

# Symbolic Model Checking for Timed Concurrent Constraint Programs<sup>\*</sup>

M. Alpuente<sup>1</sup>, M. Falaschi<sup>2</sup>, and A. Villanueva<sup>1</sup>

<sup>1</sup> Departamento de Sistemas Informáticos y Computación, UPV, Spain  
{alpuente,villanue}@dsic.upv.es

<sup>2</sup> Dipartimento di Matematica e Informatica, U. di Udine, Italy  
falaschi@dimi.uniud.it

**Abstract** 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 formal verification technique which allows one to automatically check whether a specific property is satisfied by a model of the system; otherwise it provides a counterexample which helps the programmer to debug the wrong code. In this paper we develop a symbolic model checking technique for the *Timed Concurrent Constraint Language* (tccp), a declarative language within the concurrent constraint paradigm which allows one to program reactive systems in a very natural way. Two of the most important features of the language are the use of constraints and a notion of time which is within the operational model. By taking advantage of these two features and using an extension of *Difference Decision Diagrams* (DDD<sub>s</sub>), we formulate a symbolic model checking algorithm for tccp which mitigates the state explosion problem that is common to model checking approaches. The symbolic approach to model checking tccp leads to an important improvement w.r.t. previous approaches based on the classical Linear Time Logic (LTL) model checking algorithm.

**Keywords:** Timed Concurrent Constraint, Model Checking, DDD<sub>s</sub>

## 1 Introduction

In the last decades, formal verification of industrial applications has become a hot topic of research. Verification is usually carried out by using model checking algorithms which are able to demonstrate the satisfiability of certain properties formalized as logical formulas which are automatically checked on a model of the system.

Recent advances in model checking deal with huge state-spaces by using symbolic manipulation algorithms inside model checkers [6,11]. Other

---

<sup>\*</sup> This work has been partially supported by the MCYT under grant TIC2001-2705-C03-01

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 [5].

The *concurrent constraint* paradigm (cc) was first introduced in [15] to model concurrent systems. A global store consisting of constraints contains the information accumulated during the computation. Constraints are dynamically added to the store, and it is also possible to consult the store. 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 programming model was extended in [2] over a discrete notion of time. Concurrent systems are more difficult to be manually debugged, simulated or verified than sequential systems. This difficulty makes yet more interesting the availability of automatic formal verification tools and techniques for these systems.

In this paper, we formulate a symbolic model checking algorithm to verify reactive systems which are specified in a timed concurrent constraint language. This is a significant improvement w.r.t. previous works [8,9,18], where the classical LTL model checking algorithm was adapted to the concurrent constraint paradigm. The symbolic approach adopted in this paper allows us to verify more complex reactive systems.

In Section 2, we introduce the specification language for reactive systems which we consider in this work. In Section 3, we recall and extend the notion of DDD in order to be able to deal with lists which we use to model the value of system variables along the time. Section 4 introduces the symbolic model checking technique for DDDs with lists, together with an illustrative example. Finally, in Section 5 we show our conclusions and future work.

## 2 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 [15,17] with a notion of discrete time.

Basically, a cc program describes a system of agents that can add information into a store as well as check 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. It is assumed that *ask* and *tell* actions take one time unit, and the parallel operator is interpreted in terms of

maximal parallelism. Computation evolves in steps of one time unit by adding or asking (entailment test) some information to the store. Moreover, it is assumed that constraint entailment tests take a constant time independently of the size of the store<sup>3</sup>.

Let us first informally 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 satisfying some standard set-based properties. Examples of such constraint systems are the Herbrand constraint system, the FD constraint system [10] and the **Gentzen** constraint system [16].

A *cylindric* constraint system consists of a simple constraint system plus an existential quantification operator which is monotone, conservative under information loss and gives support to 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].

We define the set  $AP$  of atomic propositions as the set of atomic propositions of the cylindric constraint system underlying the tccp language. In the rest of the paper, we identify the notion of (finite) constraint with an atomic proposition.

Let us now recall the syntax of tccp, defined in [2] as follows:<sup>4</sup>

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

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

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

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

<sup>3</sup> In practice some syntactic restrictions are imposed in order to ensure that these hypotheses are reasonable (see [2] for details).

<sup>4</sup> The operational and denotational semantics of the language can be found in [2].

The agent for the procedure call is  $p(x)$ . It is important to remark that only the tell, the ask and the procedure call agents imply an increment of one time-unit, whereas other primitives are interpreted as instantaneous.

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*. The notion of timeout is defined as the ability to wait for a specific signal and, if a limit of time is reached and such signal is not present, then an exception program is executed. The notion of preemption is defined as the ability to abort a process when a specific signal is detected. The conditional agent executes  $A$  if the store entails  $c$ , otherwise it executes  $B$ .

Now we consider a specific constraint system which allows us to define an interesting class of software systems. In particular, we need the traditional arithmetic for reals including addition, equality and order comparison. This part of the constraint system handles the information coming from the constrained nature of the system. On the other hand, we are also interested to handle lists since streams are modeled in *tccp* as lists of terms as in many other logic languages. In *tccp*, lists are used for modeling streams which 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, the tail of the list models the future, and it is instantiated to a list variable. The entailment relation for lists is specified by Clark's Equality Theory. For example,  $[X|Z] = [a|Y]$  entails  $X = a$  and  $Z = Y$ .

Roughly speaking, we define the set of tokens of our constraint system as the set of terms of the form  $X - Y \leq c$ ,  $X - Y < c$ ,  $V = []$ ,  $V = [X|W]$  or  $V = [c|W]$  where  $X$  and  $Y$  are real or integer variables,  $V$  and  $W$  are list variables, and  $c$  is a real or integer value constant.

### 3 Extended Difference Decision Diagrams

Difference Decision Diagrams (DDD) are an extension of the Binary Decision Diagrams (BDD) defined in [3] to represent difference constraint expressions symbolically. While BDDs can be seen as an efficient representation of boolean formulas, DDDs data structures efficiently represent difference constraint expressions, which 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. These constraints naturally appear when we model timing systems where different clocks need to be synchronized.

DDD 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. Nevertheless, a semi-canonical structure which can be handle more efficiently, suffices in our setting for checking satisfiability and tautology.

We note that, to correctly represent constraints, we cannot simply use a boolean structure as OBDDs. Constraints are able to encode some implicit information which cannot be directly represented by boolean functions. This 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 OBDDs, then we do not obtain a canonical representation for the considered difference constraint expression, as opposed to the case of OBDDs. However, it is still possible to obtain a semi-canonical<sup>5</sup> structure which can be used to decide satisfiability, validity, falsifiability or unsatisfiability of expressions. There is also an algorithm to obtain a canonical representation of DDDs which is quite expensive. We do not consider it in order not to overly complicate the development.

### 3.1 Difference Decision Diagrams + Logical Streams

We intend to represent `tccp` programs (as defined in Section 2) by using some symbolic data structure. Since the constraint system underlying the language contains difference constraint expressions *and* list (or stream) expressions, we can extend the DDD data structure defined in [12,13] to handle logical lists or streams. Therefore, in this section we define *Difference Decision Diagram + Logical Streams* (DDD+LS) structure as an extension of DDDs.

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  $\mathbf{0}$  and  $\mathbf{1}$ ). In addition,  $V$  contains a set of non-terminal vertices with out-degree two. Each non-terminal vertex  $v$  has nine attributes. Intuitively, the four first attributes (called  $pos(v)$ ,  $neg(v)$ ,  $op(v)$  and  $const(v)$ ) represent a difference constraint of the form  $pos(v) - neg(v) op(v) const(v)$  where  $op(v) \in \{LE, LEQ, LIST\}$  indicate whether the node represents a strict disequality, a disequality or a stream constraint, respectively. Then, the  $left(v)$ ,  $head(v)$  and  $tail(v)$  attributes represent the stream constraint

---

<sup>5</sup> A DDD is semi-canonical if (i) an expression  $\phi$  is represented by  $\mathbf{1}$  iff  $\phi$  is valid, and (ii) an expression  $\phi$  is represented by  $\mathbf{0}$  iff  $\phi$  is unsatisfiable.

$left(v) = [head(v)|tail(v)]$ . We note that there are two kinds of variables:  $\mathbb{D}$ -variables and list variables, where  $\mathbb{D}$  ranges on  $\mathbb{R}$  or  $\mathbb{Z}$ . The remaining two attributes ( $high(v)$  and  $low(v)$ ) represent the two branches that can be followed in the graph from the vertex  $v$ .

Some shorthands are defined to reference combinations of attributes. For instance, the operator  $var(v)$  represents the pair  $(pos(v), neg(v))$  whereas the operator  $bnd(v)$  corresponds to the pair  $(op(v), const(v))$ . We let  $list(v)$  denote the pair  $(head(v), tail(v))$  and  $listExp(v)$  is the pair  $(left(v), list(v))$ ; by  $varl(v)$  we represent the pair  $(head(v), tail(v))$ . 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.

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 2. **Exp** stands for difference constraint expressions and stream expressions.

**Definition 2.** *Let  $v$  be a vertex of a DDD+LS structure. We define the function  $\mathcal{S} : V \rightarrow \mathbf{Exp}$ :*

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

The semantics of both, terminal vertices and vertices with  $op(v) = \text{LE}$  or  $op(v) = \text{LEQ}$ , coincide with the semantics of nodes of DDDs ([12,13]). Definition 2 just adds the semantics derived from the new attributes introduced in DDD+LS.

**Ordered DDD+LS.** To formalize the notion of Ordered DDD+LS, we need to define a total order on the vertices of the graph. First of all, we assume to have an order between variables. We require that given a node  $n$ , both  $var(v)$  and  $varl(v)$  are *normalized*. This means that  $pos(v) > neg(v)$  and  $left(v) < tail(v)$ . Then we assume that  $\text{LIST} < \text{LE} < \text{LEQ}$ . 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.

**Definition 3 (ODDD+LS).** An Ordered DDD+LS (*ODDD+LS*) is a DDD+LS where each non-terminal vertex  $v$  satisfies:

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

**Semi-canonical structure.** In order to verify properties, we need a semi-canonical structure. We propose some local and path reductions for ODDD+LSs, which are inspired by the reductions defined for ODDDs.

**Definition 4 ( $R_L$ Reduced DDD).** Let  $D$  be an ODDD+LS, and let  $u$  and  $v$  be non-terminal vertices of  $D$ . Then  $D$  is Locally Reduced DDD+LS ( $R_L$ DDD+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  $u$  and  $v$ , if  $attr(u) = attr(v)$ , 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  $high(v) \neq high(low(v))$

Roughly speaking, the first point states that, if we are working with integers, then we can represent the  $<$  operator in terms of  $\leq$  as occurs in DDDs. The second and third points coincide with the requirements for reduced BDDs. The fourth point states that, if the variables in the low branch of node  $v$  coincide with  $var(v)$ , then the high branches must differ. Finally, we add a new restriction, similar to the fourth point, concerning the reduction of nodes containing stream expressions.

The next step towards the semi-canonical representation of DDD+LSs is the definition of the path reduction. We first need to define the semantics of edges and paths. Note that 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 5.** Let  $u, v$  be vertices of a DDD+LS structure. Let  $u$  and  $v$  be adjacent vertices. The function  $\mathcal{E} : E \rightarrow \mathbf{Exp}$  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 is defined as a finite sequence of edges of the form  $\langle (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k) \rangle$ . 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, 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 with  $R_P\text{DDD}+\text{LS}$  the structure resulting from applying this reduction step to  $R_L\text{DDD}+\text{LS}$ s. A semi-canonical representation has exactly one DDD+LS for 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 [12]. 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  $R_P\text{DDD}+\text{LS}$  in a safe way. We know that the expression  $\phi_u$  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 ( $R_P\text{DDD}+\text{LS}$ s are semi-canonical).** *In a  $R_P\text{DDD}+\text{LS}$ , the terminal vertex  $\mathbf{1}$  is the only representation of valid expressions, and the terminal vertex  $\mathbf{0}$  is the only representation of unsatisfiable expressions.*

### 3.2 Construction of DDD+LSs

In this section, we show how a  $R_P\text{DDD}+\text{LS}$  structure can be built from a stream expression. Vertices and edges of the DDD+LS are stored in a graph data structure called *Graph*. Let  $G$  be a *Graph*. Initially,  $G$  contains only the two terminal vertices  $\mathbf{0}$  and  $\mathbf{1}$ . 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,  $\text{insert}(G, a)$  creates a new vertex  $v$  in  $G$  with attribute  $a$ , and returns  $v$ . The function  $\text{member}(G, a)$  returns true if there exists a vertex in  $G$  with attribute  $a$ . Finally,  $\text{lookup}(G, a)$  returns the vertex in  $G$  with attribute  $a$ .

We also need some operations which obtain information about the attributes of a graph. Since names and behavior of these operations are very intuitive, we will use them directly in the algorithms. For a formal definition, the reader can consult [1].

In the following, we extend the algorithms in [12] (defined to construct DDDs), to construct the DDD+LS structure. We also develop the new algorithms to deal with stream expressions.

```

vertex MKD(x: DVar, y: Dvar, o: op, c: Const, h: vertex, l: vertex)
if  $\mathbb{D} = \mathbb{Z} \wedge o = \text{LE}$  then  $c = c - 1$ 
            $o = \text{LEQ}$ 
if member( $G, (x, y, o, c, \aleph, \aleph, \aleph, h, l)$ ) then
  return lookup( $G, (x, y, o, c, \aleph, \aleph, \aleph, h, l)$ )
else if  $l = h$  then return  $l$ 
  else if  $(x, y) = \text{var}(l) \wedge h = \text{high}(l)$  then return  $l$ 
  else return insert( $G, (x, y, o, c, \aleph, \aleph, \aleph, h, l)$ )

```

**Figure 1.** MKD Algorithm that creates a vertex for a difference constraint expression

In Figure 1, the algorithm MKD for difference constraints is given as an extension of the algorithm presented in [12]. We have modified the attributes of nodes to take into account that nodes in DDD+LSs have three additional attributes. The preconditions for the algorithm are those of DDDs plus an extra condition which applies when we are dealing with difference constraints (namely,  $op(v) \neq \text{LIST}$ ).

In Figure 2 the algorithm MKL is given which builds the vertex representing the stream expression. The set of preconditions for the MKL algorithm is similar to the preconditions for the MKD algorithm except that we require that  $op(v) = \text{LIST}$

```

vertex MKL(x: LVar, y: Var, z: LVar, h: vertex, l: vertex)
if member( $G, (\aleph, \aleph, \text{LIST}, \aleph, x, y, z, h, l)$ ) then
  return lookup( $G, (\aleph, \aleph, \text{LIST}, \aleph, x, y, z, h, l)$ )
else if  $l = h$  then
  return  $l$ 
  else if  $(x, z) = \text{plist}(l) \wedge h = \text{high}(l)$  then
  return  $l$ 
  else
  return insert( $G, (\aleph, \aleph, \text{LIST}, \aleph, x, y, z, h, l)$ )

```

**Figure 2.** Algorithm MKL that creates a vertex for a list expression

In [12], some functions are given which normalize pairs of variables before the construction of vertices. We can use similar procedures for the difference expressions contained in our DDD+LS structures.

The last step for the construction of the DDD+LS structure, is to design 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 [3], 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 algorithm for DDD+LSs. We do not include this algorithm in the paper since it follows closely the design of APPLY with some little adjustment to include the handling of the stream expressions. The reader can find the algorithm in [1].

## 4 Symbolic Model Checking

In [9,18], a model checking algorithm was proposed which allows one to verify tcp programs. The idea was to *automatically* construct a compact model which represents the tcp program; then, a logic dealing with constraints was proposed as the basis to develop a classical LTL model checking 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 directly check properties over the built model. The problem with the exhaustive approach, which is based on a tableau algorithm, is in the state-explosion problem (as occurs in the classical approaches) which shows up when we try to combine the model with the property that we want to verify.

By considering the constraint system defined in Section 2 for the tcp language, the model which can be automatically obtained by following [9,18] only contains difference and stream constraints. Thus, if we represent the tcp Structure by means of DDD+LSs, then we can use the efficient algorithms for checking DDD+LSs in order to verify tcp programs. In the following we formalize our verification strategy and illustrate it by means of a detailed example.

### 4.1 Model construction

Following [8,9,18], given a tcp program, we can build a tcp Structure which represents a model of the system.

A tccp Structure is a graph structure analogous to a Kripke Structure, which is widely used in many different model checking approaches [5,4]. Actually, the main different point is the definition of state. A state in a tccp Structure is defined as a (satisfiable) set of constraints which represent a set of possible values of variables. A state in a Kripke Structure is defined as an assignment of values to variables. In other words, a state in a tccp Structure is a set of states in a Kripke Structure. We refer to [8,18] for the formal definition of tccp Structure.

**Modelling Example: The Soldiers' problem.** The soldiers' problem is formulated as follows ([14]).

Four soldiers who are heavily injured, try to flee to their home land. The enemy is chasing them and in the middle of the night they arrive at a bridge that spans a river which is the border between the two countries at war. The bridge has been damaged and can only carry two soldiers at a time. Furthermore, several land-mines have been placed on the bridge and a torch is needed to sidestep all the mines. The enemy is on their tails, so the soldiers know that they have only 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured. The crossing time (one-way) for soldier  $S_0$  is 5 minutes, for soldier  $S_1$  is 10 minutes, for soldier  $S_2$  is 20 minutes, and finally the crossing time for soldier  $S_3$  is 25 minutes.

Does a schedule exists getting all four soldiers to the safe side within 60 min.?

We model the problem as a tccp program. Then, from the tccp program we construct the corresponding tccp Structure. Note that the arithmetic underlying the program can be expressed as difference expressions.

The tccp program corresponding to the main predicate is shown in Figure 3. The program consists of some initial statements, a procedure call to the RIVERCROSS predicate and another procedure call to the CHECK predicate. All these three calls are run in parallel. We omit the code for the CHECK predicates, which can be found in [1].

```

SOLDIERS() ::=
tell( $C = 0$ ) || tell( $S_0 = 0$ ) || tell( $S_1 = 0$ ) || tell( $S_2 = 0$ ) || tell( $S_3 = 0$ ) ||
RIVERCROSS( $[C|C']$ ,  $[S_0|S'_0]$ ,  $[S_1|S'_1]$ ,  $[S_2|S'_2]$ ,  $[S_3|S'_3]$ ,  $[T|T']$ ) ||
ask(true) → CHECK( $[C|C']$ ,  $[S_0|S'_0]$ ,  $[S_1|S'_1]$ ,  $[S_2|S'_2]$ ,  $[S_3|S'_3]$ ,  $[T|T']$ )

```

**Figure 3.** Example: tccp program for the Soldiers' problem

The tccp predicate modelling the most important aspect of the program is the RIVERCROSS predicate. It states that two soldiers can cross the bridge to the safe land only if they can take the torch. Moreover, only one soldier is allowed to come back to the unsafe zone to bring the torch to the remaining soldiers. The code is (partially) shown in Figure 4.

```

RIVERCROSS( $C, S_0, S_1, S_2, S_3, T$ ) ::=
  ask( $C = [C_1|C'] \wedge C_1 \leq 60 \wedge S_0 = [0|S'_0] \wedge S_1 = [0|S'_1] \wedge T = [0|T'] \wedge$ 
     $S_2 = [X_2|S'_2] \wedge S_3 = [X_3|S'_3]$ )  $\rightarrow$  tell( $S'_0 = [1|S''_0]$ ) ||
    tell( $S'_1 = [1|S''_1]$ ) || tell( $T' = [1|T'']$ ) ||
    tell( $C_a = C_1 + 10$ ) || tell( $C' = [C_a|C'']$ ) ||
    tell( $S'_2 = [X_2|S''_2]$ ) || tell( $S'_3 = [X_3|S''_3]$ ) ||
    RIVERCROSS( $C', S'_0, S'_1, S'_2, S'_3, T'$ ) +
    ...
  ask( $C = [C_1|C'] \wedge C_1 \leq 60 \wedge S_2 = [0|S'_2] \wedge S_3 = [0|S'_3] \wedge T = [0|T'] \wedge$ 
     $S_0 = [X_0|S'_0] \wedge S_1 = [X_1|S'_1]$ )  $\rightarrow$  tell( $S'_2 = [1|S''_2]$ ) ||
    tell( $S'_3 = [1|S''_3]$ ) || tell( $T' = [1|T'']$ ) ||
    tell( $C_a = C_1 + 25$ ) || tell( $C' = [C_a|C'']$ ) ||
    tell( $S'_0 = [X_0|S''_0]$ ) || tell( $S'_1 = [X_1|S''_1]$ ) ||
    RIVERCROSS( $C', S'_0, S'_1, S'_2, S'_3, T'$ ) +
    ...
  ask( $C = [C_1|C'] \wedge C_1 \leq 60 \wedge S_0 = [1|S'_0] \wedge T = [1|T'] \wedge S_1 = [X_1|S'_1] \wedge$ 
     $S_2 = [X_2|S'_2] \wedge S_3 = [X_3|S'_3]$ )  $\rightarrow$  tell( $S'_0 = [0|S''_0]$ ) || tell( $T' = [0|T'']$ ) ||
    tell( $C_a = C_1 + 5$ ) || tell( $C' = [C_a|C'']$ ) ||
    tell( $S'_1 = [X_1|S''_1]$ ) || tell( $S'_2 = [X_2|S''_2]$ ) ||
    tell( $S'_3 = [X_3|S''_3]$ ) ||
    RIVERCROSS( $C', S'_0, S'_1, S'_2, S'_3, T'$ ) +
    ...
  ask( $C = [C_1|C'] \wedge C_1 \leq 60 \wedge S_3 = [1|S'_3] \wedge T = [1|T'] \wedge S_0 = [X_0|S'_0] \wedge$ 
     $S_1 = [X_1|S'_1] \wedge S_2 = [X_2|S'_2]$ )  $\rightarrow$  tell( $S'_3 = [0|S''_3]$ ) || tell( $T' = [0|T'']$ ) ||
    tell( $C_a = C_1 + 25$ ) || tell( $C' = [C_a|C'']$ ) ||
    tell( $S'_0 = [X_0|S''_0]$ ) || tell( $S'_1 = [X_1|S''_1]$ ) ||
    tell( $S'_2 = [X_2|S''_2]$ ) ||
    RIVERCROSS( $C', S'_0, S'_1, S'_2, S'_3, T'$ )

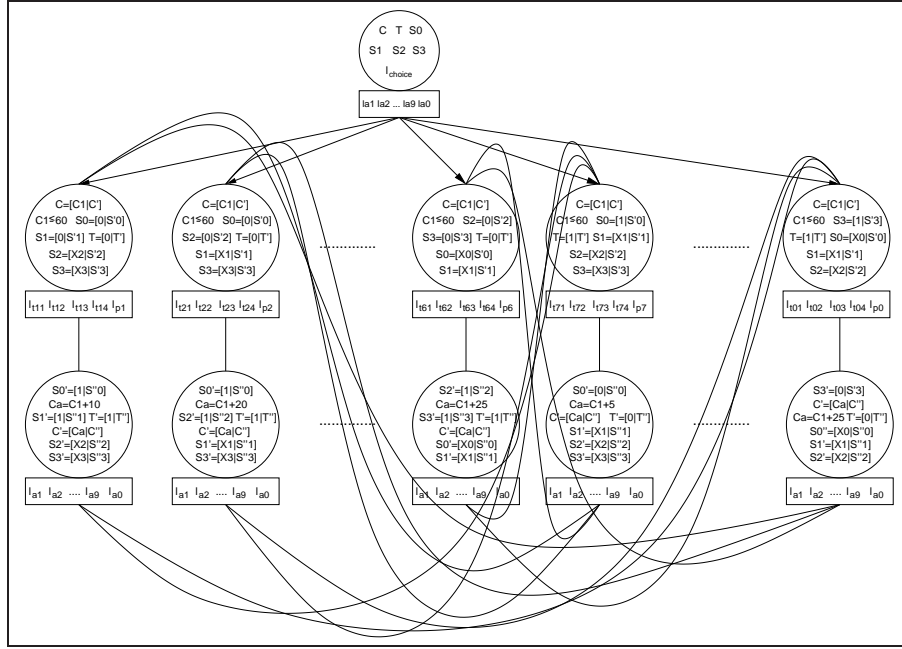
```

**Figure 4.** Example: tcsp program for the allowed soldiers movements

RIVERCROSS specifies the movements that can do the soldiers. Each ask branch of the agent RiverCross describes a possible move. The idea is that either one single soldier or two soldiers can cross the bridge from one side to the other provided they can take the Torch ( $T$ ). The time which is necessary to cross corresponds to that required by the most injured of the two soldiers. The total time ( $C$ ) which has passed is increased accordingly.

In Figure 5, we (partially) show the tcsp Structure we construct for the predicate CROSSRIVER. There is an initial node at the top which has ten children corresponding to the ten possible behaviors defined by the procedure. For the sake of simplicity, in the figure we only show the first two sons, the sixth, the seventh and the tenth ones. Since no conditional nor choice agent is executed, the nodes in the second level evolve deterministically.

The subsequent evolution of nodes at the second level depend on the position of the torch. Consequently, the six nodes in the left side of the figure are connected to the four nodes at the right, and vice-versa. This



**Figure 5.** tccp Structure for the soldiers' program

fact is automatically detected in the construction of the model since the accumulated store determines the possible behaviours.<sup>6</sup>

In order to improve clarity, we draw in a separate box the labels of each node of the tccp Structure. This differentiates the store in a specific node from the labels corresponding to the agents that must be executed at that execution point.

## 4.2 Representation using DDD+LSs

As we have said before, the main idea of this work is to define a method to represent a tccp Structure by using DDD+LSs. This allows us to mitigate the state explosion problem by following a symbolic approach which significantly reduce the search space. In [12,13], efficient algorithms which check validity or tautology of such structures w.r.t. some property were defined. The idea here is to take advantage of such algorithms in order to improve the verification of tccp code.

A tccp Structure can be translated to a formula of the logic underlying our constraint system. In particular, we can represent the relation  $R$  of the structure as a disjunction of conjunctions in a way similar to the method used for BDDs.

<sup>6</sup> We note that two nodes are considered equivalent if the two corresponding stores of a node are equal modulo variable renaming.

If we are able to represent the system in the logic of DDD+LSs, then we can construct the symbolic structure corresponding to the formula, which represents an encoding of the system.

Let us explain the procedure by using the example in Figure 5. Each arc in this figure corresponds to an element in the relation  $R$ . Thus, we can code each transition in a similar way as in the classical approach. For example, the following formula models the leftmost arc of the figure:

$$\begin{aligned} S0 &= [0|S0'] \wedge S1 = [0|S1'] \wedge S2 = [X2|S2'] \wedge S3 = [X3|S3'] \\ \wedge T &= [0|T'] \wedge C = [C1|C'] \wedge C1 \leq 60 \\ \wedge S0' &= [1|S0''] \wedge S1' = [1|S1''] \wedge S2' = [X2|S2''] \wedge S3' = [X3|S3''] \\ \wedge T' &= [1|T''] \wedge Ca = C1 + 10 \wedge C' = [Ca|C''] \end{aligned}$$

Following this idea, we can obtain the formula which represents the whole tccp Structure. Such formula corresponds to the kind of expressions that we are able to handle by using DDD+LSs. Thus, we can construct the DDD+LS structure corresponding to the system by applying the algorithms introduced in this paper.

Once we have obtained the DDD+LS structure, we can check whether a given property is satisfied by the system in the way explained in Section 3.2.

## 5 Conclusions

We have generalized DDDs to a new structure which allows us to represent tccp programs symbolically. We have formalized the corresponding notions and algorithms which allow us to check properties on these models. This novel symbolic methodology improves the automatic verification of reactive systems.

In [7], a different data structure called CST is presented which allows one to represent integer linear constraints symbolically and is used to define a parameterized verification method for (infinite state) Petri nets.

As future work, we plan to extend the language to consider more general constraint expressions. We are also implementing the method in order to evaluate how our approach works in practice.

## References

1. Alpuente, M., Falaschi, M., Villanueva, A.: Symbolic Model Checking for Timed Concurrent Constraint Programs. Technical Report DSIC-II/19/03, DSIC, Technical University of Valencia (2003) Available at <http://www.dsic.upv.es/users/elp/villanue/papers/techrep03.ps>

2. de Boer, F.S., Gabbrielli, M., Meo, M.C.: A Timed Concurrent Constraint Language. *Information and Computation* **161** (2000) 45–83
3. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* **C-35** (1986) 677–691
4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: *Proceedings of Workshop on Logic of programs*. Volume 131 of LNCS, Berlin, Springer-Verlag (1981) 52–71
5. Clarke, E.M., Grumberg, O., Peled, D.: *Model Checking*. The MIT Press, Cambridge, MA (1999)
6. Clarke, E.M., McMillan, K.M., Campos, S., Hartonas-Garmhausen, V.: Symbolic Model Checking. In: *Proceedings of the 8th International Conference on Computer Aided Verification*. Volume 1102 of LNCS, Springer-Verlag (1996) 419–422
7. Delzanno, G., Raskin, J.F., Begin, L.V.: CSTs (Covering Sharing Trees): compact Data Structures for Parametrized Verification. *Software Tools for Technology Transfer* (2003) To appear
8. Falaschi, M., Policriti, A., Villanueva, A.: Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language - I. In: *Selected papers from 2000 Joint Conference on Declarative Programming*. Volume 48 of ENTCS, Elsevier Science Publishers (2000)
9. Falaschi, M., Villanueva, A.: Automatic verification of timed concurrent constraint programs (2003) Submitted for publication.
10. Hentenryck, P.V., Saraswat, V.A., Deville, Y.: Constraint processing in cc(fd). Technical report, Computer Science Department, Brown University (1992)
11. McMillan, K.L.: *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic (1993)
12. Møller, J., Lichtenberg, J.: Difference Decision Diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Tech. University of Denmark (1999)
13. Møller, J., Lichtenberg, J., Andersen, H., Hulgaard, H.: Difference Decision Diagrams. In: *Proceedings of the 13th International Workshop on Computer Logic Science*. Volume 1683 of LNCS (1999) 111–125
14. Ruys, T., Brinksma, E.: Experience with literate programming in the modelling and validation of systems. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Volume 1384 of LNCS, Springer-Verlag (1998) 393–409
15. Saraswat, V.A.: *Concurrent Constraint Programming Languages*. In: PhD Thesis, Carnegie-Mellon University (1989)
16. Saraswat, V.A., Jagadeesan, R., Gupta, V.: Programming in Timed Concurrent Constraint Languages. In: *Constraint Programming Proceedings 1993 NATO ASI*, Berlin, Springer-Verlag (1994) 361–410
17. Saraswat, V.A., Rinard, M., Panangaden, P.: Semantic Foundations of Concurrent Constraint Programming. In: *Proceedings of 18th Annual ACM Symposium on Principles of Programming Languages*, New York, ACM Press (1991) 333–352
18. Villanueva, A.: *Model Checking for the Concurrent Constraint Paradigm*. PhD thesis, University of Udine in cotutela with Technical University of Valencia (2003)