a vertex in graph G with attribute a. Finally, $\mathsf{lookup}(G, a)$ returns the vertex in G with attribute a.

In the following, we extend the algorithms defined in [10] for DDDs, to construct and handle the DDD+LS structures. We introduce some notation: *var(n)* represents the variables of the constraint, and *high(n)* (*low(n)*) represents the left-successor (right-successor) of node n.

In Figure 4, the algorithm MkD for difference constraints is given as a suitable extension of the algorithm presented in [10]. Also the algorithm MkL is presented which builds the vertex representing the stream expression $x{=}[y|z] \rightarrow h, l$.

The next step for the construction of the DDD+LS structure is to define the algorithms which combine difference and stream expressions with boolean operators. The idea is to recursively apply a specific operator to all the vertices in the DDD+LS Structure. In [5], this procedure is called Apply. The same idea can be used for our DDD+LS structure.

We have called AppLS the corresponding algorithm for DDD+LSs. We show in Figure 5 this algorithm which follows closely the design of Apply with some suitable adjustment to include the handling of the list expressions. In the pseudocode, 'Connective' denotes a boolean connective of the logic, whereas eval is a function which takes the two terminal vertices and a boolean connective as input and returns the truth value depending on the boolean connective. Additional notation is used in this algorithm: *op(n)* tells us which kind of constraint the node represents, whereas *bound(n)* states which kind of relation there exists between variables of the constraint. Moreover, *head*, *tail*, and *left* represent the different components of a list constraint.

Vertex AppLS(G: graph c: Connective, u: Vertex, v: Vertex)
r: Vertex
if $u, v \in \{\mathbf{0},\mathbf{1}\}$ **then return** eval(c, u, v)
else if member$(G, (c, u, v))$ **then return** lookup$(G, (c, u, v))$
  else if $var(u) < var(v)$ **then**
    if $op(u) = \text{LIST}$ **then** $r \leftarrow$ MkL$(left(u), head(u), tail(u),$ AppLS$(c, high(u), v),$
                                AppLS$(c, low(u), v))$
      else $r \leftarrow$ MkD$(var(u), bnd(u),$ AppLS$(c, high(u), v),$ AppLS$(c, low(u), v))$
      return r
  else if $var(u) = var(v)$ **then**
    if $bnd(u) < bnd(v) \wedge op(u) = \text{LIST}$ **then** $r \leftarrow$ MkL$(left(u), head(u), tail(u),$
                                AppLS$(c, high(u), high(v))$, AppLS$(c, low(u), v))$
    else if $bnd(u) < bnd(v) \wedge op(u) \neq \text{LIST}$ **then**
      $r \leftarrow$ MkD$(var(u), bnd(u),$AppLS$(c, high(u), high(v))$,AppLS$(c, low(u), v))$
    else if $bnd(u) = bnd(v) \wedge op(u) = \text{LIST}$ **then** $r \leftarrow$ MkL$(left(u), head(u), tail(u),$
                                AppLS$(c, high(u), high(v))$, AppLS$(c, low(u), low(v)))$
    else if $bnd(u) = bnd(v) \wedge op(u) \neq \text{LIST}$ **then** $r \leftarrow$ MkD$(var(u), bnd(u),$
                                AppLS$(c, high(u), high(v))$, AppLS$(c, low(u), low(v)))$
    else if $bnd(u) > bnd(v) \wedge op(v) = \text{LIST}$ **then** $r \leftarrow$ MkL$(left(v), head(v), tail(v),$
                                AppLS$(c, high(u), high(v))$, AppLS$(c, u, low(v)))$
    else if $bnd(u) > bnd(v) \wedge op(v) \neq \text{LIST}$ **then**
      $r \leftarrow$ MkD$(var(u), bnd(u),$ AppLS$(c, high(u), high(v))$,AppLS$(c, u, low(v)))$
    else if $var(u) > var(v)$ **then**
      if $op(u) = \text{LIST}$ **then** $r \leftarrow$ MkL$(left(v), head(v), tail(v),$
                                AppLS$(c, u, high(v))$, AppLS$(c, u, high(v)))$
      else $r \leftarrow$ MkD$(var(v), bnd(v),$ AppLS$(c, u, high(v))$, AppLS$(c, u, high(v)))$

Fig. 5. Algorithm AppLS

5 Verification

In this section, we show how the symbolic structure can be used to formalize a symbolic model checking method for tccp programs. Assume that we express the property that we want to verify by using a CTL logic [6], where the atomic propositions of the logic are the same set of atomic propositions of the constraint system considered above. Note that we can use the corresponding entailment relation to obtain the truth value of formulas (see [4]).

We illustrate the method by an example. Assume that we want to verify that whatever state we check where the variables have been initialized, there exists a successor state where the same test succeeds. This property is expressed by the formula (3). We use the standard notation for temporal operators, thus $\mathbf{AG}(f)$ is the logic operator meaning that the formula f holds at each state in the future and $\mathbf{EX}(g)$ means that there exists a successor state where g is satisfied.

$$\mathbf{AG}(\neg \texttt{lask} \vee \mathbf{EX}(\texttt{lask})) \tag{3}$$

The classical symbolic model checking algorithm would take this formula as input and would return an OBDD representing the set of states of the system satisfying that formula. Temporal operators of the logic are represented as fix-points [6] and then, symbolic structures are manipulated. In our approach we would substitute OBDDs by DDD+LSs and the CTL logic by the temporal logic of [4] which is interpreted over constraints.

[6] shows that is possible to associate a fix-point operator to each CTL formula which obtains the set of states starting from which the property holds. Since the formula $\mathbf{AG}(f)$ is equivalent to $f \wedge \mathbf{AX}(f)$, where $\mathbf{AX}$ means that the formula holds at each successive state, then it suffices to consider the operator associated to $f \wedge \mathbf{AX}(f)$. This operator allows us to compute a (greatest) fix-point which corresponds to the set of states starting from which the property to be proven holds. Finally if all initial states of the model (the tccp Structure) are included in the fix-point, then the formula is proven to hold in the system. In our example, the resulting algorithm proves that the formula holds.

6 Conclusions

We have generalized DDDs to a new structure which allows us to represent tccp programs symbolically. We have introduced the corresponding notions and algorithms for automatically construct the symbolic structures and we have shown how they can be used to formulate a lightweight, symbolic model checking algorithm. This novel symbolic methodology improves the automatic verification of reactive systems specified by using tccp as it reduces the search space significantly.

As future work, we plan to extend the language to consider constraint expressions more general than difference constraints. We also plan to complete and make publicly available a very cheap implementation of our method that we have already used for a preliminary evaluation of the methodology proposed in the paper over a small set of examples.

References

1. M. Alpuente, M. Falaschi, and A. Villanueva. Symbolic Representation Timed Concurrent Constraint Programs. Technical Report DSIC-II/12/04, DSIC, Technical University of Valencia, 2004. Available at www.dsic.upv.es/users/elp/villanue/papers/techrep04.ps.
2. M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. Abstract Model Checking of tccp programs. In *Proc. of the 2nd Workshop on Quantitative Aspects of Programming Languages (QAPL 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2004.
3. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.
4. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In G. Smolka, editor, *Proc. of 8th International Symposium on Temporal Representation and Reasoning*, pages 227–233. IEEE Computer Society Press, 2001.

5. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
7. E. M. Clarke, K. M. McMillan, S. Campos, and V. Hartonas-GarmHausen. Symbolic Model Checking. In *Proc. of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer-Verlag, July/August 1996.
8. M. Falaschi, A. Policriti, and A. Villanueva. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language - I. In A. Dovier, M. C. Meo, and A. Omicini, editors, *Selected papers from 2000 Joint Conference on Declarative Programming*, volume 48 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.
9. M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Programming*, 2004. To appear.
10. J. Møller, J. Lichtenberg, H.R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proc. of the 13th International Workshop on Computer Logic Science*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, 1999.
11. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proc. of 18th Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, New York, 1991. ACM Press.