Symbolic Representation of tccp Programs*

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

[1] DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain. {alpuente,villanue}@dsic.upv.es.
[2] Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. falaschi@dimi.uniud.it.

Abstract In this paper, we develop a symbolic representation for *timed concurrent constraint* (tccp) programs, which can be used for defining a new 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. In particular, 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:** Timed Concurrent Constraint Programming, Model Checking, 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, automatic verification tools which are able to guarantee 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.

The *concurrent constraint* paradigm (cc) was first introduced in [16] 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 [2] 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 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 ([9,10,19]) 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. Unfortunately, we are able to verify only small programs due to the explicit exploration of the graph.

Recent advances in model checking deal with huge state-spaces by using symbolic manipulation algorithms inside model checkers [5,7,13]. Other techniques such as abstract interpretation, partial evaluation, and on-the-fly methods have also been proposed in the literature as a mean to (partially) solve the state-space explosion problem [6].

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 [10], in this paper we formalize a symbolic representation of reactive systems specified in tccp. Such representation allows us to formulate a symbolic model checking algorithm which allows us to verify more complex reactive systems in 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 requiring the user to indicate a finite time interval for limiting the duration of tccp computations, as we did in [9,19]. This idea could be extended also to our new framework. To the best of our knowledge, we define the first symbolic model checking algorithm for tccp.

* This research is partially supported by the MCyT under grants TIC2001-2705-C03-01 and HU 2003-0003.

The paper is organized as follows. In Section 2 we introduce the tccp programming language and the tccp Structure constructed from the program specification and which is the reference point of this work. We introduce also an example which is used in the remaining sections to illustrate formal definitions. In Section 3 we introduce the verification method that we propose and in Section 4 we define the technical mechanisms that we need to apply the verification method. In particular we introduce the symbolic structure used to represent tccp programs. In Section 5 we show the algorithms that allow us to automatize the construction process and finally, in Section 6 we develop an example of property verification. Section 7 is devoted to conclusions and future work.

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.

2.1 The tccp language

The *Timed Concurrent Constraint Language* (tccp) was developed in [2] 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 [16,18] 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 cylindric constraint system as it is used in the cc paradigm. A simple constraint system can be defined as a set of tokens (or primitive constraints) together with an entailment relation. Examples of such constraint systems are the Herbrand constraint system, the FD constraint system [12] and the Gentzen constraint system [17].

Definition 1 (Simple constraint system [18]). Let D be a non-empty set of *tokens* (primitive constraints). A *simple constraint system* is a structure $\langle D, \vdash \rangle$ where $\vdash \subseteq 2^D \times D$ is an *entailment relation* satisfying:

C1 $u \vdash P$ whenever $P \in u$,
C2 $u \vdash Q$ whenever $u \vdash P$ for all $P \in v$ and $v \vdash Q$.

A *cylindric* constraint system consists of a simple constraint system plus an existential quantification operator which is monotonic, conservative and supports renaming. The existential quantification allows one to model local variables in a given agent. The formal definition of the notion of *cylindric constraint system* can be found in [2,11].

In this work, we consider a specific constraint system which allows us to verify a class of software systems. In particular, we consider the traditional arithmetic for real numbers including addition, equality and order comparison. This part of the constraint system handles the information related to the constrained nature of the system. On the other hand, we are also interested to handle streams which in tccp are modeled as lists of terms. Each stream represents the value of a given system variable along the time. Intuitively, in the current time instant, the head of the list represents the

[1] In practice, some syntactic restrictions are imposed in order to ensure that these hypotheses are reasonable (see [2] for details).

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, $[X|Z] = [a|Y]$ [2] 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 the set of *difference constraints* of the form $X - Y \leq c$ and $X - Y < c$, and 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 [2] as follows:[3]

Definition 2 (tccp Language). Let C be a cylindric constraint system. The syntax of agents of the language is given by the following grammar:

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

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 $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ 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.

2.2 The tccp Structure

The reference point of this work is a model of tccp programs introduced in [10], 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 tccp Structure represents a set of states of a Kripke Structure since it contains a conjunction of constraints instead of a valuation of the system variables. Formally, a tccp Structure is as follows. In [10] the reader can find how to automatically obtain the tccp Structure from a given tccp program.

Definition 3 (tccp *structure*). Let AP be a set of atomic propositions. We define a tccp Structure M over AP as a 5-tuple $M = (S, S_0, R, C, T)$, where

1. S is a finite set of states,
2. $S_0 \subseteq S$ is the set of initial states,
3. $R \subseteq S \times S$ is a transition relation,
4. $C : S \rightarrow 2^{AP}$ is the function that returns the set of atomic propositions which hold in a given state, and
5. $T : S \rightarrow 2^{L}$ is the function that returns the set of labels in a given state.

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.

[2] We follow the Prolog notation for lists.
[3] The operational and denotational semantics of the language can be found in [2].

2.3 The scheduler example

In Figure 1 we show a tccp program which we use to motivate different points of the paper. The program consists of a predicate with three output variables. We use streams to simulate the values of the system variables along the time, since the constraint system in tccp is monotonic (see [2] for details).

Intuitively, the program gets the value of variables D1, T1 and E1 by calling the auxiliary process get_constraints. These variables represent the duration of three different tasks of the process of building a house. This is executed in parallel with an ask agent which simply checks if the values of the variables are integer numbers and, in that case, some constraints are added to the global store which contains the available information of the system. Finally, a recursive call to the building process is made which would allow to recalculate the planning schedule.

```
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_)).
```

Figure 1. Example of a tccp program

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 showing, in each state, only the new information added to the store. At each state, we also show the set of labels (beginning with the character 'l') representing the agents that must be executed, and the local variables.

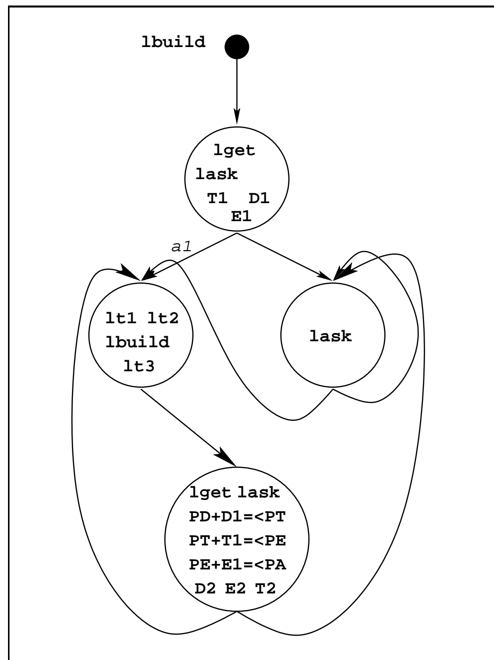

Figure 2. tccp Structure of build

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) 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. 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 [10,19], a logic dealing with constraints was proposed as the basis to develop a classical LTL model checking algorithm based on a tableau algorithm. The most important advantage of this approach is that the use of constraints leads to a compact representation of the system which we also exploit to effectively check properties on the model. 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 defined in Section 2.1 for the `tccp` language, the model which can be automatically obtained by following [10,19] only contains difference and stream constraints. Thus, our aim is to represent the `tccp` *Structure* by means of a new symbolic structure called DDD+LSs, then we use the efficient algorithms for checking DDD+LSs in order to verify `tccp` programs. In the following, we formalize our verification strategy and illustrate it by means of an example. We use the temporal logic with constraints of [3] to specify the properties we are interested to verify.

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 for boolean functions in the classical symbolic approach. The idea is to encode states and to represent the relation R (i.e., the arcs of the graph) with a logic formula which is defined from the labels and the store of states.

Once we have the formula, we can construct a symbolic BDD-like structure corresponding to the 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. Each arc of this graph corresponds to an element in the relation R. Now we can encode each arc as a conjunction of constraints. For example, the formula

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

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 in order to express their value in the following time instant. This is equivalent to the use of program counters in (imperative) classical model checking approaches.

It is easy to see that the R relation can be represented by a disjunction of arc-formulas. The resulting formula is the subject of our next task: we have to symbolically represent this formula and, for this purpose, we define a new structure (similar to a BDD) and the algorithms which automatically construct it from the formula.

4 The Symbolic Structure

Difference Decision Diagrams (DDDs) are an extension of the Binary Decision Diagrams defined in [4] 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. $d \to a, b$ where d is a difference constraint and a and b are difference constraint expressions means that, if d holds, then a, else b.

DDDs and BDDs share some common features. For example, both BDDs and DDDs can be ordered and reduced, and the algorithms to handle them are quite similar. A drawback of DDDs is the fact that maintaining them as a canonical data structure is more expensive than for BDDs.

It is important to remark that, in order to correctly represent `tccp` Structures, we cannot directly use boolean structures such as BDDs. Nodes in DDDs contain constraints which can encode some implicit information whereas nodes in BDDs contain only boolean variables [14]. This implicit information is the main reason why a DDD-like structure is necessary. This is also the reason why, 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 Ordered 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. There is also an algorithm to obtain a canonical representation of DDDs which is quite expensive ([15]).

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

4.1 Extending Difference Decision Diagrams with Logical Streams

As we have shown in Section 3, we need to represent symbolically a graph structure which contains difference constraints and stream constraints.

Formally, we define *Difference Decision Diagrams + Logical Streams* (DDD+LSs) to handle the logic defined by the following grammar:

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

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 $\forall x.\phi$. Similarly to DDDs, equality can be modeled by using the $<$ and $\leq$ operators.

Similarly to DDDs, a DDD+LS is a directed acyclic graph (V, E) where V is a set of vertices and E a set of arcs connecting pairs of vertices. The set V contains two terminal vertices with out-degree zero (called **0** and **1**). In addition, V contains a set of non-terminal vertices with out-degree two. Each non-terminal vertex v has nine attributes which can be classified in three subsets: (i) the first three attributes ($pos(v), neg(v)$ and $const(v)$) which are defined in the case when the node v represents a difference constraint (otherwise they are set to $\perp$), (ii) the attributes $left(v)$, $head(v)$ and $tail(v)$ that stand for the case when v represents a stream constraint (otherwise they are set to $\perp$), and (iii) the last three attributes $op(v)$, $high(v)$ and $low(v)$ which are defined in both cases. Intuitively, $op(v)$ determines which kind of expression represents v: if $op(v) \in \{\text{LE}, \text{LEQ}\}$, then v represents the expression $pos(v) - neg(v)\ op(v)\ const(v)$; otherwise (if $op(v) = \text{LIST}$), then v represents the stream constraint $left(v) = [head(v)|tail(v)]$ The remaining two attributes ($high(v)$ and $low(v)$) represent the two branches that can be followed from the non-terminal vertex v in the graph.

Some shorthands are defined to reference combinations of attributes. Notation $var(v)$ represents the pair $(pos(v), neg(v))$ whereas we use $bnd(v)$ to refer to the pair $(op(v), const(v))$. By $varl(v)$ we represent the pair $(left(v), tail(v))$ and $listExp(v)$ is the pair $(head(v), varl(v))$. Finally, we denote by $attr(v)$ the set of attributes of the node v.

The set of edges E is defined as the set of pairs of the form $(v, low(v))$ and $(v, high(v))$, where $v \in V$ and v is not a terminal vertex.

[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.

A node of a DDD+LS Structure represents an expression which can be either a difference constraint (as in DDDs) or a stream constraint. The semantics of DDD+LS nodes is formalized in Definition 4 which just adds to DDDs the semantics derived from the new DDD+LS attributes. **Exp** stands for difference constraint expressions and stream expressions. The auxiliary function $(op(v))$ is used to check whether the node represents a difference constraint expression (returning values LE or LEQ), or a list expression (returning value LIST).

Definition 4. Let v be a vertex of a DDD+LS Structure. We define the function $\mathcal{S} : V \to \mathbf{Exp}$:

$$\mathcal{S}[\![\mathbf{0}]\!] \stackrel{\text{def}}{=} \mathit{false}$$
$$\mathcal{S}[\![\mathbf{1}]\!] \stackrel{\text{def}}{=} \mathit{true}$$
$$\mathcal{S}[\![v]\!] \stackrel{\text{def}}{=} \begin{cases} (pos(v) - neg(v) < const(v)) \to \mathcal{V}[\![high(v)]\!], \mathcal{V}[\![low(v)]\!] & \text{if } op(v) = \text{LE}, \\ (pos(v) - neg(v) \leq const(v)) \to \mathcal{V}[\![high(v)]\!], \mathcal{V}[\![low(v)]\!] & \text{if } op(v) = \text{LEQ}, \\ (left(v) = [head(v) | tail(v)]) \to \mathcal{V}[\![high(v)]\!], \mathcal{V}[\![low(v)]\!] & \text{if } op(v) = \text{LIST} \end{cases}$$

For example, the meaning of the first row in the semantics of v means that, if $pos(v) - neg(v) < const(v)$ holds, then $\mathcal{V}[\![high(v)]\!]$ is true, otherwise $\mathcal{V}[\![low(v)]\!]$ holds.

We show in Figure 3 a DDD+LS graph representing the formula in (2).

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

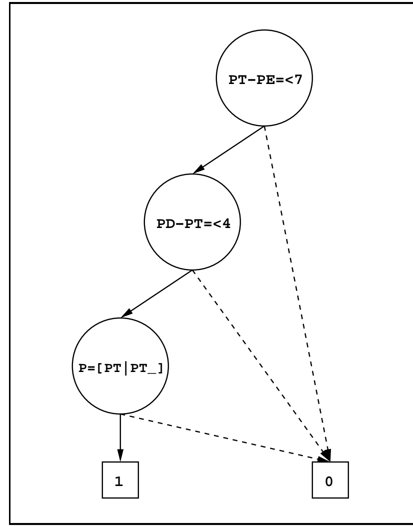

Figure 3. Example: DDD+LS from the formula in (2).

In order to obtain an ordered graph structure, we extend the total order on the vertices of the graph defined in [14] to consider the new attributes in DDD+LS. First of all, we assume an order between variables. We require that both, pairs of variables of a vertex corresponding to the difference expression, as well as pairs of list variables in the list expression, are *normalized.* This means that $pos(v) > neg(v)$ and $left(v) < tail(v)$[5]. Then we assume that LIST < LE < LEQ. Finally, tuples formed by the set of attributes in a specific vertex u, i.e., tuples of the form $(pos(v), neg(v), op(v), const(v), left(v),$
$head(v), tail(v))$, are ordered lexicographically. Note that $\forall X, \bot \leq X$ where $X \in \{\mathcal{V}, \mathcal{LV}\}$. We also have that for any constant c, $\bot \leq c$.

[5] We note that this property is reasonable when we use this kind of constraints to model streams, as it simply establishes that the list on the right hand side of the constraint is greater than the other one.

By using the order $<$ above, the notion of *Ordered* DDD+LS (called ODDD+LS in short) is formalized:

Definition 5 (Ordered DDD+LS). Let O be a DDD+LS Structure. We say that O is an *Ordered DDD+LS* (ODDD+LS) Structure if each non-terminal vertex v defined in O satisfies the following properties:

1. $neg(v) < pos(v)$,
2. $left(v) < tail(v)$,
3. $var(v) < var(high(v))$
4. $varl(v) < varl(high(v))$,
5. $var(v) < var(low(v)) \vee (var(v) = var(low(v)) \wedge bnd(v) < bnd(low(v)))$,
6. $varl(v) < varl(low(v)) \vee (varl(v) = varl(low(v)) \wedge head(v) < head(low(v))$

Intuitively, nodes containing difference expressions will appear in the graph structure before the nodes containing stream expressions. The first two conditions in Definition 5 require that variables in a given node are normalized. The third and fourth requirements establish that, given a node v, variables of the child in the high branch of v must be greater than the variables in v. The last two points ensure that variables of the child in the low branch must be greater or equal than the variables of v; if they are equal, then $bnd(low(v))$ $(head(low(v)))$, must be greater than $bnd(v)$ $(head(v))$, respectively. The structure in Figure 3 is an ODDD.

In order to verify properties, we can consider semi-canonical structures as mentioned before. To get them, we propose some local and path reductions for ODDD+LSs, which are inspired in the reductions defined for Ordered DDDs.

Definition 6 (Locally Reduced DDD). Let D be an ODDD+LS with domain $\mathbb{D} \in \{\mathbb{N}, \mathbb{Z}\}$, and let u and v be non-terminal vertices of D. Then D is a *Locally Reduced DDD+LS* (LRDDD+LS) if it satisfies:

1. if $\mathbb{D} = \mathbb{Z}$ then, for all v, $op(v) = \text{LEQ}$ or $op(v) = \text{LIST}$,
2. for all v and u, if the set of attributes of u is identical to the set of attributes of v, then $u = v$,
3. for all v, $low(v) \neq high(v)$,
4. for all v, if $var(v) = var(low(v))$ then $high(v) \neq high(low(v))$,
5. for all v, if $list(v) = list(low(v))$ then $high(v) \neq high(low(v))$

The intuition of the first item is that, if the domain of the structure are integers, then we can eliminate any occurrence of the LE operator since it can be reduced to the LEQ operator by decreasing in one unit the value of the constant and then comparing with LEQ. The second condition ensures that there is no pair of different nodes with the same attributes. The rest of requirements avoid redundant tests on the same variables.

The next step towards the semi-canonical representation of DDD+LSs is the formalization of the notion of path reduction. We first need to define the semantics of edges and paths. By abuse, we define the negation of lists as the absence of information. That is, when we negate a stream expression we mean that the current store does not entail it.

Definition 7. Let u, v be vertices of a DDD+LS Structure. Let u and v be two adjacent vertices. The function $\mathcal{E} : E \to \mathbf{Exp}$ is defined as follows:

$$\mathcal{E}[\![(u,v)]\!] \stackrel{\text{def}}{=} \begin{cases} (pos(u) - neg(u) < const(u)) & \text{if } v = high(u) \text{ and } op(v) = \text{LE}, \\ (pos(u) - neg(u) \leq const(u)) & \text{if } v = high(u) \text{ and } op(v) = \text{LEQ}, \\ \neg(pos(u) - neg(u) < const(u)) & \text{if } v = low(u) \text{ and } op(v) = \text{LE}, \\ \neg(pos(u) - neg(u) \leq const(u)) & \text{if } v = low(u) \text{ and } op(v) = \text{LEQ} \\ left(v) = [head(v)|tail(v)] & \text{if } v = high(u) \text{ and } op(v) = \text{LIST}, \\ \neg(left(v) = [head(v)|tail(v)]) & \text{if } v = low(u) \text{ and } op(v) = \text{LIST}. \end{cases}$$

The notion of *path* in a DDD+LS Structure is defined as a finite sequence of edges of the form $\langle (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$. We say that such path has length k. The semantics of a path is defined as the conjunction of all difference constraints, negated difference constraints, stream constraints, and negated stream constraints in the path.

Now we are ready to define the notion of path reduction which provides us the semi-canonical representation. We denote by PRDDD+LS the structure resulting from applying this reduction step to a LRDDD+LS. A semi-canonical representation has exactly one DDD+LS to denote valid expressions and also a single DDD+LS for unsatisfiable expressions.

Essentially, we can identify redundant edges regarding difference constraint expressions by checking how expressions divide the domain. Each edge e_i splits the domain into two disjoint subsets. If one of these subsets is empty, then we know that the edge is redundant. This is the method applied in [14]. Regarding nodes representing stream expressions, since we have defined the negation as the absence of information, then the domain is split into two *possibly non* disjoint subsets, and no path-reduction can be done.

Theorem 1 allows us to check properties in the PRDDD+LS in a safe way. We know that the expression ϕ_u 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.*

In Figure 4 we show a DDD+LS Structure representing the same formula in (2), which has a redundant edge (the second one from the top). We know that this node 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

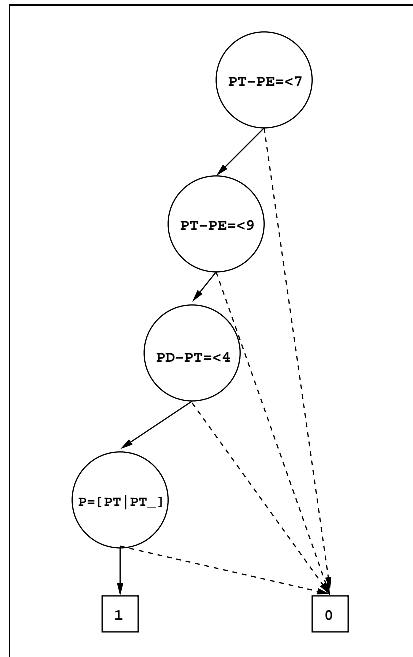

Figure 4. Example: Non Path-reduced DDD+LS representing the formula in (2).

5 Construction of DDD+LSs

In this section we show the algorithms which automatically construct a Locally Reduced DDD+LS from a given formula. In the rest of the section we assume that we are always considering Locally Reduced DDD+LS. Vertices and edges of the DDD+LS are stored in a graph data structure simply called *Graph*. Let G be a *Graph*. Initially, G contains only the two terminal vertices **0** and **1**. The set of edges 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, $\mathsf{insert}(G, a)$ 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 G with attribute a. Finally, $\mathsf{lookup}(G, a)$ returns the vertex in G with attribute a.

We also need some operators which obtain information about the attributes of a graph. Since names and behavior of these operations are very intuitive, we will use the name of attributes directly in our pseudocode.

In the following, we extend the algorithms in [14] (defined to construct DDDs), to construct the DDD+LS Structure. We also develop the new algorithms to deal with stream expressions.

```
vertex MkD(G: graph, x ∈ V, y ∈ V, o: operator,
           c ∈ D, h: vertex, l: vertex)
  if D = Z ∧ o = LE then c := c − 1
                          o := LEQ
  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))
```

Figure 5. MKD Algorithm

In Figure 5, the algorithm MKD for difference constraints is given as an extension of the algorithm presented in [14]. We have modified the attributes of nodes to take into account that nodes in DDD+LSs have three additional attributes.

The algorithm MKL is presented in Figure 6. It builds the vertex representing the stream expression $x = [y|z] \to h, l$.

```
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) = plist(l) ∧ h = high(l) then
            return l
          else
            return insert(G, (⊥, ⊥, LIST, ⊥, x, y, z, h, l))
```

Figure 6. Algorithm MKL that creates a vertex for a list expression

These two algorithms have some preconditions which are similar to those for DDDs. For example, pairs of variables must be normalized. In [14], some functions are given which normalize pairs of variables before constructing vertices. We can use similar procedures for the difference expressions

contained in our DDD+LS Structures. A novel precondition for the MKD algorithm is that we must ensure that we are dealing with difference constraints (namely, $op(v) \neq$ LIST). Similarly, for the MKL algorithm we require that $op(v) =$ LIST.

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 Structure. In [4], this procedure is called APPLY. The same idea can be used for our DDD+LS Structure. The APPLY algorithm returns a DDD which is locally reduced, hence it is still necessary to path reduce the resulting DDD.

We have called APPLYLS the corresponding algorithm for DDD+LSs. We show in Figure 7 this algorithm which follows closely the design of APPLY with some little adjustment to include the handling of the list expressions. In the pseudocode, 'Connective' denotes a boolean connective of the logic. Moreover, 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.

```
Vertex APPLYLS(G: graph c: Connective, u: Vertex, v: Vertex)
r: Vertex
if u, v ∈{0,1} 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) = LIST then
        r ← MKL(left(u), head(u), tail(u),APPLYLS(c, high(u), v), APPLYLS(c, low(u), v))
      else r ← MKD(var(u), bnd(u), APPLYLS(c, high(u), v), APPLYLS(c, low(u), v))
      return r
  else if var(u) = var(v) then
    if bnd(u) < bnd(v) ∧ op(u) = LIST then
      r ← MKL(left(u), head(u), tail(u), APPLYLS(c, high(u), high(v)), APPLYLS(c, low(u), v))
    else if bnd(u) < bnd(v) ∧ op(u) ≠ LIST then
      r ← MKD(var(u), bnd(u), APPLYLS(c, high(u), high(v)), APPLYLS(c, low(u), v))
    else if bnd(u) = bnd(v) ∧ op(u) = LIST then
      r ← MKL(left(u), head(u), tail(u), APPLYLS(c, high(u), high(v)),
            APPLYLS(c, low(u), low(v)))
    else if bnd(u) = bnd(v) ∧ op(u) ≠ LIST then
      r ← MKD(var(u), bnd(u), APPLYLS(c, high(u), high(v)), APPLYLS(c, low(u), low(v)))
    else if bnd(u) > bnd(v) ∧ op(v) = LIST then
      r ← MKL(left(v), head(v), tail(v), APPLYLS(c, high(u), high(v)),
            APPLYLS(c, u, low(v)))
    else if bnd(u) > bnd(v) ∧ op(v) ≠ LIST then
      r ← MKD(var(u), bnd(u), APPLYLS(c, high(u), high(v)), APPLYLS(c, u, low(v)))
    else if var(u) > var(v) then
      if op(u) = LIST then
        r ← MKL(left(v), head(v), tail(v), APPLYLS(c, u, high(v)), APPLYLS(c, u, high(v)))
      else r ← MKD(var(v), bnd(v), APPLYLS(c, u, high(v)), APPLYLS(c, u, high(v)))
```

Figure 7. Algorithm APPLYLS

6 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 defined entailment relation to obtain the truth value of formulas (see [3]).

We illustrate the method by a simple example. Assume that we want to verify that whatever we check that the variables have been assigned, there exists a successor where the same check is done. This property is expressed by the following formula. We use the standard notation for the 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 the formula in (3) 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 is interpreted over constraints.

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. [6] shows that is possible to associate a fix-point operator to each CTL formula. Thus, we 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 holds in the system. In our example, this algorithm [6] proves that the formula holds.

7 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 within a symbolic model checking method. We think that this novel symbolic methodology improves the automatic verification of reactive systems specified as tccp programs 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.

[8] presents a different data structure called CST, which allows one to represent integer linear constraints symbolically and is used to define a parameterized verification method for (infinite state) Petri nets.

We also plan to implement our method and compare it with the exhaustive algorithm defined in previous works in order to quantify the improvement of the symbolic method w.r.t. the exhaustive one.

References

1. M. Alpuente, M. Falaschi, and A. Villanueva. Symbolic Model Checking for Timed Concurrent Constraint Programs. Technical Report DSIC-II/19/03, DSIC, Technical University of Valencia, 2003. Available at www.dsic.upv.es/users/elp/villanue/papers/ techrep03.ps.
2. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.
3. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In G. Smolka, editor, *Proceedings of 8th International Symposium on Temporal Representation and Reasoning*, pages 227–233. IEEE Computer Society Press, 2001.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2002.
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 *Proceedings 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. G. Delzanno, J.-F. Raskin, and L. Van Begin. CSTs (Covering Sharing Trees): compact Data Structures for Parametrized Verification. *Software Tools for Technology Transfer*, 2003. . To appear.
9. 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.
10. M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. 2003. Submitted for publication.
11. L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras Part I*. North–Holland, Amsterdam, 1971.
12. P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University, 1992.
13. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.
14. J. Møller and J. Lichtenberg. Difference Decision Diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, 1999.
15. J. Møller, J. Lichtenberg, H.R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proceedings of the 13th International Workshop on Computer Logic Science*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, 1999.
16. V. A. Saraswat. Concurrent Constraint Programming Languages. In *PhD Thesis, Carnegie-Mellon University*, 1989.
17. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Programming in Timed Concurrent Constraint Languages. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI*, pages 361–410, Berlin, 1994. Springer-Verlag.
18. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of 18th Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, New York, 1991. ACM Press.
19. A. Villanueva. *Model Checking for the Concurrent Constraint Paradigm*. PhD thesis, University of Udine and Technical University of Valencia, May 2003.