Clock Difference Diagrams

Kim G. Larsen Carsten Weise
BRICS*, Aalborg University, Denmark
Wang Yi Justin Pearson
Department of Computer Systems, Uppsala University, Sweden

Abstract

We sketch a BDD-like structure for representing unions of simple convex polyhedra, describing the legal values of a set of clocks given bounds on the values of clocks and clock differences.

1 Introduction

The basic problem we are trying to tackle is the combination BDD's and DBM's (difference bound matrices) in order to allow a completely BDD-based approach to the verification of continuous real-time systems. Early approaches in this direction include [WTD95] and [Bal96]. Another inspiration for this work comes from [ST98]. Some of the ideas come from the implementation of a decision algorithm for timed bisimulation ([WL97]).

2 Definition of CDD's

We assume a finite set of real-valued clocks $\mathcal{C} = \{X_1, \ldots, X_k\}$. We are interested in a data structure to represent and manipulate sets of possible values of these clocks. In particular, we shall confine ourselves to sets being the finite unions of simple convex polyhedra. The simple convex polyhedra are described by bounds on the individual clocks and clock differences of the form $X_i - X_j$. These kind of sets occur typically in the analysis of real-time systems when modelled as timed automata.

For a uniform treatment, we assume an additional clock X_0 which always has value zero. Then the absolute value of clock X_i is referred to as $X_i - X_0$.

*BRICS: Basic Research in Computer Science, Centre of the Danish National Research Foundation

Let $\mathcal{V} := \{v : \mathcal{C} \to \mathbb{R}\}$ be the set of *clock valuations*. Then any polyhedra described by clock differences is a subset of $\mathcal{V}$.

A *clock constraint* has the form $m \leq X_i - X_j \leq n$, where $i, j \in \{0, \ldots, k\}$ and m, n are integers. Instead of $\leq$ also $<$ can be used in the lower and the upper bound. We will always require $i > j$, as $m \leq X_i - X_j \leq n$ is the same as $-n \leq X_j - X_i \leq -m$. Note that for $i = j$, we always have $0 \leq X_i - X_j \leq 0$.

For any constraint, we call the pair (i, j) the *type* of the constraint. It is obvious that in any description of a convex set, at most one constraint per type is needed. As mentioned before, we always assume $i > j$. We define a linear ordering on the types, which we denote by $\sqsubseteq$. This ordering is the "reversed lexicographical" ordering, i.e. $(i, j) \sqsubseteq (i', j')$ iff either $j < j'$ or $j = j'$ and $i \leq i'$.

In the following, any set describable by a conjunction of clock constraints will be called a *zone*. A *federation* is any finite union of zones.

We will now define a data structure called *clock difference diagrams* (short: CDD) which can be used to represent federations. The basic idea is derived from BDD's, but adapted to the fact that our variables do not range over a simple two-valued alphabet, but over the real numbers instead.

While the idea of a BDD is that of a decision-diagram branching on the different single values of the variables of a term, a CDD node branches with respect to intervals of the reals for a given clock difference, i.e. a CDD node is associated with the type of the clock difference for which it is testing. The size and the number of the intervals is not fixed. However there can only be a finite number of intervals, and they must be compatible with the clock constraints (i.e. they are intervals with integer bounds, and any bound can either be included or not). Remember that a major idea in BDD's is that of sharing identical substructures, therefore a BDD is in fact an acyclic graph rather than a tree, and so will be the CDD's.

In the general case we will not require that the intervals belonging to one node are disjoint. We will however show that there is an easy way to obtain disjoint intervals if needed, as most of our operations will require disjoint intervals.

The finite number of intervals within a node is assured by not differentiating between points when a clock exceeds a given maximal value M. Thus the intervals $(+M, +\infty)$ and $(-\infty, -M)$ cannot be further partitioned. If an operation yields a subinterval intersecting one of these, then this subinterval must be extended with the whole interval.

There are two special nodes called TRUE and FALSE. They are used to indicate that along a certain path all valuations belong to the federation or do not belong to the federation. In general, our graphs will be complete, i.e. for any valuation there is at least one path leading to either a TRUE or a FALSE node.

Thus a CDD consists of the following kinds of nodes:

- end-nodes, which are either TRUE or FALSE, and which have no successor,

- inner nodes, which for a fixed type branch to nodes for different constraints (intervals) of this type

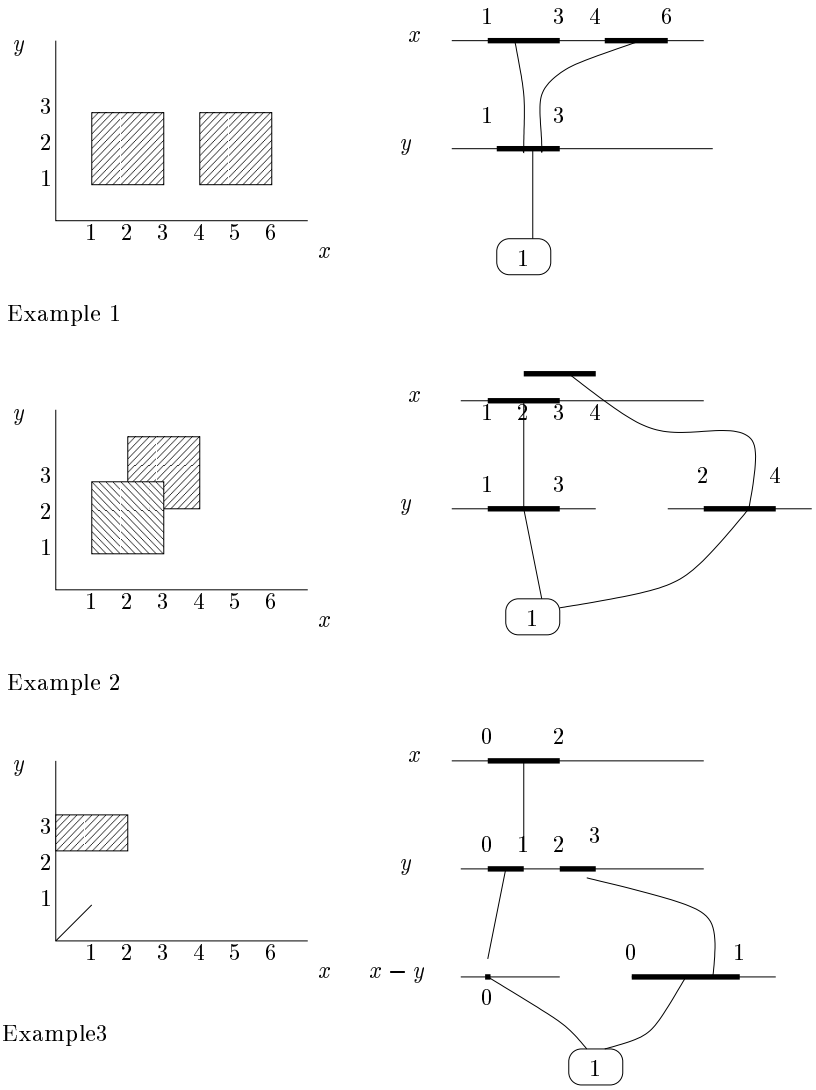

Figure 1: Three Simple Examples

Example 1, 2 and 3 as given in Figure 1 are simple examples of CDD's. Note that instead of TRUE we use boxes with a 1 inside (like in BDD's). We have also not drawn the FALSE end-nodes. Remember that there are many different ways to represent a federation as a CDD. Note also that Example 1 uses sharing of the nodes of type $(2, 0)$, i.e. those giving constraints on Y.

A *clock difference diagram* is defined as a directed, acyclic graph, which has

- a node called the start node from which all nodes of the graph are reachable,
- inner nodes written as $((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ where (i,j) is the type of a constraint, the I_n are intervals of the real numbers, and the T_n are CDD's again. We require *completeness*, i.e. $\bigcup_{n\in\{1,\ldots,q\}} I_n = \mathbb{R}$.
- end-nodes which are either TRUE or FALSE.

Note that a node in a CDD defines a subgraph of all nodes reachable from this node. This subgraph can be seen as a CDD with the current node as the start node. We will identify a node and the sub CDD defined by it.

The interpretation of such a CDD should now be obvious: If we take a path from the start node to an end-node, this path describes a zone given by the conjunction of the clock constraints on this path. A clock constraint on a path is the constraint representing the interval which we had to choose to follow the branches building the path. If the path ends in TRUE, then all valuations of this zone belong to the federation. The CDD represents the union of all zones described by paths leading to TRUE.

In order to formalize the semantics of CDD's, we use the following notation:

- given a type (i,j) and an interval I of the reals, then $I(i,j)$ denotes the clock constraint having type (i,j) which restricts the value of $X_i - X_j$ to the interval I.
- given a clock constraint ϕ and a valuation v, by $\phi(v)$ we denote the application of ϕ to v, i.e. the boolean value derived from replacing the clocks in ϕ by the values given in v.

Note that typically we will use the notation jointly, i.e. $I(i,j)(v)$ expresses the fact that v fulfills the constraint given by the interval I and the type (i,j).

As an example, if the type is $(2,1)$ and $I = [3,5)$, then $I(i,j)$ would be the constraint $3 \leq X_2 - X_1 < 5$. For v where $v(X_2) = 9$ and $v(X_1) = 5.2$ we would find that $I(i,j)(v)$ is true, while for v' with $v'(X_2) = 3$ and $v'(X_1) = 4$ we would have $I(i,j)(v')$ is false.

Using this, we can formally define the semantics of a CDD by

- $[\![True]\!] := \mathcal{V}$,
- $[\![False]\!] := \emptyset$,
- $[\![((i,j),(I_1,T_1),\ldots,(I_q,T_q))]\!] := \bigcup_{n\in\{1,\ldots,q\}} \{v \in [\![T_n]\!] | I_n(i,j)(v) = True\}$

Note that this semantics is defined so that any valuation v for which there is a path to TRUE in the CDD will be part of the set described by the CDD. This includes the case where there might be another path for v leading to a zero, which seems rather counter intuitive. In fact, the FALSE end-nodes could be clipped from the CDD's without doing harm. They are mainly used here to make some formal treatment more easy (e.g. definition of union and intersection). A good implementation would not need to store them.

2.1 Reducing Redundancy

A major point in the coding is to get rid of redundancies as much as possible. Therefore we require *orderedness* and *disjointness*.

Orderedness: Remember that we assume some ordering on the types. We further assume that TRUE and FALSE have a type. Their types are the maximal elements in the order, i.e. they are larger than any other type. Given a node N of a CDD, we can now speak of the node's type denoted by $\mathsf{type}(N)$. We say that a CDD is *ordered* if for any node $T = ((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ it is true that for all $n \in \{1,\ldots,q\}$ that $\mathsf{type}(T) \sqsubseteq \mathsf{type}(T_n)$ and $\mathsf{type}(T) \neq \mathsf{type}(T_n)$. This means that along any path, the types are increasing, and that no type occurs twice along the path.

In the following, we will always assume that our CDD's are ordered. All our operations will keep orderedness. Note that in fact Examples 1 to 3 were already ordered.

Disjointness: We say that a CDD $T = ((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ is *disjoint* if all intervals are disjoint, i.e. for all $n,m \in \{1,\ldots,q\}$, from $n \neq m$ it follows that $I_n \cap I_m = \emptyset$, and if all sub CDD's T_n are disjoint as well. The CDD's TRUE and FALSE are disjoint by definition.

Basically we will always require disjointness. However some of our operations will destroy disjointness. But there is a simple way to go from an ordered CDD to a disjoint ordered CDD which will be explained later.

Figure 2 shows how to represent Example 2 as a disjoint CDD. It also gives a different graphical representation of the set fitting more the CDD representation.

Sharing: In order to reduce memory for the storing of a CDD we want to have as much sharing of subgraphs as possible. So within a CDD we require that if there are two subgraphs which are isomorphic then they should be the same. It is of course always possible to construct a maximally shared CDD from any given CDD.

We can generalize this rule for sharing. If there are two neighboring or overlapping subintervals in a node which point to the same subgraph, then the two

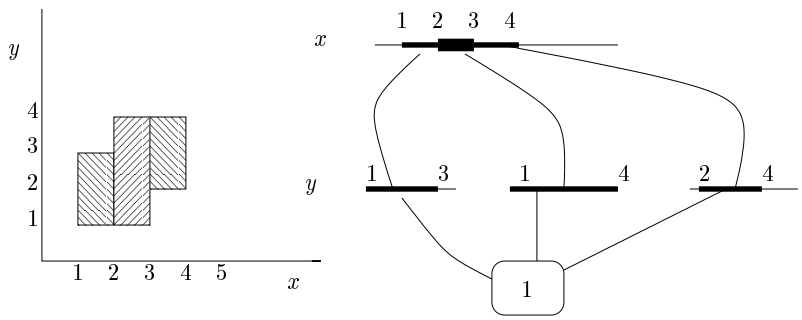

Figure 2: Enforcing Disjointness

intervals can be replaced by their union, pointing to the subgraph. Also, if a node contains $(-\infty, +\infty)$ as the only interval, this node may be removed by redirecting all pointers to the node to its subgraph.

2.2 Normal Form

Certain operations on CDD's require the CDD to be in a defined normal form, analogously to the case of DBM's for zones. A CDD in *normal form* is required to have maximal sharing, and be ordered and disjoint. Further we require that along any path leading to TRUE, the constraints must be the tightest possible, i.e. if we replace a constraint along a path by a tighter one, then the semantics of the CDD changes.

We will see further down how to obtain a CDD in normal form from an ordered and disjoint CDD.

3 Operations on CDD's

The most simple operations on CDD's are union and intersection. Assume two CDD's T_A, T_B given, which are ordered and disjoint. Then the CDD $T_C = T_A \cup T_B$ is constructed recursively in the following way:

- if $T_A = False$, then $T_C := T_B$,
- if $T_A = True$, then $T_C := True$,
- if $T_A \notin \{True, False\}$ and $\mathsf{type}(T_A) \neq \mathsf{type}(T_B)$, assume w.l.o.g. $\mathsf{type}(T_A) \sqsubseteq \mathsf{type}(T_B)$. Let $T_A = ((i,j),(I_1,T_1),\ldots,(I_q,T_q))$, then $T_C := ((i,j),(I_1,T_1 \cup T_B),\ldots,(T_q,T_q \cup T_B))$.

- if $T_A \notin \{True, False\}$ and $\mathsf{type}(T_A) = \mathsf{type}(T_B)$, then
  let $T_A = ((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ and $T_B = ((i,j),(J_1,S_1),\ldots,(J_r,S_r))$.
  Then

$$T_C := ((i,j), \quad \begin{array}{ll} (I_1 \cap J_1, T_1 \cup S_1), (I_1 \cap J_2, T_1 \cup S_2), \ldots, & (I_1 \cap J_r, T_1 \cup S_r), \\ (I_2 \cap J_1, T_2 \cup S_1), \ldots, & \\ \ldots & (I_q \cap J_r, T_q \cup S_r)) \end{array}$$

  where empty intervals can safely be discarded.

Note that this operation keeps orderedness and disjointness. Sharing can be maintained as well using the same dynamic methods as with BDD's.

The intersection is basically the same, where only the non-recursive step has to be adjusted. The CDD $T_C = T_A \cap T_B$ is constructed by

- if $T_A = False$, then $T_C := False$,
- if $T_A = True$, then $T_C := T_B$,
- else replace $T_n \cup S_m$ everywhere in the above by $T_n \cap S_m$.

3.1 Enforcing Disjointness

Given these basic operations, it is now easy to see how to enforce disjointness. Assume a CDD $T_A = ((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ where $I_{q-1} \cap I_q \neq \emptyset$. Then the CDD $T = ((i,j),(I_1,T_1),\ldots,(I_{q-1},T_{q-2}),(I_{q-1} \setminus I_q, T_{q-1}),(I_q \setminus I_{q-1}, T_q),(I_{q-1} \cap I_q, T_{q-1} \cup T_q))$ will have the same semantics, but the overlap between the last two intervals is gone. Applying this iteratively to all pairs of overlapping intervals and then recursively to all sub-CDD's will yield a disjoint CDD. Note that the union operation on CDD's required here does not destroy disjointness, guaranteeing termination.

Figure 2 shows how Example 2 looks like after enforcing disjoint intervals.

3.2 Setting clocks and letting time pass

Two very important operations in the analysis of timed automata, which we are aiming at, is the setting of a clock and letting time pass. Note that for these operations we need to assume normal form of the CDD. Note that these operations are mainly generalizations of their simpler DBM counterparts (where they require canonical form).

Setting a clock X_i ($i \neq 0$) to a constant c is done by replacing intervals in nodes. All intervals describing values of the clock X_i itself now become just $[c,c]$, so a subgraph

$$((i,0),(I_1,T_1),\ldots,(I_q,T_q))$$

is replaced by

$$((i,0),([c,c],T_1),\ldots,([c,c],T_q))$$

The difference between X_i and another clock X_j is then the difference between 0 and X_j minus the new value c of X_i. Note that subgraphs $((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ where $j \neq 0$ resp. $((j,i),(I_1,T_1),\ldots,(I_q,T_q))$ are reached by going through an interval I for type $(j,0)$, due to orderedness. This interval gives the difference between X_j and zero. Replacing is done by changing the subgraphs to

$$((i,j),(c-I,T_1),\ldots,(c-I,T_q))$$

resp.

$$((j,i),(I-c,T_1),\ldots,(I-c,T_q))$$

where $c-I$ and $I-c$ are defined as extensions of normal subtraction, i.e. $c-I := \{c-x \mid x \in I\}$ and $I-c := \{x-c \mid x \in I\}$.

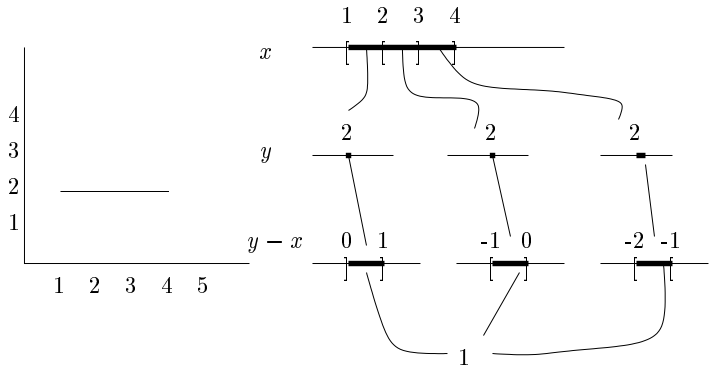

Figure 3: Setting Y to two

Figure 3 shows how Example 2 looks like after setting Y to two. Note that to do this correctly, it is necessary to make Example 2 disjoint and put it into normal form (as seen in Figure 2 and 6). However, the result is not the most compact form for this federation. Additional rules could be defined to gain a more compact form.

The future of a CDD is also computed straight forward by removing the upper bounds on all clocks, so $((i,0),(I_1,T_1),\ldots,(I_q,T_q))$ becomes $((i,0),(I'_1,T_1),\ldots,(I'_q,T_q))$ where $I'_n := (k,\infty)$ if k was the strict lower bound of I_n, and $I'_n := [k,\infty)$ if k was the non-strict lower bound of I_n. Figure 4 and 5 gives an example.

In both cases disjointness is destroyed, but can be regained using the method explained in the previous section.

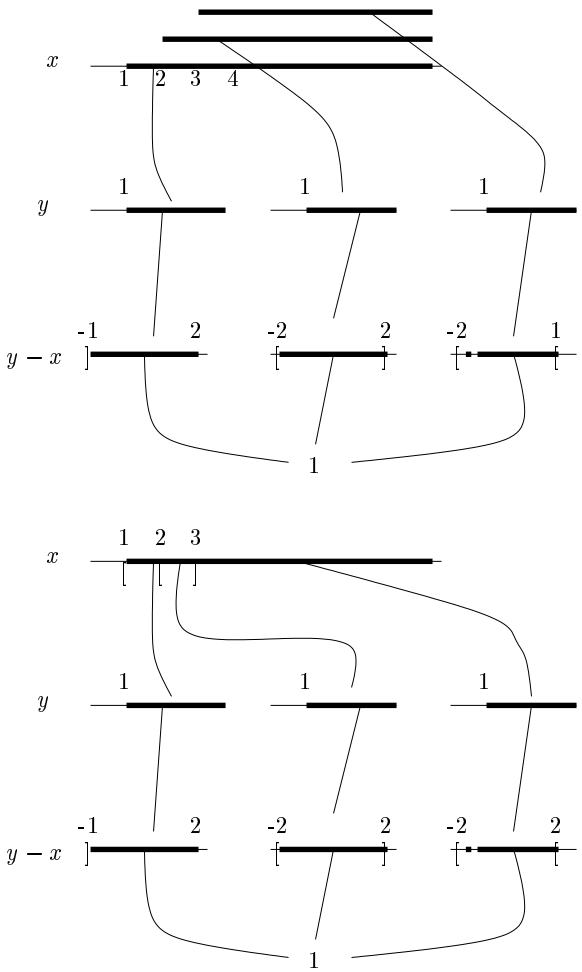

Figure 4: Computing the Future

4 Normal Form

The normal form we are using is closely related to the normal form (or canonical form) of DBM's. As a path in a CDD describes a zone, there is a translation from DBM's into paths and vice versa. Given a path, the DBM is constructed by filling in all the places where the path gives a constraint. The remaining places are just filled with the most general constraint (normally $+\infty$).

Given a DBM, a path is constructed by just filling in the constraints of the different types. In general, a DBM has two constraints for each type, defining the lower and the upper bound, thus giving the interval needed for the path.

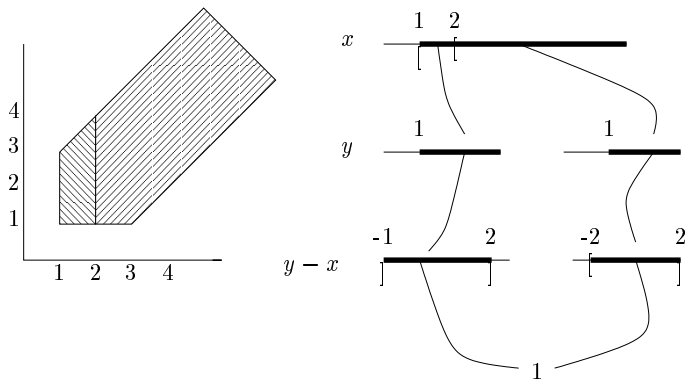

Figure 5: Computing the Future

To bring an arbitrary ordered and disjoint CDD T into normal form – i.e. each path to TRUE has only the tightest constraints – one may proceed as follows:

- start with the empty set as the result CDD R,
- for each path in the given CDD T leading to TRUE, compute its DBM,
- now compute the canonical DBM (using shortest path algorithms),
- construct a CDD P from this DBM, which has only one path leading to TRUE,
- now add P to R by computing $R \cup P$
- once the complete CDD R is constructed, start the process over again until the result is stable, i.e. the CDD one starts from is the same as the computed one

Note that termination of this procedure is guaranteed, as in the worst case all intervals will have the form $[c, c]$ or $(c, c+1)$, if they are not subintervals of $(-\infty, -M)$ or $(+M, +\infty)$.

In Figure 6 we give the normal form the disjoint version of Example 2. Note that only the consequences for $Y - X$ had to be added. Also all the CDD's in Figure 3 and Figure 4, 5 are in normal form.

Note that however for a given federation, in general there will be more than one CDD in normal form. In the following we will justify the term "normal form" despite this fact. The main idea here is that in addition to being in normal form the partitioning into intervals within a node is crucial for the structure of the CDD. We will now define what it means that a CDD is "at least as finely partitioned" or even "as finely partitioned" as another CDD.

Given two CDD's T_A and T_B, we say that T_A is *finer partitioned* than T_B iff

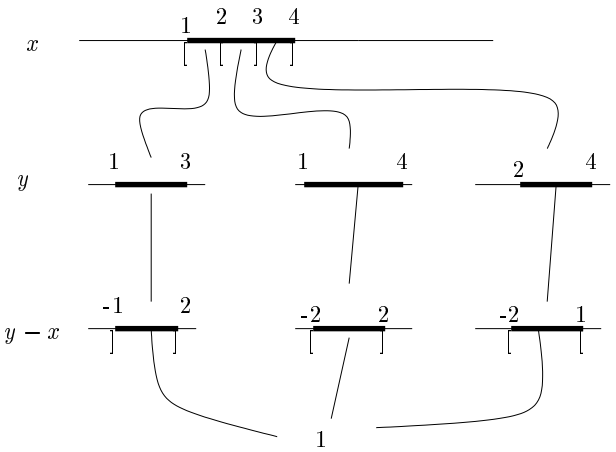

Figure 6: Normal Form

- either $T_A = False$,
- or $T_B = True$,
- or if T_A has the form $((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ and T_B the form $((i,j),(J_1,S_1),\ldots,(J_r,S_r))$, and for each $n \in \{1,\ldots,q\}$ there is $m \in \{1,\ldots,r\}$ such that $I_n \subseteq J_m$ and T_n is finer partitioned than S_m.

Note that if T_N is the CDD T after applying the normal from procedure, then T_N is always finer partitioned than T.

There is a simple way to turn a given CDD T_A into a CDD which is finer partitioned than a given CDD T_B. By $T_A \triangleleft T_B$ we describe the operation of making T_A as fine as T_B.

- if $T_B = True$ or $T_B = False$, then $T_A \triangleleft T_B := T_A$,
- if $T_A = False$, then $T_A \triangleleft T_B := T_A$,
- if $T_A = True$ and $T_B = ((i,j),(J_1,S_1),\ldots,(J_r,S_r))$, then $T_A \triangleleft T_B := ((i,j),(J_1,True \triangleleft S_1),\ldots,(J_r,True \triangleleft S_r))$.
- if $T_A, T_B \notin \{True, False\}$, then let $T_A = ((i,j),(I_1,T_1),\ldots,(I_q,T_q))$ and $T_B = ((i',j'),(J_1,S_1),\ldots,(J_r,S_r))$,
  - if $\mathsf{type}(T_A) \neq \mathsf{type}(T_B)$ and $\mathsf{type}(T_A) \sqsubseteq \mathsf{type}(T_B)$, then $T_A \triangleleft T_B := ((i,j),(I_1,T_1 \triangleleft T_B),\ldots,(I_q,T_q \triangleleft T_B))$
  - if $\mathsf{type}(T_A) \neq \mathsf{type}(T_B)$ and $\mathsf{type}(T_B) \sqsubseteq \mathsf{type}(T_A)$, then $T_A \triangleleft T_B := ((i',j'),(J_1,T_A \triangleleft S_1),\ldots,(J_r,T_A \triangleleft S_r))$

– if $\mathsf{type}(T_A) = \mathsf{type}(T_B)$, then $T_A \lhd T_B :=$

$$((i,j), \quad \begin{array}{ll} (I_1 \cap J_1, T_1 \lhd S_1), (I_1 \cap J_2, T_1 \lhd S_2), \ldots, & (I_1 \cap J_r, T_1 \lhd S_r), \\ (I_2 \cap J_1, T_2 \lhd S_1), \ldots, & \\ \ldots & (I_q \cap J_r, T_q \lhd S_r)) \end{array}$$

where empty intervals can safely be discarded.

If two CDD's are mutually finer partitioned, then we will call them *equally fine partitioned*. Using this notion, we can give a theorem which justifies our notion of normal form.

Theorem 1 *Let T_A, T_B be two CDD's which are equally fine partitioned and in normal form. Then $[\![T_A]\!] = [\![T_B]\!]$ iff T_A and T_B are graph-isomorphic.*

5 Deciding inclusion and equality

A basic question which arises in the reachability analysis of timed automata is if a given federation is included in or even equal to another. Obviously checking $T_A \subseteq T_B$ is the same as deciding if $T_A \cap T_B^{\complement}$ is the empty set.

We have already defined how to compute intersection. Complementing CDD's is very simple, one just needs to exchange the TRUE and FALSE nodes (due to completeness and disjointness). Testing for the empty set can be done by bringing a CDD into normal form. A CDD in normal form is the empty set iff all its end-nodes are FALSE.

The most costly operation involved in this is the computation of the normal form. Below we will comment on how to make this more efficient.

6 Local Transformations on CDD's

Bringing a CDD into normal form is a costly operation. In this section we point out how we can improve on this by using some local transformations on the CDD which does not change its semantics. These local transformations can then be used during other operations, or even as operations on their own, to make a CDD "more normal". At the end of the section, we will show how checking for emptyness of a CDD can be speeded up using the local transformations.

The main idea of the transformations is that we allow constraints to be propogated through the graph. These constraints can then be combined by the local information in a node, and can be used to simplify the nodes. They can also be combined in order to produce new constraints to be propagated in the CDD. This ideas are in fact extensions of the idea used in [LPW95] und [KLLPW97].

The most simple rule is that any interval in a node can propagate the constraint imposed by itself on the type of the node either up- or downwards the CDD. Further any constraint which arrives at a node can be forwarded to the other nodes up- and downwards in the graph. These very basic rules are illustrated in Figure 7.

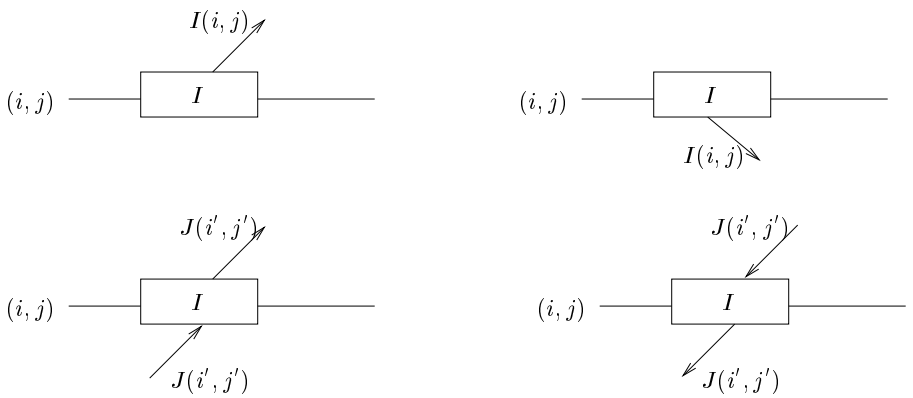

Figure 7: Simple Propagation

To make use of these rules, of course it is necessary to add some data structure to the CDD's which takes care of constraint propagation. There are many ways to do this, and we will not comment on the straight forward details here.

When a constraint of its own type arrives at a node, the node can decide to use this constraint to refine its partitioning into intervals. The stronger case is if the constraints arrives from a higher level in the graph. Then the constraint can be used to tighten the intervals in the node. However care must be taken if sharing is present, as the tightening will only be valid for the path from which the constraint was received. Therefore in the general case the node must be duplicated before tightening. This is illustrated in Figure 8. This operation is correct as we really can rely on the fact that the constraint is true within the path where I came from.

We can define this tightening formally. Assume a CDD $((i_0, j_0), (H_1, S_1), \ldots, (H_r, S_r))$ which sends the constraint $J(i,j)$ to its child S_m, which is of type (i,j). Let $S_m = ((i,j), (I_1, T_1), \ldots, (I_q, T_q))$. Then

$$S'_m = ((i,j), (I_1 \cap J, T_1), \ldots, (I_q \cap J, T_q)), (J^{\mathsf{C}}, False))$$

would be the tightening of S_m w.r.t. $J(i,j)$. We do not replace S_m by S'_m in the CDD (which would mean we would replace it for all subgraphs which share it), but we only replace it in the node the constraint $J(i,j)$ came from, i.e. the CDD now becomes $((i_0, j_0), (H_1, S_1), \ldots, (H_m, S'_m), \ldots, (H_r, S_r))$.

If the constraint comes from one of the paths starting in the node, then we cannot make such a strong tightening. However we can still split the interval,

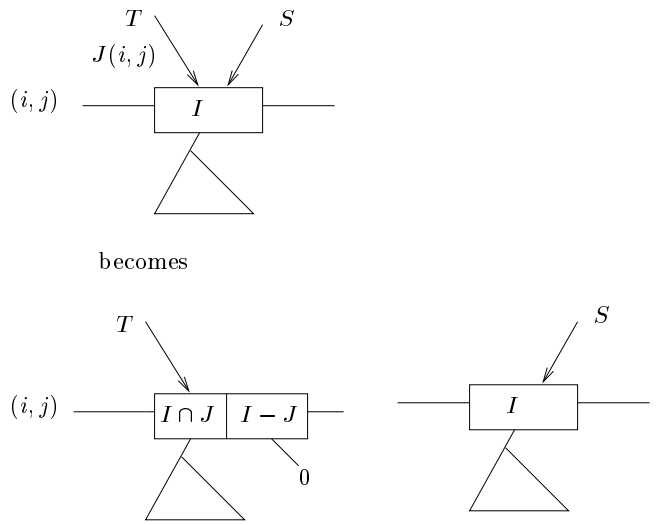

Figure 8: Tightening Intervals

hoping that this will lead to tighter bounds during further constraint propagation. Duplication happens in this step as well, see Figure 9

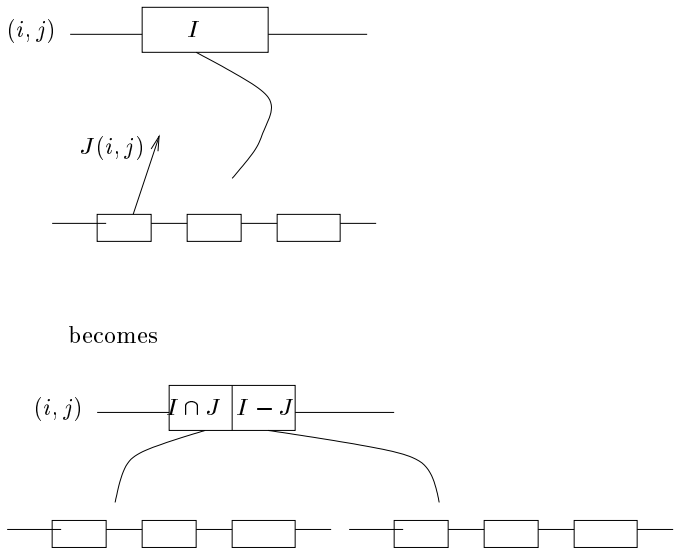

Figure 9: Tightening Intervals

So if a constraint $J(i,j)$ arrives from below at $((i,j),(I_1,T_1),\ldots,(I_q,T_q))$,

then this CDD can be turned into $((i,j),(I_1 \cap J, T_1),\ldots,(I_q \cap J, T_q),(I_1 \cap J^{\mathsf{C}}, T_1),\ldots,(I_q \cap J^{\mathsf{C}}, T_q))$.

Further we can compute new constraints by combining an arriving constraint and the constraint given by the interval in the node. This is done when the arriving constraint and the node have "neighbouring" types, i.e. there is a common index in the two types as e.g. in (i,j) and (i,k). Such neighbouring constraints can be combined into a new one by exploiting transitivity of the constraints, so the resulting constraint of the example would be of type (j,k) (assuming $j > k$). If $I(i,j)$ and $J(r,s)$ are two neighbouring constraints, then $I(i,j) \oplus J(r,s)$ is the combination of the two constraints.

The definition of this operation is rather straight forward, assuming we have interval addition $I + J := \{x + y \mid x \in I, y \in J\}$ and subtraction $I - J := \{x - y \mid x \in I, y \in J\}$. Then

$$\begin{array}{lll}
I(i,j) \oplus J(i',j') = & I + J(i,j') & j = i' \\
I(i,j) \oplus J(i',j') = & I + J(i',j) & i = j' \\
I(i,j) \oplus J(i',j') = & J - I(j,j') & i = i', j > j' \\
I(i,j) \oplus J(i',j') = & I - J(j',j) & i = i', j' > j \\
I(i,j) \oplus J(i',j') = & J - I(i',i) & j = j', i' > i \\
I(i,j) \oplus J(i',j') = & I - J(i,i') & j = j', i > i'
\end{array}$$

Figure 10 shows how to propagate combined constraints.

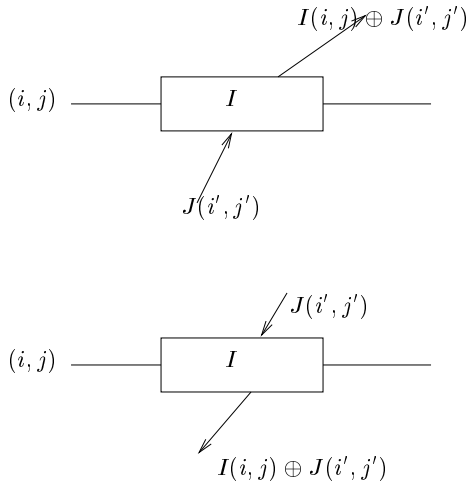

Figure 10: Combining Intervals

Note that all these rules only describe hwo to propagate simple constraints. They can of course be used to propgate more complex constraints as well.

As a simple example of how propagating constraints can be used we illustrate this with a simple example of testing for inclusion. The example is given in Figure 11 and 12.

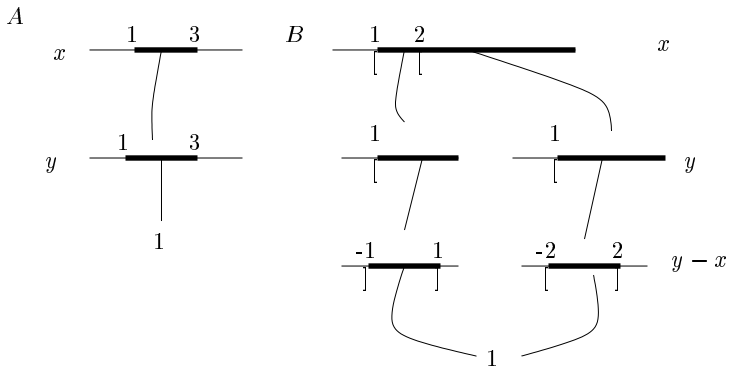

Figure 11: Testing for inclusion

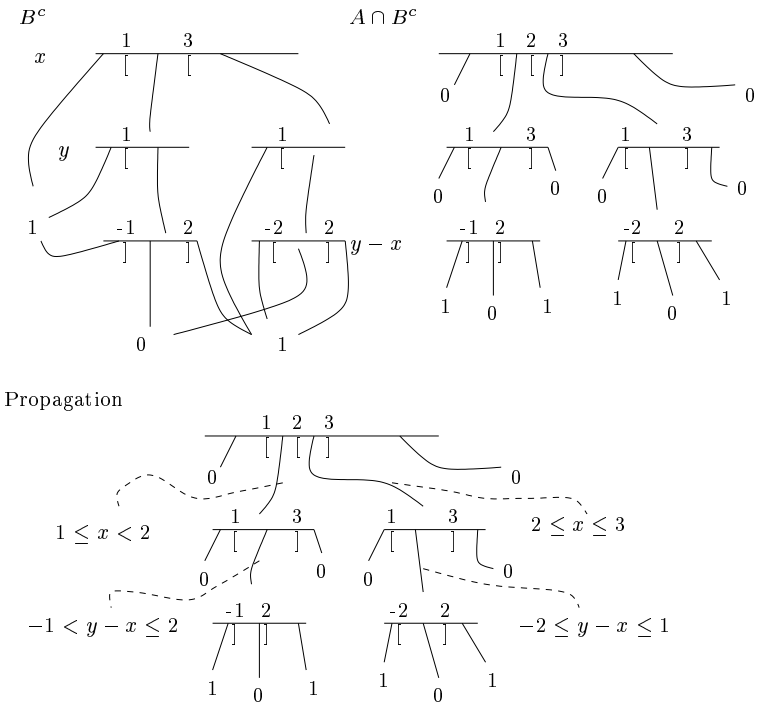

Figure 12: Testing for inclusion

The CDD A is the zone from example 2 which is more to the right (i.e. its has

smaller values for clock X). The CDD B is the future of example 2, as given in Figure 4. Obviously A must be included in B. Figure 11 shows how $A \cap B^{\mathsf{C}}$ looks like. Now we must test if this CDD is empty. The general approach would be to put it into normal form, and then check if all end-nodes are FALSE.

Here however we simply propagate the clock difference $Y - X$ downward. This is down by first propagating the constraint on X to the Y-nodes, and then propagating $Y - X$ further down. This immediately results in all end-nodes becoming FALSE by using downwards tightening. Already after five steps it can be decided that set inclusion holds, instead of going through the complete computation of the normal form.

So the basic approach for deciding emptiness of a CDD would be to propagate as much constraints as possible while traversing the graph once either upwards or downwards. In practice, often thereafter it will already be clear if the CDD is empty. Only if after traversing the CDD there are still paths leading to TRUE, the normal form must be computed. Note that for emptiness checking when traversing the graph from below, we need to start at the TRUE node only.

We also propose that for the computing of the future and setting a clock it is not necessary first to compute the normal form, but instead for the future it is only necessary to propagate the (real) differences between clocks downwards, while for the set operation the constraints implied by the clock's old value should be combined with all neighbouring constraints once while going through the graph downwards.

Acknowledgement

The first author of this paper was introduced to the idea of developing a BDD–like structure with nodes labelled by bounds on clock–differences by Henrik Reif Andersen.

References

[Bal96] Felice Balarin. *Approximate Reachability Analysis of Timed Automata.* Proc. Real-Time Systems Symposium, Washington, DC, December 1996, pp. 52–61.

[KLLPW97] Kåre J. Kristoffersen, Francois Larroussinie, Kim G. Larsen, Paul Pettersson and Wang Yi. *A Compositional Proof of a Real-Time Mutual Exclusion Protocol.* In Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Development. Lille, France, 14-18 April, 1997.

[LPW95] Kim G. Larsen, Paul Pettersson and Wang Yi. *Compositional and Symbolic Model-Checking of Real-Time Systems.* In Proceedings of the 16th IEEE Real-Time Systems Symposium, Pisa, Italy, 5-7 December, 1995.

[ST98] Karsten Strehl and Lothar Thiele. *Symbolic Model Checking Using Interval Diagram Techniques.* TIK Report No. 40, ETH Z"urich, February 1998.

[WL97] Carsten Weise and Dirk Lenzkes. *Efficient Scaling-Invariant Checking of Timed Bisimulation.* In: Proc. 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97), Lübeck, Germany, February/March 1997. LNCS 1200, pp. 177–188.

[WTD95] Howard Wong-Toi and David L. Dill. *Verification of real-time systems by successive over and under approximation.* International Conference on Computer-Aided Verification, July 1995.