Aalborg Universitet
Department of Computer Science

Abstract:

This project describes the design of a distributed implementation of the CDD data structure. CDD's are related to BDD's but handles interval instead binary values. The CDD is used in the real time verification tool Uppaal to store the symbolic part of its states, these states consists of both a discrete part and a symbolic part. This distribution is conducted for the primary reason of allowing verification of larger models. Besides designing the distribution of the data structure, four operations working on the data structure are designed. The semantics for the distributed data structure is described, and some semantic proofs of important properties for the operations are given.
The thesis tried to investigate, is whether using a single CDD data structure, compared to a number of smaller CDD's in a distributed verification environment, can save some memory by taking advantage of global sharing. That is, sharing in the symbolic representation, across several discrete states. Runtime is not considered as memory is the primary bottleneck in verification, the state space takes up several GB in minutes.
Finally tests are conducted to test the thesis, and to encounter the runtime penalty for these memory savings. The result shows that the memory saving were up to 70%, using a single CDD distributed over 4 nodes, compared to letting the 4 nodes holds four separate CDD's, this memory saving comes at a runtime penalty of 560-900%.
Besides trying to save memory by distributing the CDD data structure, some design were made to represent the CDD nodes as compact as possible. The memory representation saved up to 20% of memory compared to the memory representation used in an existing implementation of the CDD data structure. The runtime penalty for this memory saving is 50%.
During the tests we discovered that the problem of memory usage in Uppaal were not the storage of the symbolic part, but storing the discrete part. In most timed automata models, the discrete part takes up the majority of the used memory. As we had focused on the distribution of the symbolic part, we were not able to verify larger models in the distributed implementation, than on a single computer node.
In the last chapter some possible optimizations to the design/implementation is discussed. This description also includes a discussion of which part of this project might be used for other projects, trying to distribute decision diagram data structures. A possible data structure for storing the discrete part of the states, is also described briefly.

Titel:

Distributed CDD's
- interfacing Uppaal

Project period:

DAT6,
1. February 2002 -
7 June 2002

Group:

Christian Thomsen
Ronnie Kristensen

Supervisor:

Josva Kleist

Number printed: 6

Resume

Dette projekt beskriver designet af en distribueret implementation af CDD datastrukturen. CDD'er er relateret til BDD'er men håndterer intervaller istedet for binære værdier. CDD'er bruges i real tids verifications værktøjet Uppaal til at gemme den symbolske del af de tilstande der undersøges. Tilstande i Uppaal består af en diskret del og en symbolsk del. Distribueringen er lavet med det primære formål at tillade verifikation af større realtidsmodeller. Foruden at designe den distribuerede datastruktur, beskrives fire funktioner der arbejder på denne data struktur. Efter designet beskrives semantikken for datastrukturen, samt for nogle vigtige egenskaber for de designede funktioner.
Teorien der efterprøves er om en enkelt distribueret CDD datastruktur, sammelignet med et antal mindre CDD'ere i et distribueret netværk, kan spare noget hukommelse, igennem deling af symbolske tilstande mellem diskrete tilstande. For at spare så meget hukommelse som muligt, er hukommelses forbruget blevet overvejet igennem alle faser af designet, hvorved vi har accepteret et forøget tidsforbrug. Tidsforbrug er ikke taget i betragtning da hukommelse er den primære begrænsning i verificering, tilstandsrummet kommer på minutter til at fylde gigabytes.
Til sidst testes systemet for at se om teorien holder, og se hvilket forøget tidsforbrug distribueringen medfører, igennem synkronisering osv. Resultaterne viser at der kan spares op til 70% hukommelse, ved at distribuere en CDD datastruktur over fire computere, i forhold til at placere fire CDD'ere på de samme fire computere. Det forøgede tidsforbrug viste sig at ligge imellem 560% og 900%.
Foruden at spare hukommelse ved at distribuere CDD datastrukturen, er det også forsøgt at spare hukommelse ved at repræsentere CDD knuderne så kompakte som muligt. Hertil er der designet tre forskellige knude repræsentationer. Den mest besparende knude repræsentation sparede 20% hukommelse i forhold til knude repræsentationen i en eksisterende CDD implementation. Det maksimalt tilføjede tidsforbrug ved denne hukommelses repræsentation var 50%, men ved nogle modeller var det forøgede tidsforbrug minimalt, eller negativt (så vi sparede køretid).
Under testen fandt vi ud af at det største problem ved hukommelsees forbruget i Uppaal, ikke var at lagre den symbolske del, da denne kun optog en mindre del af det samlede hukommelses forbrug. Da vi havde fokuseret på distribueringen af den symbolske del, blev vi ikke i stand til at verificere større modeller, som var en del af vores vores formål.
I det sidste kapitel diskutere vi nogle optimeringer der er mulige til designet/implementeringen. Beskrivelsen indeholder også en beskrivelse af hvilke dele af vores projekt der kan bruges i andre projekter der forsøger at distribuere andre decision diagram data strukturerer. Til sidst beskrives en datastruktur til at lagre den diskrete del tilstandene.

Christian Thomsen

Ronnie Kristensen

Contents

1 Uppaal **3**

1.1 Overview . . . 3

1.2 Uppaal Engine . . . 5

1.3 Data Structures . . . 6

1.4 Versions of Uppaal . . . 9

2 Purpose **13**

2.1 Approach . . . 13

2.2 Purpose . . . 14

3 Data Structures **17**

3.1 Syntax . . . 17

3.2 Operations . . . 22

3.3 Current Uppaal . . . 22

3.4 Mapping CDD's to set formulas . . . 23

3.5 Mapping DBM's to CDD's . . . 23

4 Design / Data structures **25**

4.1 Non Distributed Algorithms . . . 25

4.2 Distributing the Data Structure . . . 31

4.3 Communication . . . 33

4.4 Operations . . . 34

4.5 Node Representation . . . 43

5 Semantics **49**

5.1 Semantics of the Distribution . . . 49

5.2 Data Structures . . . 50

5.3 Distribution . . . 51

5.4 Operations . . . 52

5.5 Union . . . 54

5.6 Semantics of Backtrace . . . 55
5.7 Reducing CDD's . . . 59

6 Cost Benefit Analysis **61**
6.1 Operations . . . 61
6.2 State Exploration . . . 62
6.3 Groups . . . 64
6.4 Memory Overhead . . . 64

7 Implementation **67**
7.1 Uppaal interface . . . 67
7.2 Use Pipelining . . . 68
7.3 Hash lists . . . 68
7.4 Distributed Garbage Collection . . . 68

8 Test **71**
8.1 Limitations of the Implementation . . . 71
8.2 Purpose of the test . . . 71
8.3 Premises . . . 72
8.4 Test description . . . 74
8.5 Expected Results . . . 74
8.6 Results and Analysis . . . 76
8.7 Summary . . . 83

9 Conclusion **85**

10 Future Work **87**
10.1 CDD Implementation of Waiting List . . . 87
10.2 Distributing the Discrete part . . . 88
10.3 Representing the Discrete Part as MTIDD . . . 89
10.4 Pack Messages . . . 91
10.5 CPU/Memory Load . . . 92
10.6 Distributed Shared Memory . . . 92

A Union/Reduction Example **95**

Introduction

During the last decade computer aided verification has established itself as a powerful technique for verifying whether a given formal model satisfies certain properties. In the last years tools has been extended to the verification of real-time systems, examples of such tools are Uppaal[11] and KRONOS[16].

The major problem in computer aided verification is the memory consumption caused by the state space exploration, as the space usage in worst case is exponential to the number of states in the verified system. System verification size can easily reach multiply of GB in less than an hour on a standard computer. There are several solutions to solving the state explosion problem, one is simply to buy a large mainframe computer, but a computer that is sufficient today might become insufficient tomorrow. Another problem to this approach is the cost, as large mainframe computers are expensive compared to standard cost of the shelf computers. Using standard computers though limits the memory usage to 4GB, for a 32 bit architecture. This leads to another solution to the state exploration problem, that can be solved by utilizing the memory on a network of workstations by distributing the verification process.

Several approaches tries to solve this problem in a more algorithmic way by designing compact data structures to store the states of the system being verified. The best data structure so far for the Uppaal verifier has been the clock difference diagram (CDD) that report average saving of 42% compared to the standard used data structure being difference bounded matrices (DBM). The cost for this saving is a modest increase in runtime of 7%[4].

This project aims at allowing Uppaal to verify larger model using a Network Of Workstations (NOW). which is done by distributing the CDD data structure.

Premises

Uppaal uses timed automata for verifying real time systems. A timed automata is a finite automata extended with variables and real valued clocks. The states used by Uppaal is divided into three categories:

Locations This part of the state represent in which location the timed automata is in.

Variables This part describes the valuation of all variable in the timed automata.

Zones This part represents the valuation of the clocks in the timed automata.

Locations and variables will in this report be denoted the *discrete part* of the state, and *Zones* is referred to as either *Zones* or the *symbolic part* of the state.

This project focus on representing the Zones part of the already searched states in Uppaal. This were chosen from the hypothesis that it were the zones that took up the main part of the memory. Many articles report on trying to representing the zones as compact as possible, example articles

on such is: [4], [10] and [9]. Therefore we decided to try to distribute the zones as it seemed to hold the greatest problems with respect to memory. Later in this project it has been revealed that this is not the case, as the discrete part normally uses the main part of the required memory during a verification. As this fact was discovered late in the project it has some influence on the outcome of the tests conducted in chapter 8. The report is structured chronologically as the project went on, but in chapter 1 we give a more in depth explanation of why the symbolic part did not account for the main part of the used memory.

this were not the case, and in chapter 8 we see what the consequences of only distributing the symbolic part of the state space, and finally in chapter 10 we discuss ideas on how this problem can be solved.

Related Work

The work in [4] describe the use of the CDD data structure in the Uppaal engine, which has inspired this project. Also the work in [3] on implementing a distributed version of Uppaal has been inspiring for the work made in this project.

Outline

The report is structured as follows:
First we give an introduction to Uppaal with a brief introduction to the used data structures.
After this the purpose of the project is stated.
Then a more in depth description of clock difference diagrams and difference bounded matrices with syntax is given.
The designed algorithms will follow these descriptions.
The semantics of the distributed data structures are then described together with semantics proofs of the algorithms designed.
After the semantics a cost/benefit analysis is conducted to reason on the expected memory/runtime overhead.
After the cost benefit analysis we describe the implementation of the distributed CDD data structure.
This is followed by a test of the implementation with an analysis of the results.
Finally we conclude on the project followed by some perspectives on the project.

Uppaal

Chapter 1

This chapter describes the verification tool Uppaal, first we give a general overview, followed by a description of Timed Automata, hereafter a description of the Uppaal engine including the used data structures, is given . Finally different versions of Uppaal are described.

1.1 Overview

Uppaal is a verification tool for real time systems based on constraint-solving. It is especially suited for systems that can be modeled as a collection of non-deterministic processes with a finite control structure and real valued clocks. Typical application areas are: Real time controllers and communication protocols where timing aspects are critical. Uppaal can check for reachability and invariant properties. The models used with Uppaal consists of a network of timed automata (TA),[1], extended with integer variables. Uppaal includes a graphical tool which allows the user to draw timed automata and run simulations.

1.1.1 Timed Automata

Timed automata has established itself as a standard for verifying real time systems. In this subsection we give an informal introduction to timed automata.

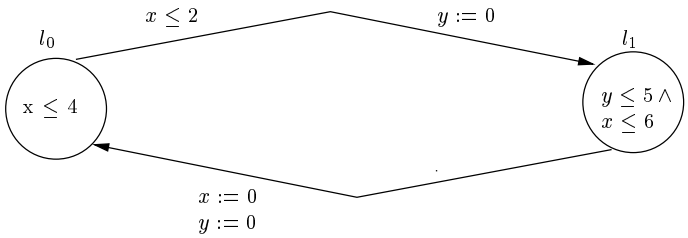


Figure 1.1: Timed automata

Figure 1.1 shows a timed automata with two locations: l_0 and l_1 and the real valued clocks x and y.

A timed automata is a finite state automaton extended with a finite collection of real valued clocks.

A formal definition of TAs is:
A timed automaton A is a 7 tuple $< L, l_0, E, C, clocks, guard, I >$

- L is a finite set of locations with l_0 being the start location.
- $E \subseteq L \times L$ is a set of transitions between locations.
- C is a finite set of clocks. (in the example x and y)

- *clocks* is a function that assigns each transition with set of clocks, to be reset to 0 when taking the transition.
- *guard* is a function that assigns each transition with a clock constraint (a guard) over C. A constraint is over a set of clocks and hold the following syntax: $X_i - X_j \sim n$ or $X_i \sim n$, where $X_i, X_j \in C$, $\sim \in \{<, \leq, \geq, >\}$, and n is an integer.
- I is a function that assigns an invariant over C to each location.

To stay at a location the invariant must be satisfied, likewise for taking a transition, the guard on the edge denoting the transition must be satisfied. Invariance of a location is described inside it, whereas guards for taking a transition is described on the edges in the graphical representation of the automaton.

A state of a timed automata A is a pair (l, D) where l is the discrete part of A and D represent the values of all clocks that A range over.

If the TA in figure 1.1 is in state $(l_0, \{0, 0\})$ meaning that it is currently at location l_0, with the clocks x, y both having the value zero. It can stay in this location letting time pass, as long as the invariant of l_0, being $x \leq 4$, is satisfied, and at least one transition is possible: $x \leq 2$. The transition to l_1 can only be taken when both the invariant of l_1 and the guard on the transition from l_0 to l_1 are satisfied thus requiring that $x \leq 2 \wedge x \leq 6 \wedge y \leq 5$. When taking the transition l_0 to l_1 the value of y is reset to zero.

To use timed automata for reachability analysis Alur and Dill's region technique [1] is used in Uppaal to represent the infinite state space of a TA as a finite collection of symbolic states. These symbolic states will represent the clock constraints of a system, and thus provide a convex subset of the Euclidean space. We will refer to these convex subsets as *Zones* with typical element Z. We define a *federation* to be any finite union of *Zones*, note that a *federation* is not necessarily convex see figure 1.2, where the *federation* P is the non convex union of the *Zones* Z_1 and Z_2, thus *Zones* are not closed under union.

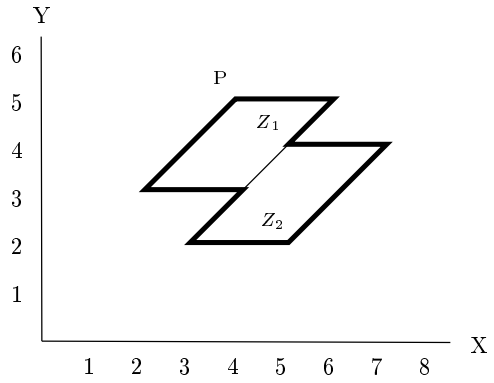


Figure 1.2: Two convex *Zones*, representing
$Z_1 : (2 \leq x \leq 6 \wedge 3 \leq y \leq 5 \wedge -1 \leq x - y \leq 1)$
$Z_2 : (3 \leq x \leq 7 \wedge 2 \leq y \leq 4 \wedge 1 \leq x - y \leq 3)$
Federation $\mathrm{P} = Z_1 \cup Z_2$

A TA can be traversed by following edges that is not violating any constraints in the destination location and the guards on the transition. Thus all reachable states $l \in L$ from a state l' are those:

– that does not contain any invariant that violates any clocks constraints,

– that the transition from l' to l does not have a guard that violates any clocks constraints.

Note that l' must be left before the invariants of this node is violated.

When reaching a new location the TA enters a new state, being (l, D), where l is the new discrete state (consisting of the location vector and the values of all integer variables) and D is the set of all clock values representing the possible *Zone*.

Constraints

As mentioned earlier constraints may have the form:

- $X_i - X_j \sim n$, or
- $X_i \sim n$

where $X_i, X_j \in C$, $\sim \in \{<, \leq, \geq, >\}$, and $n \in \mathbb{N}$. In this subsection we argue that only constraints of the form: $X_i - X_j \leq n$ is necessary for simulating the others. First we will argue that $X_i \sim n$ can be simulated by $X_i - X_j \sim n$. This is done by introducing a special zero clock X_0 whose value is always 0. Then $X_i \sim n$, may be simulated by $X_i - X_0 \sim n$. Next we will argue that $>$ and $\geq$ can be simulated by $<$ and $\leq$ respectively. This is done by negation:

- $X_i - X_j > n \Leftrightarrow X_j - X_i < -n$
- $X_i - X_j \geq n \Leftrightarrow X_j - X_i \leq -n$

The last property to show is that $X_i - X_j < n$ can be simulated by $X_i - X_j \leq n$, which it cannot be in general, but it becomes possible as all guards and invariants only check on integer values ($n \in \mathbb{N}$). The method used is by multiplying all constraints by two on both sides, and if the inequality sign is $<$ the bound is subtracted by one, that is the rules is translated to:

- $X_i - X_j < n \rightsquigarrow 2 \cdot (X_i - X_j) \leq 2 \cdot n - 1$
- $X_i - X_j \leq n \rightsquigarrow 2 \cdot (X_i - X_j) \leq 2 \cdot n$

1.2 Uppaal Engine

The verification engine of Uppaal is called *verifyta*, it holds the responsibility for doing the actual verification, that is reachability analysis and checking invariance properties.
The *verifyta* is given a network of TAs and a formula to check as input. For reachability analysis the algorithm in figure 1.3 is used, where φ is the formula for which it should be fulfilled.

This algorithm uses two lists, a *Passed* list and a *Waiting* list. The *Passed* list denotes all the states seen so far, used to avoid exploring a state twice, and thus assuring termination when the total state space has been searched. The *Waiting* list is a queue of states waiting to be explored. The states held in these lists are of the form (l, D) where l denotes the discrete part, being a vector telling in which discrete nodes all the TA's are in, together with the values of all variables

```
1. Passed := {}
2. WAITING := {(l_0, D_0)}
repeat
    begin
      3. get (l, D) from WAITING
      4. if (l, D) ⊨ φ then return YES
      5. else if D ⊈ D' for all (l, D') ∈ Passed then
            begin
              6. add (l, D) to Passed
              7. SUCC := {(l_s, D_s) : (l, D) ⇝ (l_s, D_s) ∧ D_s ≠ ∅}
              8. for all (l_s', D_s') in SUCC do
                    9. put (l_s', D_s') to WAITING
            end
    end
10. until WAITING = {}
11. return NO
```

Figure 1.3: An algorithm for symbolic reachability analysis.

of the TA's. D is a *Zone* representing the clock constraints. The algorithm works as we starts out with an empty *Passed* list and with the initial state as the only state in the *Waiting* list (lines 1 and 2).

We repeatedly takes a new state (l, D) from the *Waiting* list (line 3), checks if it satisfies the formulae φ, if not we perform an inclusion test to see whether (l, D) has already been explored (line 5). If we fail the inclusion test (l, D) is added to the *Passed* list and all states that is reachable from (l, D) is added to the *Waiting* list (line 7). If we have searched the entire state space without finding a state satisfying the formulae φ, we return NO (line 10 and 11). We can "invert" the algorithm by interchanging the YES and the NO (line 4 and 11), to obtain an algorithm that checks for invariants for the $\neg\varphi$ formulae:

$$(\neg\, \exists\, d \in STATESPACE \,|\, d \models \varphi) \;\equiv\; (\forall\, d \in STATESPACE \,|\, d \models \neg\, \varphi)$$

1.3 Data Structures

This section describes the data structures used to represent the *Passed* list. As this list is used to store previously explored state, it will at the end of a full state space search represent the entire symbolic state space, implicating that we must make this list as compact as possible. The following subsection describes two data structures used to make a compact representation of this part of the *Passed* list, namely the **difference bounded matrices** and **clock difference diagrams**. The algorithm in figure 1.3 shows that there is a need for an efficient inclusion test on the data structure holding the *Passed* list (line 5). Another action that the data structure must support is union of the *Passed* list with a new state (line 6)

1.3.1 Difference Bounded Matrices

Difference bounded matrices was first proposed in [5], later it were use for constraint systems as they can offer a canonical representation of a such.

Definition

A DBM representation of a constraint system D is a weighted directed graph $G = (V,E)$ where the vertices V describes the clocks in C and an additional zero vertex, that designates a clock whose value is always zero. There is an edge E from x to y with weight m if there is a constraint of the type $x - y \leq m$. Also there is an edge from x to the zero vertex with weight m if there is a constraint of the type $x \leq m$.

Description

The canonical representation that DBM's can offer, is when they are closed. To describe the closedness of a constraint system, we need to calculate the shortest path closure of the graph describing the constraint system. Standard algorithms such as Bellman-Ford can do this in $O(n^3)$ with n being the number vertices in the graph (the number of clocks in the constraint system).

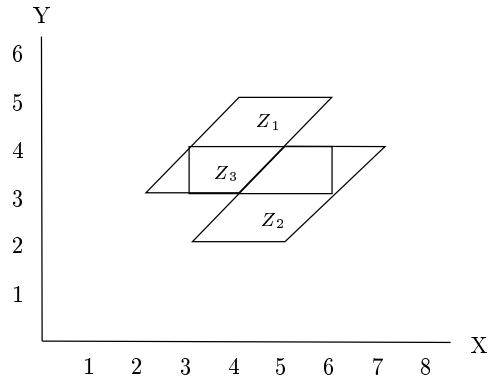


Figure 1.4: Three convex *Zones*, representing by the closure
$Z_1 : (2 \leq x \leq 6) \wedge (3 \leq y \leq 5) \wedge (-1 \leq x - y \leq 1)$
$Z_2 : (3 \leq x \leq 7) \wedge (2 \leq y \leq 4) \wedge (1 \leq x - y \leq 3)$
$Z_3 : (3 \leq x \leq 6) \wedge (3 \leq y \leq 4) \wedge (-1 \leq x - y \leq 3)$

The shortest path closure of a constraint system contains redundant constraints, the constraint for figure 1.4(Z_1) could be described only by the constraints :
$(3 \leq y \leq 5) \wedge (-1 \leq x - y \leq 1)$
meaning that the last constraints for x is implicitly given by the others.

As the inclusion test $D \subseteq D'$ runs in $O(n)$ with n being the number of constraints in D', saving as few constraints as possible is desirable.

To do so, an $O(n^3)$ algorithm has been developed to calculate the shortest path reduction, that converts a DBM in shortest path closure, to an equivalent reduced system with a minimal number of constraints. The algorithm works essentially by saving all zero cycles in the graph together with the edges that interconnect these zero cycles. The algorithm is described in [4].

DBM's are limited as they only describe convex *Zones*. If the constraint system D is included in the union of more than one DBM as $Z_3 \subseteq (Z_1 \cup Z_2)$ in figure 1.4. The inclusion tests $Z_3 \subseteq Z_1$ and $Z_3 \subseteq Z_2$ will both fail despite the fact that Z_3 already has been explored, partly in Z_2 and partly in Z_1, thus forcing redundant state exploration and redundant storing of Z_3.

Uppaal uses DBM's to store the symbolic part of a state (the D part of the state (l, D)), thus for every discrete state l there is a number of DBM's, that together describes the searched *federation* for this discrete state. These DBM's are stored in the *Passed* list of figure 1.3. The space used to store DBM's are $O(n^2)$ for the closed form. The average case is a lot better for the DBM in its reduced form, as most closed DBM's contains redundant constraints. [10] reports on savings up to 97% for using reduced DBM's instead of closed DBM's.

1.3.2 Clock Difference Diagrams

In this subsection we present an informal description of clock difference diagrams (CDD), which is an extension to Reduced Ordered Binary Decision Diagrams (ROBDD) presented by Randal E. Bryant in [6]. CDD was first introduced by Larsen et al. in [8], as a data structure to store constraints in a constraint system. In Uppaal CDD's are used to store the symbolic part of the *Passed* list of the algorithm in figure 1.3. The CDD data structure is greatly inspired by the IDD data structure described by [14].

Definition

A CDD is a directed acyclic graph $T = (V, E)$ where V are vertices of two kinds, either inner nodes or terminal nodes.
A clock constraint is of the form $X_i - X_j \leq m$ with X_i and X_j being real valued clocks with integer bound m. For any constraint (X_i, X_j) is the type of the constraint. Inner nodes has a type and a finite number of successor nodes each representing an interval of reals with integer bounds referring to another CDD node. Terminal nodes are either **true** or **false** and have no successors. All types of nodes must be globally ordered meaning that when traversing a path in a CDD the types are increasing, and no types will appear twice in the path. A recursive definition of CDD's are :
$\langle (X_i, X_j), [I_1, T_1], \ldots, [I_n, T_n] \rangle$ where (X_i, X_j) is the type of the node, and I_i is an interval of reals, and T_i is a CDD.

The union of the intervals must be complete, thus $\bigcup_{i \in \{1,\ldots n\}} I_i = \mathbb{R}$ The intervals must be disjoint so $\forall I_i, I_j \,|\, i \neq j \,.I_i \cap Ij \; = \emptyset$

An example CDD is pictured in figure 1.5, which describes the *federation* $P = Z_1 \cup Z_2$ of figure 1.2.

An interval $I = [a; b]$ for the type (X_i, X_j) represents the clock constraint:
$X_i - X_j \leq a \;\wedge X_j - X_i \leq b$.

Note that we omit all edges leading to **false** for simplicity reasons, this will apply to all figures through out the report.

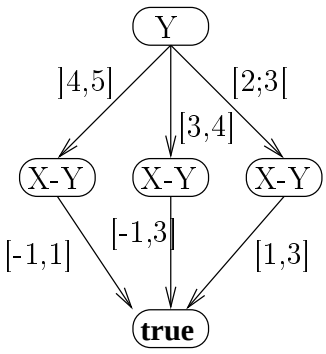


Figure 1.5: The CDD for describing the non convex *federation* $P = Z_1 \cup Z_2$ of figure 1.2 All edges not represented leads to the **false** node, these are omitted for simplicity.

Description

A CDD describes a *federation* meaning that the previously described problem of redundant state exploration using DBM's can be eliminated. The inclusion test of the 'Z_3' *Zone* of figure 1.4 will succeed as the shortest path closed constraint system is described by $(3 \leq x \leq 6) \wedge (3 \leq y \leq 4) \wedge (-1 \leq (x - y) \leq 3)$, and these constraints are all included in the CDD in figure 1.5. Note that there are no x CDD node represented, meaning that this node implicitly cover $\mathbb{R}$.
CDD's allow sharing of clock constraints over location borders. That is if two zones from two different discrete states l and l' share some common constraints the CDD data structure allow sharing between these by giving the CDD several handles - namely one handle per discrete state.

The memory usage of CDD are difficult to reason about, as the sharing between handles are hard to foresee. Also the sharing within the same location node is hard to foresee, theoretical the size is exponential. In the next section we will state some experimental results about the memory usage of CDD's compared to reduced DBM's, which shows considerable memory-savings.

1.4 Versions of Uppaal

This section gives an overview of the different Uppaal versions that will be referred to in this project.

1.4.1 Sequential Uppaal

There are two sequential version of Uppaal, one that implements the DBM data structure and one that implements the CDD data structure.

- The basic version of Uppaal uses a shortest-path reduced form of the DBM data structure[10]. This implementation showed space savings between 74% and 97%, compared to an Uppaal implementation using standard non reduced DBM's.
- In [4] Uppaal is tested with the CDD data structure, which compared to the shortest-path reduced form of the DBM's saves an additional 42% in average with moderate increase in runtime (7%).

1.4.2 Distributed Uppaal

[3] present a distributed engine for Uppaal based on the DBM data structure for storing the searched states. This distributed Uppaal allowed verifying larger models by distributing the state space. The state space is distributed in the following manner: (please refer to the algorithm in figure: 1.3)
Whenever a new state $(l_{s'}, D_{s'})$ is found in line 8, the state is hashed to a specific computer node responsible for this state. When the node responsible for $(l_{s'}, D_{s'})$ receives this request, it simply puts it into it's *Waiting* list, for later exploration and possible storage.

Each node holds a part of the global *Passed* list, this part contains symbolic convex *Zones* for all discrete states which the node is responsible for.

The current distributed Uppaal uses DBM's for storage of the symbolic part of the state space. Using the CDD data structure would implicate that each node holds a single CDD data structure instead of several lists of DBM's. This scheme allow two kind of sharing.

- The CDD data structure holds the *federation* searched instead of holding a set of *Zones*, this removes the problem of redundant state exploration as described earlier.

- Two different discrete states might share common constraints. If such two discrete states were located on the same node the shared constraints need only be stored once. But stored on two different nodes will not lead to any space reduction.

1.4.3 Premises

As mentioned in the introduction a more in depth description of why we chose to distribute the CDD data structure is given in this subsection. The CDD data structure was chosen as it was expected to account for the main part of the memory memory used during a verification, this was derived from, as previously mentioned a lot of articles on this topic, but also from the fact that the number of types in the CDD is quadratic to the number of clocks, whereas the entries in a discrete state is linear to the number of locations and variables in the timed automata. Meaning that complexity wise it is to be expected that the clocks will account for a higher memory usage that the variables. This has not been the case primarily from two reasons, first even though the clocks are quadratic in size compared to the variables and locations, the "normal" layout of a Uppaal model make use of the same number of locations and variables as the quadratic size of the clocks. Secondly the behavior of the Uppaal models it that there are many states where the locations and the variables differ from other states but the clocks values are the same, meaning that the locations and variables are stored twice whereas the clocks are only stored once.

In chapter 8 we see what consequences this mismatch between our interpretation of the Uppaal behavior and the actual behavior gives.

The report continues as if the problem still were the representation of the symbolic part, until chapter 8 were the consequences of our misinterpretation is given.

1.4.4 Summary

The different versions of Uppaal together indicates that it might be possible to verify larger models if we can distribute the CDD data structure among several computer nodes. In this way we can utilize the larger amount of memory available in a network of workstations, and at the same time take advantage of the memory savings that the sequential version shows. Again verification is very memory intensive and not especially CPU dependent, as the state space explodes exponentially with the size of the model.

After stating the used data structures in Uppaal, together with the different version, are we ready to state the purpose of this project.

Purpose

Chapter 2

The purpose of this project is to investigate what distributed computing has to offer in the verification of timed automata. As mentioned in the introduction formal verification is very memory intensive, therefore the main purpose of this project is to utilize the larger amount of memory available in a distributed system efficiently, in the area of formal verification.

2.1 Approach

The purpose of this project is to distribute Uppaal to be able to verify larger models. Our purpose is increased memory-savings in favor of increased speed. Therefore this project tries to design and implement a distributed CDD data structure for storing the symbolic part of the *Passed* list stored by Uppaal.

The CDD is distributed by partitioning the CDD and store different partitions on different computer nodes. The current distributed Uppaal distributes the *Passed* list by distributing DBM's to computer nodes, by hashing on the discrete part of the state. A single extended distributed Uppaal implementation using CDD's, is to use that same approach, by storing a single CDD on each node. A further improvement could be to make a single CDD span all computer nodes, there by archive sharing between *all* discrete states.

The differences between the mentioned distribution possibilities of Uppaal is depicted in figure 2.1

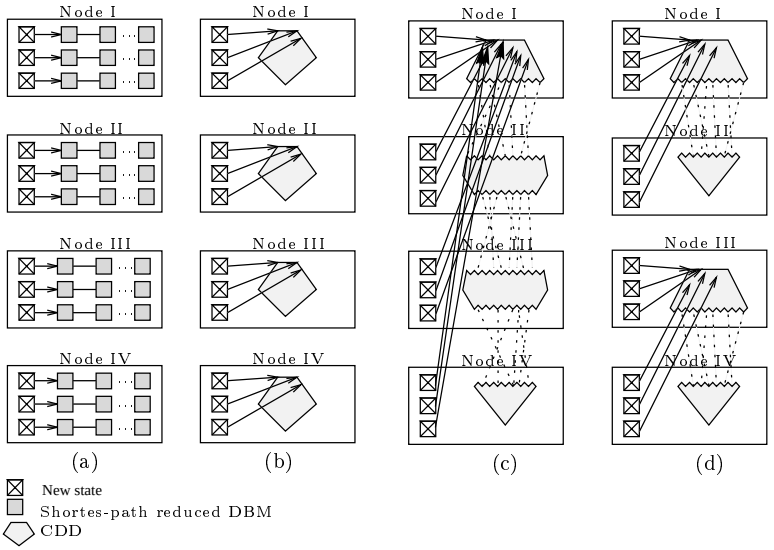


Figure 2.1: Four approaches for distributing Uppaal.
Figure (a) show the current distribution approach used.
Figure (b) show an approach similar to the current distribution Uppaal version, but using the CDD data structure.
Figure (c) show a CDD data structure distributed across all nodes participating in the verification.
Figure (d) shows a hybrid between (b) and (c), with more than one CDD, all spanning more than one node.

The approach of using a single distributed CDD has some advantages as well as disadvantages, which will be discussed in the following.

Characteristics of a Single Distributed CDD

Global Sharing The most obvious advantage of this approach is naturally that it allow sharing between all discrete states.

Heterogeneous Another advantage of the distributed CDD approach compared to the other two approaches is that the distributed CDD approach handles heterogeneous configurations better. If e.g. some nodes has more memory than others special hash functions has to be implemented in the current distributed Uppaal, whereas in the distributed CDD approach it is possible to implement runtime memory distribution by moving a CDD layer from one computer node to another.

Dynamic load sharing In the current distributed Uppaal only the computer node responsible for a discrete state hold information about which *Zones* has been explored, therefore such states have to be send to this computer node. This may introduce a bottleneck problem if at a certain time many states with the same hash value is found, then the responsible computer node become a bottleneck. When using only one CDD, all computer nodes can explore all states which means that tasks should only be send to other computer nodes when these has an empty *Waiting* lists. If a computer node become a bottleneck, it may simply distribute it's *Waiting* list to the other participating computer nodes.

Decreased scalability A problem with the distributed CDD approach compared to the other distribution approaches is that the distributed CDD approach might not scale to many computer nodes, it time performance might decrease. The reason for this expected decrease in speed performance is that whenever a new state should be explored an inclusion test must be performed. To make a complete inclusion test, CDD nodes on every computer node must be searched, that is at least $k-1$ synchronization messages must be send for each inclusion test in a system with k computer nodes. The same counts for union - when the state has been searched it must be unioned into the existing CDD, this union also involve at least $k-1$ synchronization messages. Further cost/benefit analysis is conducted in chapter 6.

2.2 Purpose

The tests on the non distributed CDD version of Uppaal, showed significantly memory savings being 42% compared to the shortest path reduced DBM version. This should also apply to the distributed version, but as a decrease in time performance is expected due to the larger synchronization overhead introduced by the distributed CDD approach, we will try to find the trade off for how many computer nodes a distributed CDD should span. The assumptions is that the more computer nodes a CDD span the larger the overhead in synchronization is, but it gives possibility of reducing the overall memory usage. On the contrary, a CDD spanning fewer computer nodes minimize the synchronization overhead, but does not allow maximum memory saving in form of global sharing. Trying different hybrid version as the one in figure 2.1(d) is a way to find this trade off. This way it is possible to see how much memory can be saved utilizing global sharing, and at which runtime cost this memory-saving comes at. To summarize our purpose is:

How much memory can be saved by global sharing through the CDD data structure for saving the symbolic part of the Passed list in formal verification of timed automata. Find the relation ship between synchronization overhead/ memory usage when using a hybrid CDD model where the number of CDD's are ranging between one and the number of computer nodes used.

Data Structures

Chapter 3

This chapter describes the data structures which are used in this project, how they are used and some syntax for these.
This project uses two main data structures:

CDD The main data structure in this project is the CDD data structure, which stores the symbolic part of the *Passed* list in Uppaal. This data structure is to be distributed over all participating computer nodes, or within a group of computer nodes.
Most decision diagram data structures holds a single handle (one node in the top of the graph representing the value of the first variable in the variable ordering), but the CDD data structure allow several handles to allow maximal sharing between control states, figure 3.1 show an example on sharing between control states using two handles.

DBM All communication with the Uppaal engine is via the DBM data structure. To perform the *inclusion test* (figure 1.3 line 5) the DBM representation for the new state is converted into its shortest path closure before the *inclusion test* is performed, as this describes all constraints that must be satisfied for the *inclusion test* to succeed. When a state should be unioned into the *Passed* list the same DBM is converted into its shortest path reduced counter part(converted to a CDD), before inserted into the CDD, as this DBM contains the minimum required information for describing the constraint system.

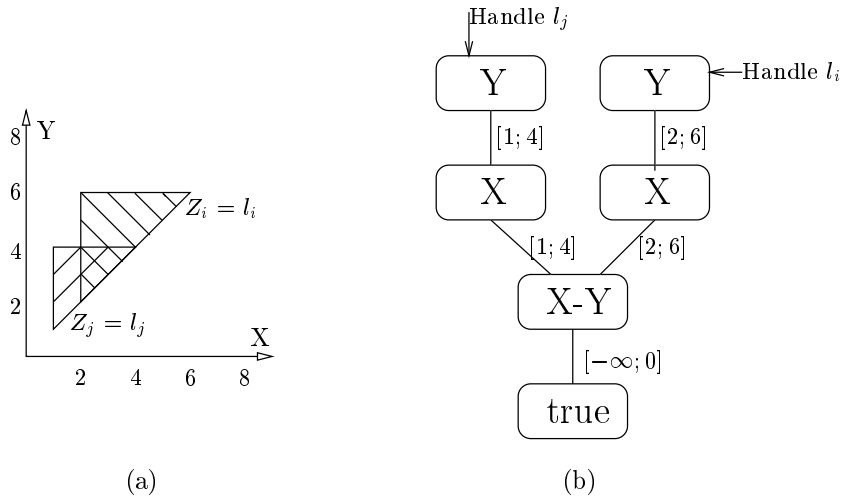


Figure 3.1: Example on sharing between two different discrete states. The (a) part show two *Zones* Z_1 and Z_2 belonging to two different discrete states. Figure (b) show the CDD representing these two areas. This CDD has two handles, one for each discrete state. As can be seen the two discrete states share CDD nodes.

3.1 Syntax

Before any other properties for the data structures is given, some syntax is provided to support further descriptions.

3.1.1 Clock Difference Diagrams

The CDD data structure is used to store the *Passed* list in Uppaal, that is the CDD data structure only holds the symbolic part of the *Passed* list, with the discrete part given by handles into the CDD data structure.
CDD's are used to store the constraints $(X_i - X_j \leq n)$ which together spans the union of searched *Zones* for each discrete state (a *federation*).
The set of all constraint for a given discrete location l in a timed automata A, is denoted D, with typical element c. A constraint is a three-tuple (X_i, X_j, n) for the constraint $X_i - X_j \leq n$

A CDD is a directed acyclic graph (DAG), typically denoted T with two kinds of CDD nodes: Inner and terminal nodes. Terminal nodes represents the constants **true** and **false**, while inner nodes are associated with a *type* (X_i, X_j) where $i, j \in \{0..n\}, i \neq j$[1]. Arcs labeled with interval bounds of the difference of the clocks given by the type (X_i, X_j). An interval $I = [a; b]$ for the type (X_i, X_j) represents the clock constraint $X_i - X_j \leq a \ \wedge \ X_j - X_i \leq b$.

Example CDD's are shown in fig. 3.2.

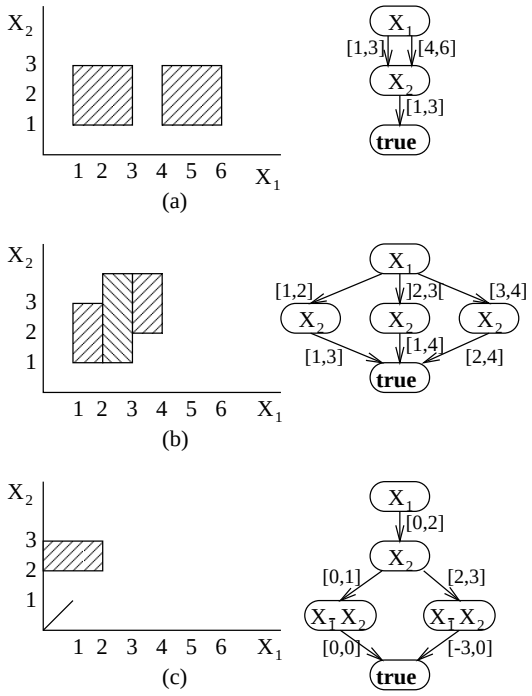


Figure 3.2: Three example CDD's. Intervals not shown implicitly leads to **false**; e.g. in (a) there are arcs from the X_1-node to **false** for the three intervals $]-\infty, 1[,]3, 4[$, and $]6, \infty[$.

A *type* is a pair (X_i, X_j), which corresponds to an inner node in the CDD. The set of all types is written $\mathcal{T}$, with typical element t. $\mathcal{T}$ is assumed to be equipped with a linear ordering $\sqsubset$ and a special bottom element $\perp$, in the same way as BDD's assume a given ordering on the boolean variables. The bottom elements being **false** and **true** has the largest type, whereas the handle has the smallest type.

[1]Remember X_0 is the zero clock always being zero.

A possible ordering could be:

- $(X_{i_1}, X_{j_1}) \sqsubset (X_{i_2}, X_{j_2})$ **if** $j_1 < j_2$
- $(X_{i_1}, X_{j_1}) \sqsubset (X_{i_2}, X_{j_2})$ **if** $j_1 = j_2 \wedge i_1 < i_2$.

Let the function $\mathit{first}(T : CDD)$ return the uppermost CDD node in the CDD T, being the handle.

$\mathcal{I}$ denotes the set of all non-empty, convex, inter-bounded subsets of the real line. $\mathcal{I}$ contains both open, closed and half-open intervals, which is: $]a, b[$, $[a, b]$, $]a, b]$, and $[a, b[$. A typical element of $\mathcal{I}$ is denoted I.
Two intervals are named *neighbored* if they may be joined by union into a larger interval - overlapping intervals are called neighbors too.

An interval cover for a CDD node n_i is denoted $I(n_i)$, and describes the intervals leaving n_i. That is, an interval cover is a set of intervals $I_{n_i} = \{I_1, I_2, \ldots, I_n\}$. Each interval in the interval partition $I_j \in I(n_i)$ must be a subset of the total interval leaving n_i. That is, for nodes having types $X_i - X_0,\ i \neq 0, I_j \subseteq \mathbb{R}^+$, and for nodes having types $X_i - X_j\ i, j \neq 0\ I_j \subseteq \mathbb{R}$. Furthermore an interval partition must be complete, i.e.,

- If $\mathit{type}(n_i) = (X_i - X_0)\ i \neq 0$: $\bigcup_{I \in I(n_i)} I = \mathbb{R}^+$
- If $\mathit{type}(n_i) = (X_i - X_j)\ i, j \neq 0$: $\bigcup_{I \in I(n_i)} I = \mathbb{R}$

$I(n_{X-Y})$ in figure 3.1(b) would be: $I(n_{X-Y}) = \{\,]-\infty; 0]\,;\,]0; \infty[\,\}$, the last interval being omitted in the figure, as it leads to **false**.

An interval partition is a disjoint interval cover. That is, an interval cover is an interval partition if:

$$\forall j, k \in \{1 \ldots n\} | j \neq k : I_j \cap I_k = \emptyset$$

An interval partition is ordered if the (lower/higher) bounds of all intervals build an increasing sequence. An interval partition which is ordered is named *reduced interval partition*.

$I(i, j)$ denote the clock constraint having type (i, j) which restricts the value of $X_i - X_j$ to the interval I.
Given a set of clock constraints D and a valuation v, $D(v)$ denote the boolean value derived from replacing the clocks in D by the values given by v.
The prior two notations will be used jointly, i.e. $I(i, j)(v)$ expresses the fact that v fulfills the constraint given by I and the type (i, j). $v(t)$ defines the current value of the t type given by the valuation v.

We define a CDD node as a $n+1$ tuple $\langle t, [I_1, T_1] \ldots [I_n, T_n] \rangle$ where t is the type, and successors $T_1 \ldots T_n$ being CDD's (which can be viewed as a CDD nodes, where $\mathit{first}(T_i)$ is the handle of the successor CDD), each denoting the corresponding interval $I_1 \ldots I_n$.
The set of all CDD nodes is denoted $\mathcal{N}$ with typical elements n, m.
It should be noted from the previous discussion that a CDD and a CDD-node can be used interchangeable, as a CDD node and its successors may be interpreted as a CDD. And a CDD may be interpreted as a CDD node by using the handle.

A CDD is a DAG consisting of a set of nodes $V \subseteq \mathcal{N}$ and two functions:

- $type : V \to \mathcal{T}$
- $succ : V \to 2^{\mathcal{I} \times V}$ such that:

  V has exactly two terminal nodes called **true** and **false**, where $type(\textbf{true}) = type(\textbf{false}) = \bot$ and $succ(\textbf{true}) = succ(\textbf{false}) = \emptyset$

  all other nodes $n \in V$ are inner nodes, which have attributed a type $type(n) \in \mathcal{T}$ and a finite set of successors $succ(n) = \{(I_1, n_1), \ldots, (I_k, n_k)\}$, where $(I_i, n_i) \in \mathcal{I} \times V$.

$n \xrightarrow{I} m$ is shorthand for $(I, m) \in succ(n)$.
For each inner node the following must hold:

- the successors are disjoint: $\forall (I, m), (I', m') \in succ(n)$ either $(I, m) = (I', m')$ or $I \cap I' = \emptyset$,
- the successor set is an $\mathbb{R}$-cover: $\bigcup \{I | \exists m.n \xrightarrow{I} m\} = \mathbb{R}$,
- the CDD is ordered: for all m, whenever $n \xrightarrow{I} m$ then $type(n) \sqsubset type(m)$.

The CDD is assumed to be reduced if the following holds:

- it has maximal sharing: for all $n, m \in V$ whenever $succ(n) = succ(m) \wedge type(n) = type(m)$ then $n = m$, that is no isomorphic sub CDD's can coexists in the CDD.
- all intervals are maximal: whenever $n \xrightarrow{I_1} m, n \xrightarrow{I_2} m$ then $I_1 = I_2$ or $I_1 \cup I_2 \notin \mathcal{I}$.

We define the function $child : V \times \mathcal{I} \to V$, such that $child(n, I)$ to return the successor node m of n where the edge going from n to m is labeled with the interval I. Thus for $m = child(n, I)$, then there exists a successor to n so $n \xrightarrow{I} m$.

S-CDD's

When mapping a DBM to a CDD, the resulting CDD will be a CDD where all nodes only have a single successor node not being the **false** node, thus to make semantic proofs easier we define a special syntax here for such a CDD, denoted S-CDD.
S-CDD's are CDD where all nodes have a single successor not leading to **false**, and where edges leading to **false** is not represented. This means that the successor does not form an R-cover, in contrast to CDD's.
We denote an S-CDD node as a 3 tuple $\langle t_s, I, T_s \rangle$ where $t_s \in \mathcal{T}$ is the type and $I \in \mathcal{I}$ is the interval covered, and T_s is the the successor S-CDD node, that also can be interpreted as a S-CDD. $\mathcal{N}_s$ is the set of all S-CDD nodes, with typical element n_s, m_s.

S-CDD's has a global linear ordering as CDD's, refer to previous section for details. Again the bottom element the **true** node is considered the element with the largest type, and the handle to the S-CDD has the smallest type.

A S-CDD is a DAG consisting of a set of nodes $V_s \subseteq \mathcal{N}_s$ and two functions:

- $type : V_s \to \mathcal{T}_s$

– $succ : V_s \rightarrow (I, n_s)$, where $I \in \mathcal{I}$ and $n_s \in \mathcal{N}_s$ such that:

V_s has exactly one terminal nodes called **true**, where $type(\textbf{true}) = \perp$

all other nodes $n_s = \langle t_s, I, T_s \rangle \in V_s$ are inner nodes, which have attributed a type $type(n_s) = t_s$, and a single successor: $succ(\langle t_s, I, T_s \rangle) = T_s$. $first(T_s) = m_s$ is the node in T_s with the smallest type, that is the handle.

We define $child(n_s)$, $n_s \in \mathcal{N}_s$ to be the successor node of n_s in the S-CDD. All inner nodes has a unique successor node.

We define $parent(n_s)$ to be the parent of node n_s. This method makes sense only for S-CDD's as these has a unique parent, which may not be the case for ordinary CDD's.

We define $I_{s-cdd}(n_s)$ to be the interval leaving node n_s.

$n_s \xrightarrow{I} m_s$ is short hand for $(I, m_s) = succ(n_s)$.
For each inner node the following must hold:

– the successor does **not** form an $\mathbb{R}$-cover, otherwise the node should be omitted.

– the S-CDD is linearly ordered: for all m_s, whenever $n_s \xrightarrow{I} m_s$ then $type(n_s) \sqsubset type(m_s)$.

– an S-CDD node has a unique parent and child node.

In the following part of the report, S-CDD is also referred to as single stringed CDD's.

3.1.2 Difference Bounded Matrices

The following subsection describes the Difference Bounded Matrices(DBM) data structure used in the current distributed version of Uppaal. DBM's can be used for the same purpose as CDD's, that is describe constrains of the form $X_i - X_j \leq n$. The DBM data structure is best seen as a directed graph, with vertices being the variables $X_0, X_1, \ldots, X_n$. For each constraint $X_i - X_j \leq n$ a directed edge goes from X_i to X_j with weight n. An advantage with DBM's compared to CDD's is that DBM's has a normal form which simplifies *inclusion tests*, to a test of syntactic inclusion instead of a test for semantic inclusion. The graph spanned by the constraints is described by the adjacency-matrix representation in reduced form.

Only a single *Zone* Z can be described by a DBM, so if two *Zones* Z_1 and Z_2 are explored where $Z_1 \not\subseteq Z_2 \wedge Z_2 \not\subseteq Z_1$, then two DBM's has to be constructed and stored. E.g. to represent the explored state space of figure 3.2(a) on page 18, the two matrices in table 3.1.2 has to be stored:

	X_0	X_1	X_2	X_0	X_1	X_2
X_0	0	6	6	0	12	6
X_1	-2	0		-8	0	
X_2	-2		0	-2		0

Table 3.1: DBM's for storing the state space: $(-2 \leq X_2 - X_0 \leq 6 \wedge\ -2 \leq X_1 - X_0 \leq 6 \wedge\ -\infty < X_2 - X_1 < \infty) \cup (-2 \leq X_2 - X_0 \leq 6 \wedge\ -8 \leq X_1 - X_0 \leq 12 \wedge\ -\infty < X_2 - X_1 < \infty)$
And after the representation of section 1.1.1: $(1 \leq X_2 - X_0 \leq 3 \wedge 1 \leq X_1 - X_0 \leq 3 \wedge -\infty < X_2 - X_1 < \infty) \cup (1 \leq X_2 - X_0 \leq 3 \wedge 4 \leq X_1 - X_0 \leq 6 \wedge\ -\infty < X_2 - X_1 < \infty)$

3.2 Operations

There are two main operations which is performed on the data structures used by Uppaal, these are:

Inclusion test Before the Uppaal engine explores a new state it checks whether it previously has been explored. This is done by storing all searched states in the *Passed* list, and before exploring a new state it checks whether the newly found state is included in the *Passed* list.

Union Whenever a new state has been explored it is inserted into the *Passed* list, this is done by a union of the *Passed* list and the newly explored state.

The following describes how *inclusion test* and *union* is performed in the DBM version of Uppaal and how they are performed using CDD's.

3.3 Current Uppaal

The current Uppaal version which is based on the DBM data structure, holds the *Passed* list as a simple linked list of DBM's. That is whenever a new state has been explored a union between the existing *Passed* list and the new DBM is performed, simply by adding the DBM to the linked list.
To perform an inclusion test using the DBM data structure, each DBM in the linked list is tested for being a superset or equal to the newly found state. An algorithm for the inclusion test using the DBM data structure is shown in figure: 3.3

$bool\ \ dbm\_inclusion(d : newstate; L :\ PassedLIST)$
begin
return $\bigvee\{d \subseteq d' | d' \in L\}$
end

Figure 3.3: Inclusion test using the DBM data structure. That $d' \in L$, denoted that the state d and d' has to have the same discrete state.

This algorithm does not recognize a *Zone* if this *Zone* is a subset of the union of two or more *Zones*, as shown in figure 3.4 where the marked zone is a subset of the union of two other zones.

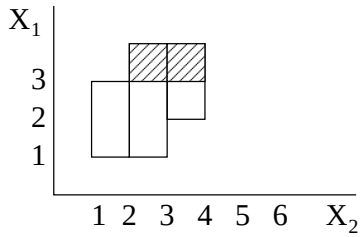


Figure 3.4: If the three vertical *Zones* has been searched, the horizontal marked *Zone* would not be accepted as searched.

3.3.1 Using CDD's with Uppaal

Both inclusion tests and union with new *Zones* is somewhat more complicated using the CDD data structure, but it offers better inclusion tests as CDD recognize that a *Zone* has been explored if the union of two *Zones* makes a superset. The inclusion test is performed from the following formula:
$T_s \subseteq T$ **iff** $T_s \cap T = T_s$
where T_s is the new state as a S-CDD, and T is the *Passed* list represented as a CDD.

When a new state has been explored the *Zone* explored has to be added to the explored state space set the *Passed* list. Actually the current CDD implementation of Uppaal uses a different approach, as it checks for $T_s \subseteq T$ **iff** $T_s \cap \neg T = \emptyset$.

3.4 Mapping CDD's to set formulas

This section provides another view on the CDD data representation, which is used to prove properties of the CDD data structures in chapter 5. CDD's are used to describe a *federation* in the multi-dimensional space spanned by the clocks C. Therefore a CDD can be interpreted as a set of intervals on the different coordinates.
Let p_t represent the number p of the type $t \in \mathcal{T}$, that is the set formula $[2_x; 4_x]$ represent the interval $[2; 4]$ on the X coordinate. Then the area in figure 3.1 on page 17, may be represented as the set formula:

$$(([1_y; 4_y] \cap [1_x; 4_x]) \cup ([2_y; 6_y] \cap [2_x; 6_x])) \cap [-\infty_{x-y}; 0_{x-y}]$$

This set formula can be generated from a CDD using the following recursive formula:

$CDDtoSET(cdd \in CDD)$
begin
  if $cdd =$ **true return** $(\mathbb{R}^+)^n$
  if $cdd =$ **false return** $\emptyset$
  else return $\bigcup_{(I,m)\in succ(n)}(I \cap CDDtoSET(m))$
end

3.5 Mapping DBM's to CDD's

All communication between the CDD data structure and Uppaal is done through the DBM data structure. Whenever an *inclusion test* is performed (figure 1.3 line 5) it is checked whether the shortest-path closure is contained in the $PASSED$ list. Whenever a new state should be inserted into the CDD, the shortest-path reduced DBM is inserted. But before these operations is conducted, it is preferable to know how to convert DBM's into corresponding CDD's, as operating on the same data structure is simpler. Note that an efficient implementation should not perform this conversion, it is merely a matter to use in semantics of algorithms proposed later on.

The structure of DBM's only allow two constraints between any two clocks: $X_i - X_j \leq n$ and $X_j - X_i \leq m$, where $X_i, X_j \in C$ and $n, m \in \mathbb{N}$. Therefore, for each type (node in a CDD) t only

a single interval may leave not leading to the **false** node, and as the CDD only starts in a single point the CDD resulting from converting a DBM into a CDD becomes a CDD where each node has a single unique successor - prior defined as a S-CDD. The result of this is that whenever a DBM is converted into a CDD, the CDD is an S-CDD. See figure 3.5.

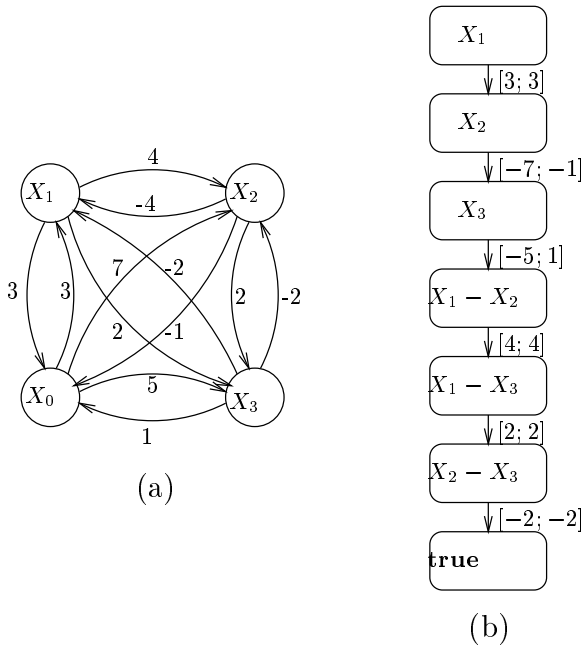


Figure 3.5: Any DBM converted into a CDD, becomes a S-CDD. The DBM of figure (a) is converted into the S-CDD of figure (b).

This chapter describes the design of the algorithms, both the sequential and the distributed ones. After the description of the algorithms the design of the CDD nodes is described.

4.1 Non Distributed Algorithms

In this section we describe the algorithms that is used on ordinary CDD data structures on a single processor architecture. These algorithms is our starting point when we distribute the data structures. The algorithms described is *inclusion test* and *union* (between an S-CDD and a CDD). This is followed by a description of how CDD's are reduced to ensure as much sharing as possible, but first we consider how intervals are merged during the union operation.

4.1.1 Merging intervals

Whenever two nodes with the same type has to be unioned, their intervals has to be merged. If CDD node n_i has intervals $I_{i_1} \dots I_{i_k}$, and CDD node n_j has intervals $I_{j_1} \dots I_{j_l}$, then the merging of these intervals is the smallest number of intervals $I_{r_1} \dots I_{r_m}$, such that $\forall I' \in \{I_{r_1} \dots I_{r_m}\}, \exists I_i \in \{I_{i_1} \dots I_{i_k}\}, I_j \in \{I_{j_1} \dots I_{j_l}\} | I_i \cap I_j = I'$. And the interval partition formed by $I_{r_1} \dots I_{r_m}$ forms a *reduced interval partition*.

When the intervals are merged the new node is created by letting the new CDD node have the same type as the input nodes. One outgoing edges exists for each interval in the merged *interval partition*, and the edge with interval I_r points to the union of the nodes n_{I_i} and n_{I_j}, where $n_{I_i} = child(n_i, I_i)$ is the node referred to by node n_i interval I_i, and $n_j = child(n_j, I_j)$ is the node referred to by node n_j, interval I_j, and $I_i \cap I_j = I_r$.

As all unions in this project is only done between S-CDD's and CDD's, and it is known that S-CDD only have one interval leaving each node (not leading to false), the algorithm in figure 4.1 show how the intervals of a CDD node $m \in \mathcal{N}$, and S-CDD node $n_s \in \mathcal{N}_s$ is performed.

The algorithm works as follows (figure 4.1(a) might help):
Line `3` assign I_s to the interval leaving the S-CDD node n_s. Line `4` creates a new empty set of intervals, which will become the resulting *reduced interval partition*. Line `6` to `16` iterates over all intervals leaving the CDD node m, and for each interval the following is done:
Line `6-7` handles if the two intervals I' and I_s does not intersect, then the interval I' is simply added to the resulting set $I(merge)$, is is the case for the interval (a, b) of $I(m)$ in figure 4.1(a), and result in the interval (a', b') in $I(merge)$. Line `8-9` handles the same case, only if the interval I_s is a subset of I', this is the case for the interval (e, f) of $I(m)$ in figure 4.1(a), and result in the interval (e', f').
Line `11-16` handles the case where I_s and I' isn't $\emptyset$ nor I_s. Such an intersection can be done in 5 different ways as shown in figure 4.1(b), by the I_s intervals: I_{s_1}, I_{s_2}, I_{s_3}, I_{s_4} and I_{s_5}. Line `11` get the bounds from both intervals, which is used in the **if** conditions in the following 5 lines. The following describes how each of the five intersections is handled:
$I_{s_1} \cup I'$: This intersection falls into the **if** conditions on line `14` and `16`, and add (a, c) and (c, f) to $I(merge)$.

```
 1: Reduced_Interval_Partition merge_intervals(n_s ∈ N_s, m ∈ N)
 2: begin
 3:       I_s = I_{s-cdd}(n_s)
 4:       I(merge) = ∅
 5:       foreach I' ∈ I(m)
 6:       if I' ∩ I_s = ∅ then
 7:             I(merge) = I(merge) ∪ I'
 8:       else if I' ∩ I_s = I' then
 9:             I(merge) = I(merge) ∪ I'
10:       else
11:             (a, b) = I', (c, d) = I_s
12:             if a < d ∧ c < a then I(merge) = I(merge) ∪ (a, d)
13:             if a < c         then I(merge) = I(merge) ∪ (a, c)
14:             if a ≤ c ∧ d ≤ b then I(merge) = I(merge) ∪ (c, d)
15:             if a < c ∧ b < d then I(merge) = I(merge) ∪ (c, b)
16:             if d < b         then I(merge) = I(merge) ∪ (d, b)
17:       return I(merge)
18: end
```

Table 4.1: Algorithm for merging intervals.

$\mathbf{I_{s_2} \cup I'}$: This intersection falls into the **if** conditions on line `13` and `14`, and add (a, d) and (d, f) to $I(merge)$.
$\mathbf{I_{s_3} \cup I'}$: This intersection falls into the **if** conditions on line `12` and `16`, and add (a, b) and (b, f) to $I(merge)$.
$\mathbf{I_{s_4} \cup I'}$: This intersection falls into the **if** conditions on line `13`, `14` and `16`, and add (a, c), (c, d) and (d, f) to $I(merge)$.
$\mathbf{I_{s_5} \cup I'}$: This intersection falls into the **if** conditions on line `13` and `15`, and add (a, e) and (e, f) to $I(merge)$.

E.g. the merging of I_s and $I(m)$ of figure 4.1(a) is shown as $I(merge)$ of figure 4.1(a).

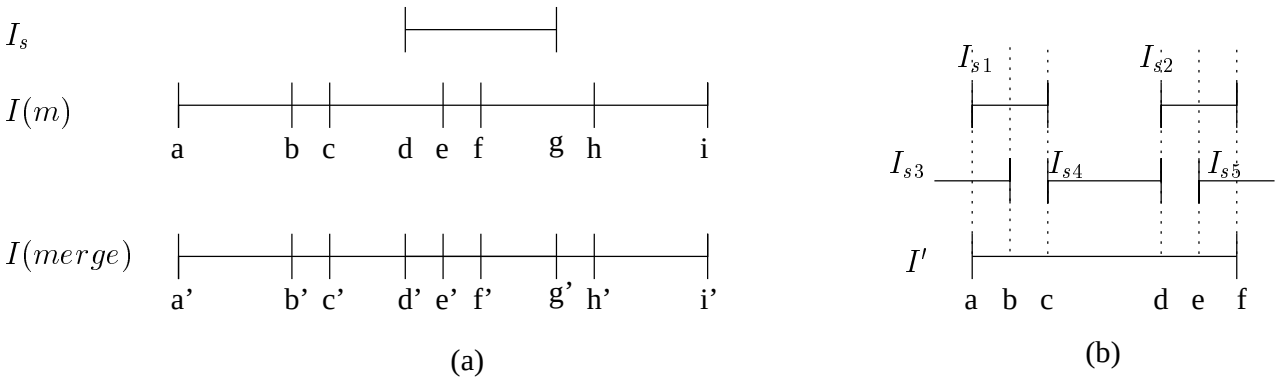


Figure 4.1: (a) shows an example to visualize why intervals must be merged and not just union the two sets of intervals.
(b) Visualize five of the seven different ways two intervals can intersect one another.

4.1.2 Union

The most basic operation of the CDD data structure is the creation of the CDD data structure. The basic operation used in construction and in union is a function called $makenode(t \in \mathcal{T}, [I_1, T_1] \ldots [I_n, T_n])$ which for a given type, and successor set either return an existing node which hold the same properties, and if such node does not exist create and return a node with the described properties. The operation is described by the algorithm in figure 4.2. The operation is important for keeping reducedness of the constructed CDD.
Note that the *makenode* relies on the *reduce* method. The *reduce* method is used to reduce S, S being the successor set$[I_1, T_1] \ldots [I_n, T_n]$, e.g. it has maximum sharing, no trivial edges and all intervals are maximal - that is *reduce* ensures that S is a reduced CDD, according to the definition of reducedness in section 3.1.1

```
 1: CDD_Node makenode(t ∈ T, [I_1,T_1] ... [I_n,T_n])
 2: begin
 3:       // Denote [I_1,T_1] ... [I_n,T_n] by S, the successor (succ) of the input description
 4:      reduce(S)
 5:      if (∃n ∈ V | type(n) = t ∧ succ(n) = S) return n
 6:       else V := V ∪ {n} // where n is a new node
 7:            type := type ∪ {n ↦ t};
 8:            succ = succ ∪ {n ↦ S}
 9:            return n
10: end
```

Figure 4.2: Algorithm for the *makenode* operation.

The algorithm for union is depicted in figure 4.3.

```
 1: CDD_Node union(n_s ∈ N_s, m ∈ N)
 2: begin
 3:      if n_s = true ∨ m = true then return true
 4:      else if m = false then return n_s
 5:      else if type(n_s) = type(m) then
 6:           I(new) = merge_intervals(n_s, m) // new is a new CDD node
 7:           return makenode(type(m), {(I, m'') |
 8:                I ∈ I(new)
 9:                I ∩ I_s = ∅ ⇒ m'' = child(m, I') | I' ∈ I(m) ∧ I ⊆ I'
10:                I ∩ I_s ≠ ∅ ⇒ m'' = union(child(n_s), child(m, I')) | I' ∈ I(m) ∧ I ⊆ I'}
11:     else if type(n_s) ⊏ type(m) then (a, b) = I_s
12:          return makenode(type(n_s), {((−∞, a'), m), (I_s, union(child(n_s), m)), ((b', ∞), m)}
13:     else if type(m) ⊏ type(n_s) then
14:          return makenode(type(m), {(I_i, union(n_s, m'))|m --I_i--> m'})
15:     endif
16: end
```

Figure 4.3: Algorithm for the *union* operation.

Line 3-4 handles the trivial cases where either of the CDD/S-CDD consists only of the **true** or **false** node.

Line 5-10 handles the case where two nodes of the same type has to be unioned, this is done by merging their intervals. An example is given in figure 4.4.

[Line 6] merge the interval partition of the CDD node, and the interval of the S-CDD node, as described in the previous section.

[Line 7] makes a call to *makenode*, with an request to make a new CDD node, with the interval partition returned in line 6.

[Line 8] Iterates over the interval in the interval partition $I(new)$.

[Line 9] If I intersected with I_s is the empty set, then the child of the interval I is unchanged.

[Line 10] If I does intersect with I_s, then the child of the interval I is the union of the prior node referred by I' unioned with the child of the S-CDD node.

Line 11-12 handles the case where the type of the S-CDD is smaller then the type of the CDD. This is handles by creating a new CDD node with the same type is the S-CDD with three intervals. One going from $-\infty$ to a' which must refer to the CDD node given as argument. One successor with the same interval as the S-CDD referring to the union of the child of the S-CDD and the CDD nodes given as argument. The last successor being $(b';\infty)$ referring to the CDD nodes given as argument. a' denotes the opposite bound than a, that is if a is $[m$, then a' denotes $m[$, the same is the case for b', which is opposite to b. An example is given in figure 4.5.

Line 13-14 Handles the case where the type of the CDD is smaller than the type of the S-CDD. This is handles by substituting all the children of the CDD node with the union of the child and the child of the S-CDD node. An example is given in figure 4.6.

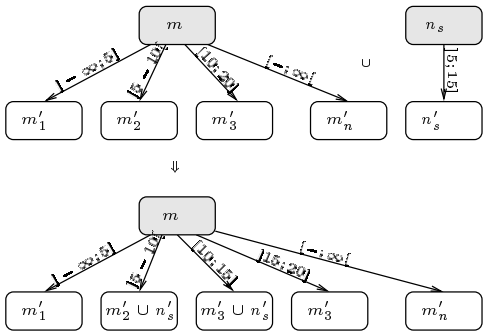


Figure 4.4: Show how union is performed on two nodes having the same type. The nodes to be unioned is marked, as well as the created node.

The way the CDD data structure is build bottom up on a single processor architecture, using the *makenode*/*reduce*, ensures that the CDD is always reduced. This can only be done on a single processor architecture as a single call stack exist here. When using distributed computing several call stack exists one on each computer node, and other actions has to be issued to ensure reducedness. We describe these action later, when the distribution scheme is described.

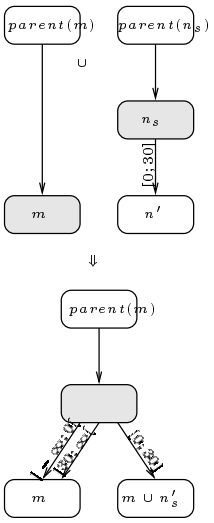


Figure 4.5: Show how union is performed when the type of the S-CDD is smaller then the type of the CDD. The nodes to be unioned is marked, as well as the created node.

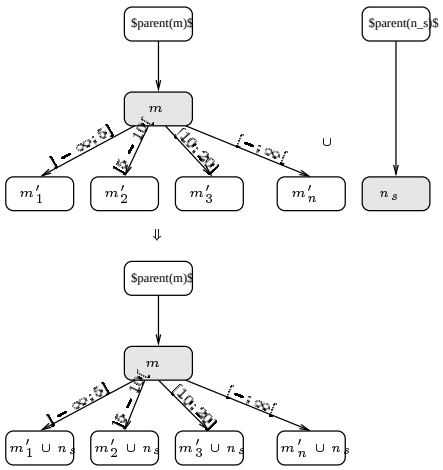


Figure 4.6: Show how union is performed when the type of the CDD is smaller then the type of the S-CDD. The nodes to be unioned is marked, as well as the created node.

Hash Table

The *makenode*() function which has to search weather an existing node with the same *type* and *successors* already exists before creating a new one, for this purpose a hash table is build so the expression $(\exists n \in V | type(n) = t \wedge succ(n) = S)$, can be performed in $O(1)$ time. The hash table is build so that the hash function accepts a *type* and a list of interval/successors, and return a list of nodes in V satisfying that hash function.

4.1.3 Inclusion Test

The next operation we describe is the *inclusion test*.

```
bool inclusion(passed : CDD_Node check : S − CDD_Node)
 1:    if passed = true return true
 2:    if passed = false return false
 3:    if type(passed) ⊏ type(check) return ⋀{inclution(T_i, check)|passed --I_i--> T_i}
 4:    if type(check) ⊏ type(passed) return inclusion(passed, child(check))
 5:    if type(passed) = type(check) return
 6:        ⋀{inclusion(passed', child(check))|
 7:            ∃passed' ∈ 𝒩 ∧ ∃I_passed ∈ ℐ.passed --I_passed--> passed'. I_passed ∩ I_check ≠ ∅}
end
```

Figure 4.7: Algorithm for the *inclusion* operation.

This algorithm shown in figure 4.7, works by simulating the check S-CDD $\cap$ CDD = S-CDD, it is not intended to calculate the actual output of this intersection, this would be of no use as there are no normal forms for CDD's. Instead we simulate all traces from the handle that has overlapping intervals with the interval of the S-CDD, to see if we reach the **true** node in all traces.

line 1,2: These lines covers the trivial cases where the *Passed* list is either the **false** node or the **true** node.

line 3: If the type of the S-CDD node is larger than type of the *passed* node, this means that the S-CDD implicitly contains a node with same type as the *passed* node, and this node covers $\mathbb{R}$ as its interval. Therefore all successors of the CDD node *passed*, need be examined.

line 4: If the type of the *check* node is smaller than the type of the passed node, then the CDD implicitly contains a node with the same type as *check* forming an $\mathbb{R}$ cover, and clearly the *check* node is included in this node. Therefore we can traverse further down the S-CDD making a recursive call with the *passed* and *child*(*check*). Remember that the *child* function for the S-CDD returns the unique successor of the node given as argument.

line 5,6,7: Here the types of the two nodes are identical, thus we recursively traverse the edges where there are overlapping intervals with the interval from the S-CDD I_{check}, to see if the entire interval I_{check} is covered, that is leading to the **true** node.

4.2 Distributing the Data Structure

In this section we describe how we distribute the CDD data structure. Our primary goal is to allow Uppaal to verify larger models, by distributing the CDD data structure. Another less significant goal is to minimize the communication overhead to a minimum. As described in the purpose, the purpose of this project is two fold. First we wish to investigate how much memory can be saved by allowing sharing between all discrete states, and secondly how large is the time/communication overhead introduced. Before introducing the distributed algorithms we consider what platform we aim our distribution at.

4.2.1 Distribution Platform

We consider two main targets for distribution, either distributed shared memory or message passing. In message passing the communication between computer nodes is via messages explicitly send to a specific node, a well known standard MPI defines syntax and semantics for the different calls, thus libraries exits for varying architectures. In distributed shared memory the entire memory area is transparently accessible by all processes as it where local memory. Shared memory libraries include Parallel Virtual Machine and IVY among others. We need to synchronize the *union* and *inclusion* operations, as we cannot allow an *inclusion* operation to "overtake" a *union* operation, as it might see the first part of the *union* but not the last which might lead to inconsistency, if the *inclusion test* tries to call a node which is not constructed yet, as it will be constructed by the *union* it just "overtook". Thus we need to either synchronize on some level, either being each node or possible at each type level. This actually means that we synchronize the entire structure, by introducing a large pipeline on all the computer nodes. Each computer node must either perform *inclusion*/*union* operation or explore states in Uppaal. We can choose to either make this synchronization explicitly with mutex locks using the distributed shared memory model, or we can do this implicitly by using message passing as MPI guarantees message ordering. We have chosen the later for several reasons: First efficiency, as we expect it to be more efficient to perform the pipeline implicitly than explicitly. Secondly the existing distributed version of Uppaal uses MPI thus minimizing the hardware requirements of the current Uppaal users. There are of course several advantages of using shared memory, one of them being that it is almost transparent to the programmer, thus easing the programming, but we consider the previous mentioned arguments to out weight these advantages.

4.2.2 Distributing Among Nodes

The idea for distributing the CDD data structure is by distributing the CDD nodes among the computer nodes and letting them communicate whenever an operation (*inclusion test*/ *union*) is to be performed.
The first issue to consider is how to distribute the CDD nodes. As we are not aware of other work distributing the CDD data structures, or the very alike IDD (Interval decision Diagrams [14]) data structures, it was investigated what work that previously has been done for distributing the BDD data structure, and the following distributing ideas were found:

Horizontal The first approach encountered, were a horizontal distributing. In this approach all CDD nodes with the same type are guaranteed to stay at the same machine node. The

partitioning is then horizontal as the name indicates. For an illustration of this idea please refer to figure: 4.8(a)

Vertical This approach distributes the CDD vertically, by keeping a number of handles on each computer node, in the same way as the distributed version of Uppaal. Sharing between all locations becomes difficult, as every time we create a new CDD node all other computer nodes has to be asked whether they already has such a node. *Inclusion/union* becomes difficult because a trace from the handle to the **true** node may jump from computer node to computer node as many times as there are CDD nodes in this trace, which may be exponential to the number of types. Figure 4.8(d) shows the idea of this approach.

Groups Another approach is to partitioning the CDD arbitrary, both horizontal and vertically. But as can be seen from figure 4.8(b), this may lead to communication between vertically distributed CDD nodes. As were the case for the vertical distribution approach.

Distributed The last idea is borrowed from [12]. This approach takes advantage from the fact that the variable ordering of the types is very important for how much sharing that can be accomplished. The idea is that each node hold a CDD, each with different variable orderings, then whenever an operation should be carried out, the explored *Zone* is send to all computer nodes. During the *union* operation the computer node that accomplish maximal sharing (need fewest new nodes to represent the new *federation*) stores the new zone, and all other nodes discharge the new zone.

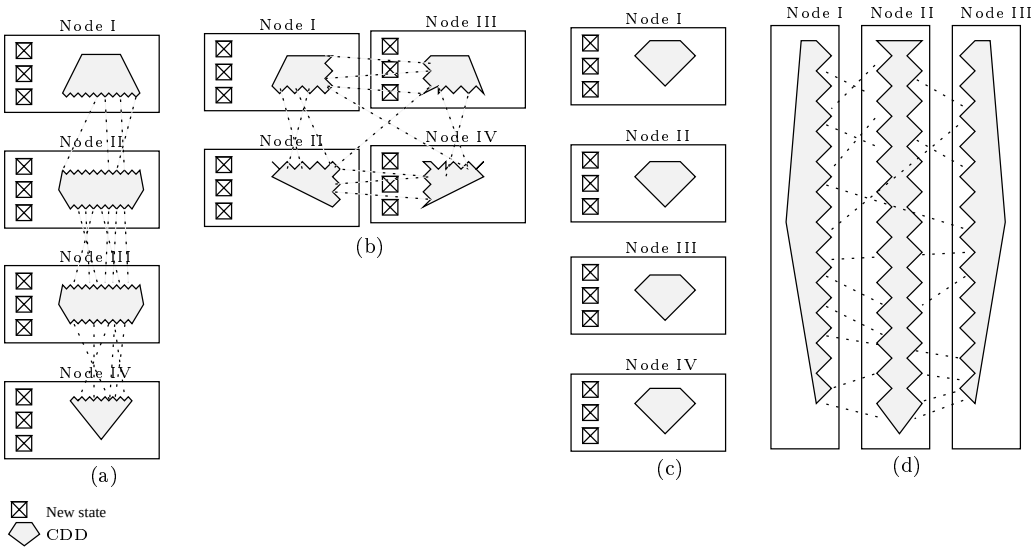


Figure 4.8: Different approaches for distributing the CDD nodes among several distributed computer nodes. (a) show a horizontal representation, (b) show an approach based on groupings, (c) shows an approach based on several distributed CDD's, and (d) shows the vertical distribution approach

The following describes the advantages and disadvantages of the listed possibilities of distributing the CDD data structure.

Horizontal Using the approach of horizontal distribution, the number of messages for both operation (*inclusion test* and *union*) is the number of nodes participating in the operation except one. Besides that the partitioning of CDD nodes among computer nodes is simple, and a simple interface may be kept.

Groups Unlike the approach of horizontal distribution, the approach using groups does not have a known number of messages for each operation. If the partitioning is good, and the right operation is performed the number of messages is less than for the horizontal approach, but if unlucky the number of messages might increase exponential. Although the approach has some advantages over the horizontal approach such as: The approach might be easier to apply to a distributed shared memory environment, as the node creating/modifying a node creates the nodes locally. Also the approach might result in a better memory distribution, as any number of CDD nodes might be moved from one computer node to another - thereby releasing a computer node with less memory. Using the horizontal approach, only one layer (all nodes with the same type) may be transfered at runtime.

Vertical This approach suffers from the same as the previous approach, though in an even more extreme sense.

Distributed The approach using several CDD's to store the *Passed* list, is somewhat alike the existing distributed Uppaal version. But instead of storing the newly explored state locally, the state is stored at the node where it takes up least space. [12] shows promising results using this approach for BDD's. Another advantage of this approach is the ease of implementation, as when a single machine version has been implemented, the only action to implement the distributed version, is to communicate new zones and synchronize on the size added.
Disadvantages of this approach is that it cannot in general handle union of *Zones* (*federations*). If the *federation* $F_1 = Z_1 \cup Z_2$, where Z_1 and Z_2 are *Zones* located on two different computer nodes, then it is not possible to recognize that any subset of F_1 has been searched. Another disadvantage is that much time is wasted during the many unions on all nodes, most of which is simply discharged.

The approach chosen for this project is the horizontal distribution as it offers the best scalability in that the number of messages needed has an upper limit.

4.3 Communication

To be able to perform the operations distributed, computer nodes next to one another need to hold some reference to the nodes on the next computer node. The following describes this, where the upper computer node is called *client* and the lower machine node is called *server*. Each computer node holds an array, which is indexed equally on each computer node.
Informations hold on the client side is:

- **Reference count:** The client need to know how many references goes into each CDD node at the server node, so that the array index can be deleted when the reference count reaches zero. The reference count is increased whenever a CDD node points to the corresponding CDD node at the server. The reference count only changes during *reduction* of the distributed data structure, which is done in another way than for the single node CDD data structure. The mechanism for reducing of the distributed CDD data structure is described later in section 4.4.3

On the server side the array contains:

– **Pointer:** The server side is an array of pointers to CDD's node which the array entry corresponds to.

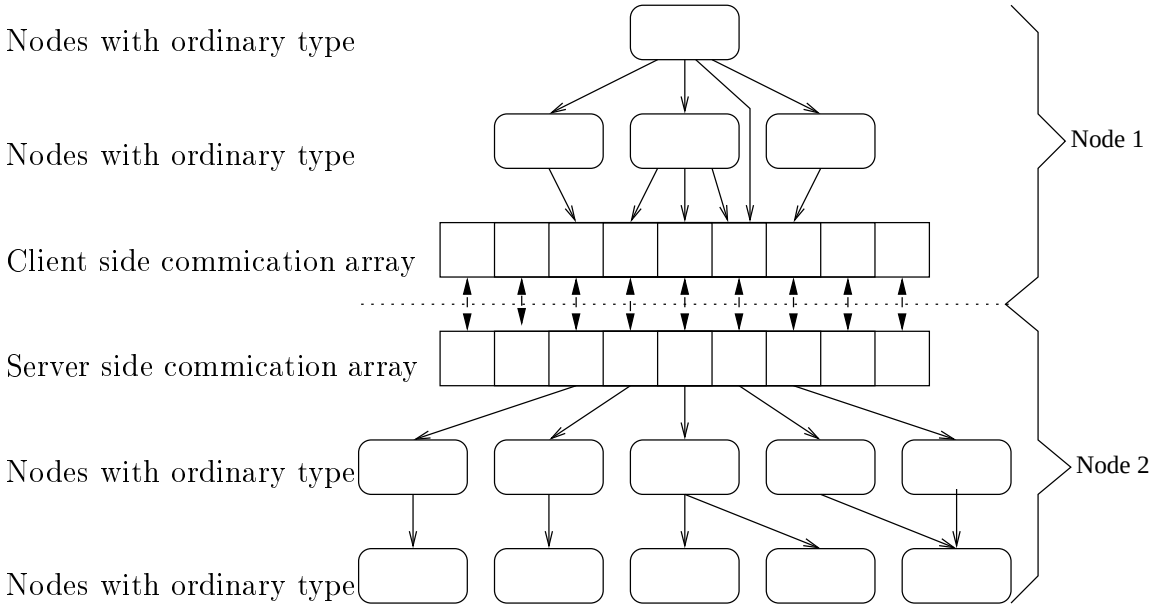


Figure 4.9: Show how the communication between machine nodes is done.

The communication layer, can be seen as an extra layer of CDD nodes, each entry in the communication layer is then seen as a CDD node, with a special type t_{comm} - not in the type set of the timed automata. The *succ*() function only holds a single outgoing interval namely the interval $]-\infty;\infty[$, thereby it can be seen that each entry in the communication layer may have several in going edges, but only a single outgoing edge. Thus each node referenced on the next computer node can be uniquely identified on both computer nodes by the array index.

4.4 Operations

In the following subsections we describe how the *union* and *inclusion test* is performed in the distributed CDD data structure, using the communication interface previously described.

4.4.1 Distributed Inclusion Test

The algorithm for making the *inclusion test* in the single processor CDD is described in section 4.1.3. The operation of the distributed version is similar, except on one point:
The single processor version uses a depth first approach, if the same approach is used in the distributed approach, many messages must be send between machine nodes. Therefore the approach taken is somewhat between depth first and breath first, at each machine node the depth first approach is used until the communication layer is reached. Information about which

nodes at the next computer node, that is part of the *inclusion test*, is then transfered (unless the *inclusion test* has already failed!), and only when the request fails on a node, or the request reaches the bottom machine node (holding the **true** CDD node) the final answer is found.
That is, each machine node receives a number of references for which the *inclusion test* is to be performed. Then a recursive depth first algorithm is run on these nodes, each time reaching the communication layer appending the found entry in the communication array to a list. If at any instance the *inclusion test* fails `false` is returned. If all incoming requests succeeds the list containing array indexes is send to the next node. If all requests succeeds at the bottom node `true` is returned.
The *inclusion test* is always initiated on the computer node holding the handle.
Let E denote the list of CDD nodes communicated from the prior (client) layer, and let E' be the list that should be passed on to the next computer node. Then the algorithm in figure 4.10 illustrates how the distributed *inclusion test* is performed:

```
 1: distributed_inclusion(T_s : S − CDD, E : [CDD])
 2: begin
 3:       E' = ∅
 4:       foreach T_i ∈ E do
 5:       begin
 6:             if T_i ∈ Communicationlayer then
 7:                   E' = E' ∪ T_i
 8:             b = recursive_inclusion(T_i, T_s)
 9:             if b = false return false
10:       end
11:       if not bottom layer pass E' to next layer.
12:       else return true
13: end
```

```
 1: bool recursive_inclusion(T_i : CDD, T_s : S − CDD)
 2: begin
 3:       if T_i = true return true
 4:       if T_i = false return false
 5:       if type(T_i) ⊏ type(T_s) return ⋀{recursive_inclution(T_j, T_s)|T_i --I_j--> T_j}
 6:       if type(T_s) ⊏ type(T_i) return inclusion(T_i, succ(T_s))
 7:       if type(T_i) = type(T_s) then
 8:             ∀T_j ∈ V.∃I_j ∈ I|T_i --I_j--> T_j ∧ I_j ∩ I_s ≠ ∅
 9:                   if T_j ∈ communication layer then
10:                         E' = E' ∪ T_j
11:                         return true
12:                   else return recursive_inclusion(T_j, succ(T_s))
13: end
```

Figure 4.10: Algorithm for distributed *inclusion test*.

The recursive part of the distributed *inclusion test* as shown in the lower box of figure 4.10 works in the same depth first fashion as the non-distributed *inclusion test* does, the only difference is the lines `9-11`, where it is tested whether the examined CDD node, is a communication node, and if this is the case, it is added to the *inclusion message* and `true` is returned. `True` is returned

as this node cannot yet decide whether the S-CDD is included or not, and as only the bottom node may give the final **true** reply this does not result in semantic faults.

Whenever a distribution requests is send to a node, the *distributed_inclusion* methods, which is shown in the upper box of figure 4.10. It receives a S-CDD, and a list of communications indices, it makes a recursive call to *recursice_inclusion* for each element in the communication array, if any of the *recursive_inclusion* fails it immediately known that the S-CDD cannot be included in the CDD, and immediately returns `false`. If all recursive calls succeed, all nodes except the bottom node send the *inclusion* request to the next node, and the bottom node simply returns `true`, as the S-CDD must be included in the CDD.

4.4.2 Distributed Union

In this subsection we describe how the distributed *union* operation is performed. The distributed *union* operation is somewhat different from the single processor version. The single processor version uses the *makenode* operation whenever a new node is needed to ensure that the CDD is reduced during the *union*, furthermore the single processor version is performed in a recursive bottom up manner, so reduction can be performed at creation time (reduction can only be performed bottom up).

The same mechanism can hardly be used in a distributed union, as the number of messages send between two machine nodes is not known and may be really large, as the recursion at least have to traverse to the bottom **true** node, and during the recursion termination have to return to the top node again. As shown in figure 4.3 on page 27, the single processor union operation call is depth first - that is, if at the top node, a merge is made which splits in to more recursive calls, then the number of messages increases exponentially, therefore another distributed *union* operation is designed.

In principle the distributed *union* works the same, but when *union* is called with a *type* in the communication layer, an entry in the communication array is allocated, and a pointer to this entry is returned. Furthermore this entry is added to a message, which is send to the server node (lower machine node), when the recursion on the current node terminated for all traces. When the next node receives this message it continues the construction, and gives request to the next node etc.
During the construction of CDD nodes at each machine node, the nodes added/modified is saved in a special data structure (called *call stack* from this point), this *call stack* is later used for reducing the CDD data structure - the *reduction* operation is described in section 4.4.3 later.

One exception to the previous rules, is the bottom machine node (holding the **true** CDD node). The union on this computer node is performed like the single processor union operation, and makes reduction implicitly as we build the CDD bottom up here.

Communication

Whenever a machine node has performed its union, it has to send a message to the lower level machine node. This message contains the following information:

ID Each *union* request is given a unique id. The *union* operation uses this to uniquely identify

which CDD-nodes has been modified/added by this *union* operation - this information is used whenever a distributed *reduction* is performed.

S-CDD Naturally the S-CDD which should be unioned need to be transfered. But S-CDD nodes with type smaller than the least type on the destination computer node can be omitted

Array Entries Finally the entries in the communication array which need to be unioned with the next node, is added to the message. The next node need this to union each CDD-node corresponding to the entries in the communication array with the S-CDD in the message.

Algorithm

The distributed algorithm for the *union* operation for a single node is shown in figure 4.11

```
 1: CDD_Node distributed_union(n_s ∈ N_s, A ⊆ V)
 2: begin
 3:     if n_s = true then foreach n ∈ A do n = true
 4:     else
 5:         foreach n ∈ A | n ≠ true
 6:             if n = false then n = SCDDtoCDD(n_s)
 7:             else rec_distributed_union(n_s, n)
 8:         SEND COMM
 9:     endif
10: end
11: CDD_Node rec_distributed_union(n_s ∈ N_s, m ∈ N)
12: begin
13:     if n_s = true ∨ m = true then return true
14:     else if m = false then return SCDDtoCDD(n_s)
15:     else if type(m) = t_comm return new_entry(COMM) + add to communication layer.
16:     else if type(n_s) = type(m) then
17:         I(new) = merge_intervals(n_s, m)
18:         return makenode(type(m), {(I, m'') |
19:             I ∈ I(new)
20:             I ⊈ I_s ⇒ m'' = child(m, I') | I' ∈ I(m) ∧ I ⊆ I'
21:             I ⊆ I_s ⇒ m'' = union(child(n_s), child(m, I')) | I' ∈ I(m) ∧ I ⊆ I' }
22:     else if type(n_s) ⊏ type(m) then
23:         return makenode(type(n_s), {(I_s, union(child(n_s)), m)})
24:     else if type(m) ⊏ type(n_s) then
25:         return makenode(type(m), {(I_i, union(n_s, m'))|m --I_i--> m'})
26:     endif
27: end
```

Figure 4.11: Algorithm for the *distributed_union* operation. The SCDDtoCDD method converts the S-CDD given as argument to a CDD describing the same *Zone*, and return a handle to this CDD.

The distributed *union* operation differs from the non-distributed union, in that the distributed *union* works in stages, one stage for each node. That is the CDD is not build from bottom and up, but bottom up on each node. This again leads to a non reduced CDD is constructed, so

a special *reduction* operation must be applied to archive sharing, and thereby reduce memory usage.

A demonstration of the *union* algorithm is provided later, where the distributed *reduction* algorithm is also demonstrated.

Optimizations

In the following we describe some optimizations to the *union* operation. Also a necessary operation needed by the distributed *reduction* is described.

Call stack: As *reduction* can only be performed bottom up, due to the nature of the reducedness rules, the top node cannot reduce before the second node has cleaned up and so on. But the top node cannot reduce if is does not hold information about which node it has modified/added during the union. Therefore each node builds a call stack during the construction of its CDD partition. The call stack is not a call stack in common sense, it is only a list of added/modified nodes, which is checked during the *reduction* phase. The call stack for a single *union* operation is kept in memory until a *reduction* phase reaches the node, or until an explicit requests come to delete it. Each *call stack* is associated with a unique id.

Temporary Hash Tables: Is temporarily in the sense that it works on a 'per operation basis'. Consider performing union between the CDD shown in figure 4.12(a) with the S-CDD shown in figure 4.12(b). To perform $X_0 \cup Y_0$ the following operations must be performed: $X_{11} \cup Y_1$ and $X_{12} \cup Y_1$, and for these to be unioned the following two operations must be performed: $X_{22} \cup Y_2$ and $X_{22} \cup Y_2$, if nothing else is known X_{22} and Y_2 may be performed twice, resulting in twice the work on the union of the nodes below, therefore whenever a union is performed the nodes are stored in a hash table, then the second time a union is needed the already performed union may be reused, this may reduce the work.

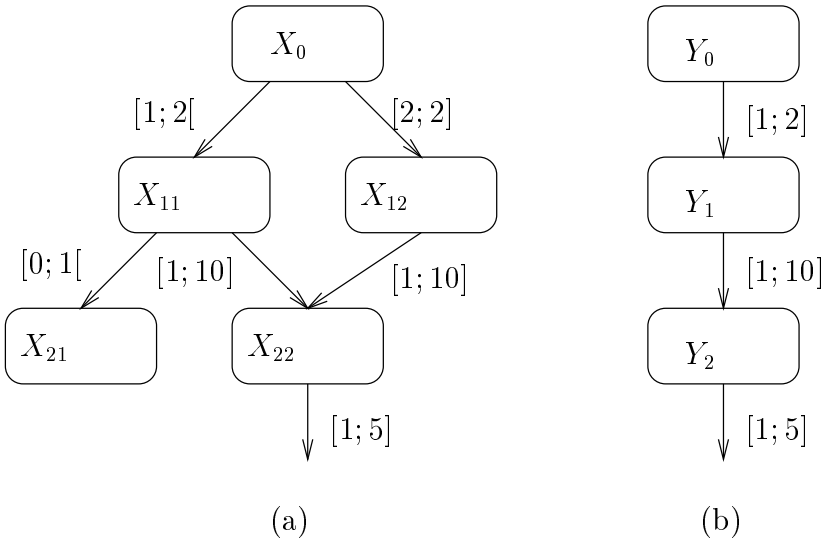


Figure 4.12: Shown a figure where the temporary hash table might save some time, by making an earlier reduction.

Distributed Hash Tables: Like the non distributed *union* operation the distributed *union* operation also depends on hash tables. Whereas the non distributed version uses hash tables to create the CDD, the hash table is only used by the last computer node to create reduced CDD nodes - but during the *reduction* operating the hash table also need to be used by the other computer nodes to ensure the reducedness properties of CDD's, so two isomorphic sub CDD's are not build. In the distributed CDD data structure implementation, each computer node holds a hash table containing only the CDD nodes located on this particular computer node. The entries in the hash table is the same as were the case for the non distributed hash table, as described on page 30.
This is only possible as the horizontal distribution approach were chosen.

4.4.3 Distributed Reduction

In the non distributed *union* operation, the call stack and the *hash table* ensures that the CDD data structure is always reduced during the construction. This is not possible in the distributed *union* operation, as the computer nodes does *not* have a common call stack, and cannot be constructed bottom up as is the case for the single processor operation. Therefore the reduction mechanism for the distributed *union* operation is a little different. First the *union* operation is performed as described in the previous section; when the request reaches the last machine node, and this node recognizes that a CDD node to create already exists, it reuses that node - just as in the single processor case. Furthermore it stores information about the reused CDD node. If two or more nodes in the communication layer becomes identical, this information is send to the prior computer node, which uses these informations to make a local reduction, and if this node recognizes that the prior computer node might make further reduction - reduction information is send to the previous computer node etc.

The description of the distributed *reduction* is split into three subsections, first the mechanism for the bottom machine node is described. Then reduction messages is described, and finally the *reduction* operation for all computer nodes other than the bottom computer node is described.

Bottom Computer Node

To describe how the distributed *reduction* is performed a little syntax is introduced. Let the computer nodes participating in the data structure be denoted by $M_1, M_2, \ldots, M_n$, and let the first type controlled by node M_i be denoted by t_{M_i}. Finally let the communication array between node M_i and M_{i+1} be denoted by $n_{comm_{M_i}}$.

When the bottom machine node M_n performs the *union* operation it does so for each entry in the array n_{comm_i} as instructed in the message from node M_{n-1}. During these *union* operations it consults a hash table of already constructed node (just as were the case in the single processor union), and whenever it discoverers that two nodes with type t_{M_n} are equal e.g. $(type(n_j) = type(n_i) = t_{M_n}) \wedge (succ(n_i) = succ(n_j))$, then the prior node M_{n-1} may redirect all pointers from n_i to n_j, or the other way around. To do that a message containing the information $\langle n_i = n_j \rangle$ is send to machine node M_{n-1}.

Communication

During the distributed *reduction* phase, messages is send from machine node M_i to machine node M_{i-1}, whenever node M_i finds two or more nodes in communication layer that are equal. The message send holds the following information:

ID: First an ID is added to the message, this ID is equal to the ID of the *union* operation that made the bottom node realize that two nodes in communication layer $n_{comm_{M_{i-1}}}$ were equal. This ID is used to access the call stack created during the *union* with id: **ID**, for further reduction.

Matching nodes: Besides the ID information of which CDD nodes are equal is send to the prior node.

Local Reduction

Let the *call stack* of a computer node be denoted by: $CS = \{n_{cs_1}, n_{cs_2}, \ldots, n_{cs_k}\} \subseteq V$. The local reduction algorithm for a computer node M_i, M_i not being the bottom computer node - is shown in figure 4.13, with 'cs' being a *call stack* and 'eq' being a message containing information of nodes that are equal, and thus used for further cleanup:

```
 1: void distributed_reduction(cs ∈ CS, eq ∈ EQ)
 2: begin
 3:     newEQ = ∅ // Message to send to M_{i-1}
 4:     foreach n ∈ cs do
 5:         if ∃(n --I--> m) ∈ succ(n) where (n_j, m) ∈ eq then
 6:             succ(n) = (succ(n) − {n --I--> m}) ∪ {n --I--> n_j}
 7:             V = V − m
 8:             if ∃n' ∈ V | type(n') = type(n) ∧ succ(n') = succ(n) ∧ n ≠ n'
 9:                 then eq = eq ∪ ⟨n', n⟩
10:             if type(n) = t_{comm_i} then
11:                 newEQ = newEQ ∪ ⟨n_p, n⟩ |
12:                     type(n) = type(n_p) ∧ succ(n_p) = n_j
13:     if newEQ ≠ ∅ then
14:         send newEQ to M_{i-1}
15: end
```

Figure 4.13: Algorithm for distributed *reduction*.

The **foreach** loop ranging from line 4 to line 12, iterates over all CDD nodes in the call stack, build during the *union* operation with the same *id*, as this *reduction* operation. During this iteration it is checked whether any of the nodes created during the *union* points to another node created during the *union*, which has a semantic equal node. Information of such two semantical equal nodes is given in the '*eq*' set.

Line 5 checks whether the current call stack node n, has a pointer pointing to the 'm' node in a element of the '*eq*' set. If it has this pointer is redirected to the semantic equivalent CDD node which in the pseudo code is denoted by 'n_j', this is done in line 6. Line 7 removes the node

m from the set of CDD nodes V. If the temporarily hash table, were not implemented, it is guaranteed that the CDD node 'm' if only referenced to by 'n', but when the temporary hash table is used several nodes from the call stack can reference a single node, but this problem is handled by the reference count of the node 'm'.
To see an example on the functionality, please refer to figure: 4.14, where the set $\langle \mathbf{2}, \mathbf{1} \rangle$ is in the 'eq' set. Then the pointer from node X_2 is redirected from **2** to **1**, and as no further CDD nodes reference **2** this node can be deleted.

If the *call stack* node 'n' has been changed by the previous lines, another node existing in the CDD '$n' \in V$' might become syntactic equivalent with n, if this is the case, the set $\langle n', n \rangle$ is added to the 'eq' set, so that the *reduction* operation on other computer nodes, will see that 'n'' is equivalent with 'n', and change it's pointer(s) from 'n' to 'n'', and delete 'n'. This is handled by line `8` through `9`. To see and example on this please refer to figure 4.15, where $X2$ has just been made to point to 1, and has the same interval as node X_1, this makes CDD node X_1, and X_2 syntactic equivalent, and $\langle X_1, X_2 \rangle$ is added to 'eq'. When the **foreach** iteration reached CDD node Z_2, it discovers that X_1 is equivalent with X_2, and redirects it's pointer from X_2 to X_1, and delete node X_2.
To ensure that all nodes is removed it is important to keep the nodes in the *call stack* sorted, so that the nodes with the higher types, is run through the iteration before nodes with smaller types (nodes with high types, is referenced by nodes with smaller types). And *reduction* can only be performed bottom-up.

Line `10` through `12`, handles the case when the local *reduction* recognizes that two nodes with the first type handled by computer node M_i, then computer node M_{i-1} might use this information to make further reduction. To see an example on this, please refer to figure 4.14, where the lower computer node has recognized that node Y_1 and Y_2 is equivalent, and adds $\langle \mathbf{2}, \mathbf{1} \rangle$ to $newEQ$, which is consequently sent to the upper node. Note that in the figure, the reference from 2 is redirected to point to Y_1, to ensure that *inclusion/union* requests made before the local *reduction* has run in the prior node, still succeed (such request might still contain the communication node 2).

Finally line `13` and `14` sends $newEQ$ to the previous node, to initiate a local *reduction* there. This is what is implicitly done between the (a) and the (b) part of figure 4.14.

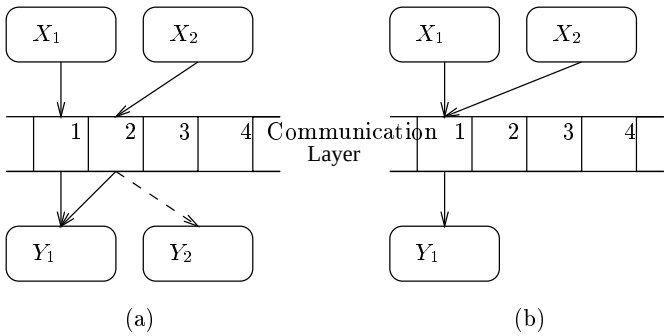


Figure 4.14: Reduction: If two subtrees is equal one may be deleted and the other one used instead. Figure (a) show the situation before the *reduction* request is send to the upper node. Node Y_1 and Y_2 has just recognized to be syntactic equal by the lower node. Figure (b) show the situation after the information $1_{comm} = 2_{comm}$ has been send to the upper node, and this node has made a *reduction*.

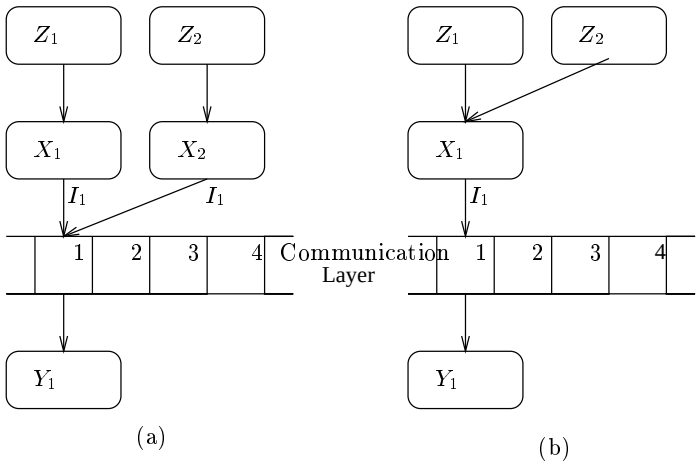


Figure 4.15: Reduction: The *reduction* from figure 4.14 lead to X_1 and X_2 became syntactic equivalent (a), and further reduction made possible (b)

Example

An example demonstrating the functionality of the *union* operation using *call stack* and *temporarily* hash table. And the functionality of the *reduction* operation is given in appendix A

4.4.4 Backtrace Algorithm

In this section we describe the *backtrace* algorithm. The *backtrace* algorithm is used when the *inclusion test* fails, and before the *union* operation is started. The idea of the *backtrace* algorithm is to find a subpart of the CDD from the bottom element, the **true** node, that covers exactly the same *Zone* as the corresponding nodes in the S-CDD that failed the *inclusion test* (see figure 4.16). The main purpose of the *backtrace* algorithm is optimization of memory usage during the *union* operations, as we expect the *backtrace* algorithm to minimize the overall runtime stack, and thus to minimize the time spend in the *reduction* phase. The CDD node found by the *backtrace* operation may only be used when unioned with the **false** node.

The *backtrace* algorithm works as follows:

```
1: CDD_node Backtrace(n_s ∈ N_s, n ∈ V)
2: begin
3:       if type(n) = t_server_comm then send Backtrace request to M_{i-1}
4:       if ∃n' ∈ V | type(n') = type(n_s) ∧
5:             ∀(I, m) ∈ succ(n')|(I = I(n_s) ∧ m = n) ∨ (m = false) then
6:                   return Backtrace(parent(n_s), n')
7:       else return n
8: end
```

The algorithm works as follows:

Overall: The algorithm is initially called with the **true** node and the S-CDD node with the largest type, that is, the S-CDD node referring the **true** node.

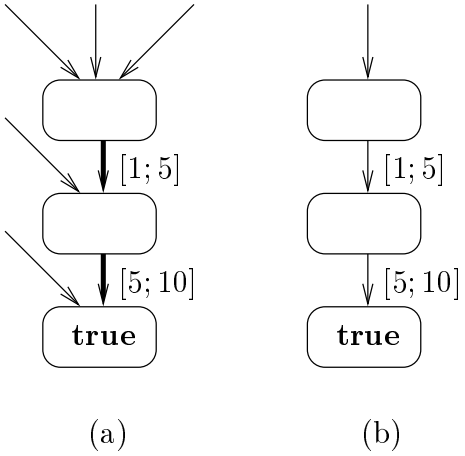


Figure 4.16: Example on the *backtrace* algorithm. The *backtrace* algorithm is run on the CDD (a), with the S-CDD (b) as argument. And the algorithm should return the marked path.

The idea is to traverse the CDD backwards, finding a single stringed subparts of the CDD that covers the exact same part of the *Zone* that the S-CDD covers. When this point is found the *union* becomes easier as we can attach to this node when reaching a node with the same type in the S-CDD, is this S-CDD node should be unioned with the **false** node.

line 3: Line `3` checks whether a trace has been found up to the communication array, if that is the case the *backtrace* request must be send to the previous node, to continue there. With the message the communication entry *id* is send, so that the *backtrace* operation can continue there starting at this node.

line 4-5: The **if** condition of line `4` and `5`, checks whether there exists a node in the existing CDD, which has exactly one interval, ranging over the same interval, and leading to a CDD node which describes the same sub zone as the child to the S-CDD node given. If that is the case, the CDD node 'n'' describes the same zone as the S-CDD node n_s, and the recursion continues to see whether such a trace also exists for the parent to n_s, this recursion is called in line `6`.

line 7: If such a node did not exist, the longest *backtrace* for the S-CDD, is the CDD node, which this recursion were called with (n), and this node is returned.

4.5 Node Representation

In this section we give an overview of the node representation in the CDD data structure. We consider the internal representation of nodes, as well as how to store nodes. First we consider how to store nodes, and after this the internal representation is covered.

Memory Management

The storage of the CDD nodes can be seen as simple memory management, as we have to allocate nodes (creating nodes) and deallocate nodes (when changing nodes). The memory management must support a minimum of memory usage overhead, that is memory used only for memory

management purposes. It should also favor efficiency of the implementation, and if possible support data locality to optimize cache accesses, as it has been done in [13].

Different solution to store nodes is stated and discussed in the following:

Segregated Keep an array of nodes for each node size.

Standard Allocate each node separately using `malloc`.

Locality Make special memory management to enhance data locality.

Segregated: A CDD consists of inner nodes of variable size, as there can be an arbitrary finite number of successors to a node. This complicates the storage of these nodes, as we cannot use an array as nodes differ in size. Nodes having the same size could be kept together using a segregated memory allocation approach, by keeping a free list for each node size - see [7] or [15][pp.36] for details. This might affect data locality if nodes are accessed using breadth first search patterns as this accesses nodes having the same type consecutively, but they might differ in size, leading to different locality for storage. Accessing nodes in a depth first manner, makes it almost impossible to enhance data locality as we cannot support data locality for all paths in the CDD. Determine the ratio at which they each are used is not trackable as the *Passed* list is a dynamic data structure, new information is added all the time.

Standard: Nodes could also be individually allocated using `malloc`, but the algorithms later in this chapter shows that we will encounter a high rate of changing CDD nodes, thus requiring to reallocate the node when changing them. The relative high cost of calling `malloc`, as it might involves a kernel trap, together with the high rate of changing nodes leads to a high cost for this approach, memory wise the cost is also high due to the memory used for headers internally in the memory management.

Locality: The algorithms described in section 4.1 are depth first on each computer node. Thus making a special memory management to enhance data locality becomes difficult, as many traces share some nodes in the the CDD data structure. Which of these traces that should be given precedence to others traces, is difficult as the data structure changes all the time as new zones are added.

We conclude that the best memory management method for our purpose is the segregated approach. This is based on the great number of allocation/deallocations needed, together with that the number of different node sizes is expected to be limited. One problem with this memory management, is that each block need some header information about the nodes in this block. These information is: *size* and *free_ elements*, where *free_ elements* should be a mechanism for finding free elements in a cheap way. This overhead introduced by the header argue for large blocks (thus minimizing the relative overhead), but the fact that nodes of size 'x' cannot be allocated in blocks with size 'y', where $x \neq y$, argue for smaller blocks, the discussion of block sizes is taken up later, as the size of the blocks is given another purpose.

The *free_ elements* item in the header, should provide a cheap way of finding an unused element. For this purpose a linked list approach is chosen. That is all free elements is put into a linked list of free elements, where the first 32 bits of the element is used to store a pointer (which, when allocated is used for real data), this way no memory is lost to hold this free list, only one header for the *free list* need to be stored in the header.

Now back to the discussion for the block size. We have chosen to extend the segregated memory management with an idea from the LISP interpreters [2], where data of the same size is stored in arrays that is aligned on a specific memory boundary. If we align each array at a 64k-byte boundary, meaning we allocate arrays of size 64k-bytes, then we can access these blocks with 16 bit pointers(*0xXXXX0000*), as the 16 least significant bits always will be zero.
As all elements we allocate is at least 13 bytes (explained later), a maximum of 5041 elements can be allocated in a single block. For addressing one of these 5041 elements we need only 13 bits, and 16 bits for accessing the block, leaving 3 bits in a 32 bit reference to other purposes. Using such bits for other purposes than addressing is called a *tagged* architecture.
Examples of tagged architectures are the LISP machine that uses tag bits for runtime determination of data types.

This leads to how we internally represent the nodes.

4.5.1 Compact Representation

In this section we describe the possibilities of the internal representation of the CDD node. In the following we give the representation of a single CDD node.

Each node holds a pointer, which has been added for time optimization purposes.
This optimization is used for *reduction*. Whenever a new node is created it must be checked whether a CDD node already exists in the CDD data structure which has the same syntactic description. To do that all nodes has to be examined, to avoid that all nodes in the CDD data structure need to be examined, all nodes is put in to a hash data structure. The pointer described is the used as a linked list in each hash bucket, this is an optimization (both runtime and memory wise) to hold an external overflow bucket. This pointer (the *next-hash* pointer) is not shown in the figures of CDD nodes, as it does not hold any semantic information.

Minimum Information

To minimize the memory usage when making the CDD data structure, we need to find the minimum required information and store this as compact as possible. Obviously we need to store the type of the CDD node and the reference count, designating the number of parent nodes, together with all the intervals and references to other CDD nodes. Each interval needs a pointer to the child node and the bounds on the interval, but we need not represent edges leading to the **false** node. Also $-\infty$ and ∞ does not need explicit representation, as they can be represented implicitly, as shown later.

The intervals is bounded by integers of varying values, thus we try to represent them with as few bits as possible, e.g. values between -128 to 127 should only be represented by one byte and so on, the most significant bit being the sign bit.

As described earlier the only operators we need to represent is $\leq$, as we can simulate all others with this, see section 1.1.1. To summarize the basic idea is first to simulate $>, \geq$ with $<, \leq$ by negation, we then multiply all bounds by 2 and simulate $<$ with $\leq$ by subtracting 1 from the strict bounds.

To avoid representing the **false** edges we design a data structure that can contain the node type, reference count, intervals and pointers jointly.
A node consists of first a "Node Header" followed by an arbitrary combination of a "Interval

pointer" and an "Interval integer" see figure 4.18. There can be two patterns of the "Interval pointers" and "Interval integers". A pointer is followed by either one or two integers, that again are followed by a pointer, see example in figure 4.17:
(a) represents the intervals $]-\infty;2[\rightsquigarrow$ ptr1, $[2;3[\rightsquigarrow$ **false**, $[3;12]\rightsquigarrow$ ptr2, and $]12;\infty[\rightsquigarrow$ **false**.
(b) represents the intervals $]-\infty;1[\rightsquigarrow$ **false**, $[1;2[\rightsquigarrow$ ptr3, $[2;3[\rightsquigarrow$ ptr4, and $[3;\infty[\rightsquigarrow$ **false**.

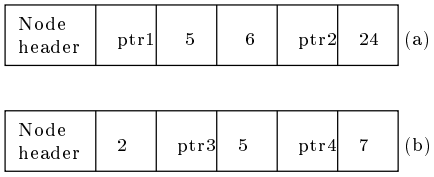


Figure 4.17: Two examples of CDD nodes and their interval representation

The first element is a pointer if the interval starts at $-\infty$ otherwise an integer, likewise if the last entry is a pointer the interval ends at ∞.
There two integers x, y following one another if the interval between x and y leads to **false**.

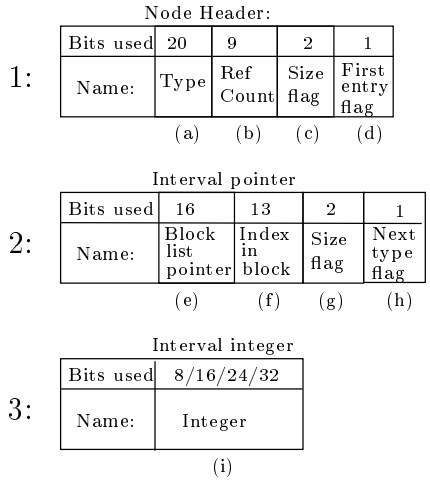


Figure 4.18: Three figures representing the node layout, being the node header, node pointer and the node integer.

The three different items in 4.18 are described in detail below, we assume w.l.o.g that memory segments are allocated on 64KB boundaries:

Node Header: Represents the node type (a), reference count(b) and some flags (c),(d). The node type (a) is represented by 20 bits meaning that we "only" can distinguish 2^{20} different type of nodes. This might prove to be a limitation for very large TA[1] models, but it can easily be extended by adding more bits to represent the type. The reference count in (b) can hold references of up to 512 parent nodes, this again might prove insufficient, but again it can easily be extended. The flag in (c) designates the size of the first entry, if this is an integer, otherwise these bits are unused. The flag in (d) designates the type of the first entry in the interval, the type can either be a pointer or an integer. If the first type is a pointer then the interval implicitly starts at $-\infty$, and ends at the integer following the pointer.

[1]Timed Automata

Interval Pointer: Represents the layout of the node reference, with the block pointer(e), the index bits(f) and two flags (g),(h). The block pointer is the bits used for accessing the block, as these are aligned at 64 K-bytes it is sufficient to use 16 bits here. The index bits(f) are used to address which entry in the array we are accessing, as the smallest block we store are :

- A node header - 32 bits
- One pointer, the interval implicitly starting at $-\infty$ - 32 bits
- An integer of size 8 bits being in the interval $[-127; 128]$.

This sums up to a minimum node size of 9 bytes, plus a pointer for a hash list which adds 4 bytes for a total of 13 bytes.

The size flag (g) consists of two bits, that are used to designate the size of the next (possible two) integer(s). If the are more than one integer then they will need to be stored with the size of the largest of them. The last flag (h) designates whether the next two entries are both integers or an integer and a new reference.

Interval Integer: This is the interval integer, that can be four different sizes, 1-4 bytes (*size-flag* `00`, `01`, `10`, `11`).

When performing the reduction, parent pointers would come in quite handy, but the memory usage in each node will increase dramatically as: The minimum node size was 13 bytes, adding one parent pointer at size 4 bytes, will increase the memory usage by $\frac{4}{13} * 100 \approx 31$ %, this being the best case, as a node can have an arbitrary number of parents. This problems grows when nodes has more than one parent, then a list of parents has to be stored. Thus we consider it a bad idea to store parent pointers, as the main purpose of this project is to enhance memory utilization to allow verification of larger models.

4.5.2 Node Representations

As it is expected that checking the *type-flag* and the *size-flag* might take up some time, as well as the converting between the different size of integer representation. Three different node representations is implemented in this project.

Memory Representation 1 This is the same node representation as used in the current Uppaal version using the CDD data structure for its passed list. That is all integers is represented by 32-bits, and pointers to **false** is also represented, an example on such a node is given in figure 4.19(a).

Memory Representation 2 This node representation introduce the *type-flag*, by **not** representing pointers leading to **false**. But all integers is still represented by 32-bits. An example on this representation is given in figure 4.19(b).

Memory Representation 3 In the last node representation implemented, we use all memory minimizing techniques. That is both the *type-flag* as well as the *size-flag* is used. This means that pointers to **false** is not represented, and integers is represented by the least number of bytes possible. An example on this representation is given in figure 4.19(c).

Implementing all three node representations also allow us to check how much memory is saved using the alternative node representations.

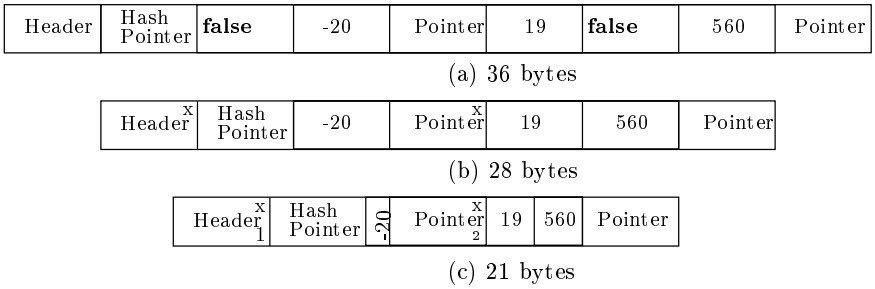


Figure 4.19: Show an example node in the three different node representations implemented. 'x"es in the figure is the *type-flag*, and small numbers in the header/pointers, is used to represent the size in bytes of the next integer(s). The node represented has the interval: $]-\infty;-10[\rightsquigarrow$ **false**, $[-10;9] \rightsquigarrow$ Pointer1, $]-10;280[\rightsquigarrow$ **false**, and $[280;\infty[\rightsquigarrow$ Pointer2. The sizes of the different nodes, using the different node representations is: Memory Representation 1: 32 bytes, Memory Representation 2: 24 bytes, and Memory Representation 3: 17 bytes.

4.5.3 Summary

To summarize we implement three different node representations, to see how much memory can be saved using different node representations, and also to see at what runtime prize this memory optimization comes at.
The distributed *inclusion* and *union* operations were designed almost as the non-distributed versions. The only difference is that whereas the non-distributed operations run depth first, the distributed operations run depth first on each node, before sending the request to the next computer node. Two additional operations were described: *backtrace* and *reduction*. The *backtrace* could also be used in a non-distributed environment, its purpose were to reduce the amount of work needed by the *union* operations, at it may use the found trace, whenever a *union* should be performed with the **false** node. The last operation described were the *reduction* operation, which has to be implemented as the distributed *union* operation cannot reduce the CDD data structure as it does not work in a global depth first fashion.

Semantics

Chapter 5

In this chapter we present the semantics of selected operations on the CDD data structure. First we provide a proof that changing the data structure into a distributed one does not change the semantics of the CDD. Secondly the semantics of common operations on our two main data structures are made, finally semantics proofs for the algorithms described in the previous chapter are conducted.

5.1 Semantics of the Distribution

The distribution of the data structure describes the partitioning of the data structure among several computer nodes. This is done horizontally for reasons described earlier, and additional nodes are added to the data structure to provide a communication layer between computer nodes. These extra nodes can be interpreted as CDD nodes with a special type (t_{comm}) with one outgoing edge not leading to the **false** node. This edge range over the interval '$]-\infty;\infty[$'. Thus we need to prove that the two CDD's in figure 5.1 semantically describes the same area.

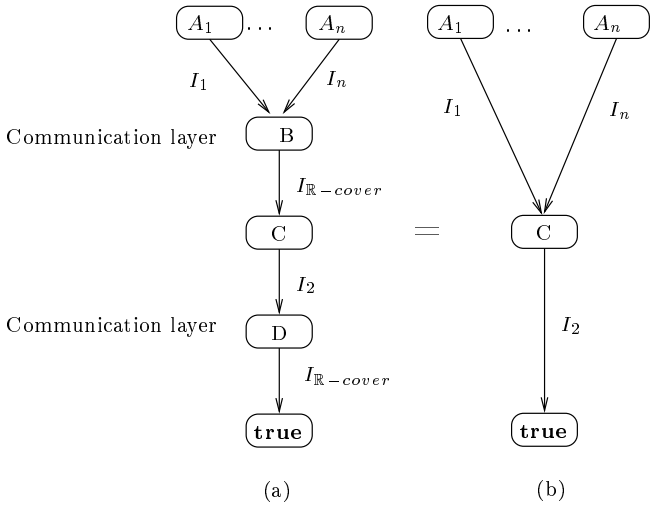


Figure 5.1: Figure (a) shows a distributed CDD with communication CDD nodes being node B and D, interval $I_{\mathbb{R}-cover}$ is an interval forming an $\mathbb{R}$ − *cover*. Figure (b) shows the equivalent CDD that describes the same *federation* as (a), as the redundant nodes B and D are omitted

The proof is simple as the reducedness properties of the CDD data structure already states that nodes forming an $\mathbb{R}$ cover can be omitted as they do not contribute with any restrictions to the *federation*. Thus we can safely conclude that the two CDD's in figure 5.1 semantically describes the same *federation*. As this applies to all CDD's we can w.l.o.g use the standard algorithm semantics on the distributed CDD data structure to prove soundness and completeness of the algorithms. This prove is extended when some semantic is defined for the CDD data structure.

5.2 Data Structures

The two main data structures for which we give semantics are the ordinary CDD data structure and the S-CDD data structure, where representing a *Zone* as the DBM's can be mapped directly to S-CDD's.

Semantics for CDD's

Each CDD node is a (n+1) tuple $\langle t, [I_1, T_1] \dots [I_n, T_n] \rangle \in \mathcal{N}$, where $t \in \mathcal{T}$, $I_i \in \mathcal{I}$, and $T_i \in \mathcal{N}$, where $p \in \{1 \dots n\}$. Besides these inner nodes two terminal nodes exists:

- **true** $\in \mathcal{N}$
- **false** $\in \mathcal{N}$

The first operation for which a semantic is described is traversal through the data structure given a clock valuation $v \in \mathcal{V}$:

- $[\![\textbf{true}, v]\!] = true$
- $[\![\textbf{false}, v]\!] = false$
- $[\![\langle t, [I_1, T_1] \dots [I_n, T_n] \rangle, v]\!] = [\![T_i, v]\!]$, where $v(t) \in I_i$

The first item states that no matter which valuation is given to the **true** node, the result is `true`. The second item states that no matter which valuation is given to the **false** node, the result is `false`. The third item states that for a general CDD node, the result of the valuation is the result of the valuation of the child node T_i, where $v(t) \in I_i$.

The second operation for which semantics are described is what *federation* the CDD describe:

- $[\![\textbf{true}]\!] = \mathcal{V}$
- $[\![\textbf{false}]\!] = \emptyset$
- $[\![\langle t, [I_1, T_1] \dots [I_n, T_n] \rangle]\!] = \bigcup_{i=1}^{n} \{I_{i_t} \cap [\![T_i]\!]\}$, where the notation I_{i_t} is based on the syntax used in section 3.4, and denotes the interval I_i restricted in the dimension of t.

The first item describe that all valuations is accepted by the **true** node. The second item describes that no valuations is accepted by the **false** node. And finally, the third item describes the set of valuations accepted by a general CDD node. This is made from the union of all the set of valuations each of it's successors accept, where the set of valuations the successor i accept is $I_i \cap [\![T_i]\!]$.

Semantics for S-CDD's

Each S-CDD node is a 3-tuple $\langle t, I, T_s \rangle \in \mathcal{N}_s$, where $t \in \mathcal{T}$, $I \in \mathcal{I}$, and $T_s \in \mathcal{N}_s$. Besides inner nodes, a single terminal S-CDD node exist:

- $\mathbf{true} \in \mathcal{N}_s$

The first operation for which a semantics is described is traversal through the data structure given a clock valuation $v \in \mathcal{V}$:

- $[\![\mathbf{true}, v]\!] = true$
- $[\![\langle t, I, T_s \rangle]\!] = v(t) \in I \wedge [\![T_s, v]\!]$

The first item states that all valuations are accepted by the **true** S-CDD node. The second item states that a valuation is accepted by a S-CDD node if the value of the valuation for this type $v(t)$ is included in the interval for this node I, and if its single child node also accepts the valuation v.

The second operation for which a semantics is described is which *federation* an S-CDD describes:

- $[\![\mathbf{true}]\!] = \mathcal{V}$
- $[\![\langle t, I, T_s \rangle]\!] = I_t \cap [\![T_s]\!]$

As where the case for the CDD, all valuations are accepted by the **true** S-CDD node. The set of valuations accepted by the general S-CDD node, are the interval leaving it intersected by the valuations accepted by it child.

5.3 Distribution

Back to the proof that the semantics for the single processor CDD data structure easily can be mapped to the distributed version.
To distribute the CDD data structure additional communication layers were added. If such a layer is added between types t_i and t_{i+1}, then whenever a CDD node n_{src} with type t_i or less wants to refer to a node n_{dest} with type t_{i+1} or higher, then this reference is made to point to a communication node n_{comm}, which only has a single successor $(]-\infty;\infty[, n_{dest})$. In this section we show that the set of valuations described by node n_{src} is the same whether it refer n_{dest} directly, or indirectly through n_{comm}.
For the following proof n_{src} denotes a CDD node with type t_{i-1} with a single successor, with interval I, leading directly to n_{dest}. And $n_{src'}$ denotes the same node, only it successor points to n_{comm}, which again refers to n_{dest}. Referring to the syntax used in section 3.4 on page 23, we need to prove that the set of valuations described by n_{src} is equivalent with the set of valuations described by $n_{src'}$, that is we need to prove that:

$$\begin{array}{rcl} [\![n_{src}]\!] & = & [\![n_{src'}]\!] \\ & \Updownarrow & \\ I \cap [\![n_{dest}]\!] & = & I \cap]-\infty;\infty[_{t_{comm}} \cap [\![n_{dest}]\!] \end{array} \tag{5.1}$$

As each type can be seen as a coordinate in a multi dimensional space, and the type t_{comm} is orthogonal to all other types, the intersection with $]-\infty;\infty[_{t_comm}$ is not a restriction to the set of valuations described by $n_{src'}$, and naturally an intersection cannot extend the set of valuations described by $n_{src'}$.

5.4 Operations

The operations that are performed on the used data structures are *inclusion test*, *backtrace*, *reduction* and *union*. The *inclusion test* must check whether a S-CDD is included in a CDD. The *union* operation must perform union between a S-CDD and a CDD.

The semantic proofs of the operations is based on set operations, that is, the set of valuations described by the CDD and S-CDD respectively. In the *inclusion test* it is argued that for a S-CDD to be included in a CDD, the interval of the CDD must be equal to or greater then the interval of the S-CDD in **all** dimensions. That is for all types t, it must be true that: $I_t^{S-CDD} \subseteq I_t^{CDD}$, where $I_t^{(S)-CDD}$ denotes the interval in dimension t for the (S)-CDD.

5.4.1 Inclusion Test

The semantic rules are[1]:

I $[\![\mathbf{true} \subseteq \mathbf{true}]\!] = true$

II $[\![\mathbf{true} \subseteq \mathbf{false}]\!] = false$

III $[\![\langle t, I, T_s\rangle \subseteq \mathbf{false}]\!] = false$

IV $[\![\langle t, I, T_s\rangle \subseteq \mathbf{true}]\!] = true$

V $[\![\mathbf{true} \subseteq \langle t, [I_n, T_n] \dots [I_n, T_n]\rangle]\!] = false$

VI $$[\![\langle t_s, I, T_s\rangle \subseteq \langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = \bigvee \begin{cases} 1: t_s < t \wedge [\![T_s \subseteq \langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] \\ 2: t_s = t \wedge \bigwedge\{[\![T_s \subseteq T_p]\!], p \in \{1 \dots n\} \mid I \cap I_p \neq \emptyset\} \\ 3: t_s > t \wedge \bigwedge\{[\![T_s \subseteq T_p]\!], p \in \{1 \dots n\}\} \end{cases}$$

To prove that this *inclusion test* is correct we prove that the semantic rules is sound and complete. First for the soundness, we prove that the set of valuations described by the S-CDD are included in the set of valuations described by the CDD:

Rule I: From the previous stated semantics, this rule states that $\mathcal{V} \subseteq \mathcal{V}$ which is trivially true.

Rule II: This rule states that the *federation* covered by the **true** node of the S-CDD covers $\mathcal{V}$, whereas the *federation* covered by the **false** node of the CDD covers $\emptyset$, and clearly $\mathcal{V} \not\subseteq \emptyset$, as $\mathcal{V}$ cannot be false, and thereby describe $\emptyset$.

Rule III: The *federation* covered by $[\![\langle t, I, T_s\rangle]\!]$ cannot be empty (this would result in an empty *federation*, and thus not represented, due to the reduction rules of S-CDD's and CDD's), and clearly $[\![\langle t, I, T_s\rangle]\!] \not\subseteq \emptyset$.

Rule IV: This rule is trivial, as any subset of $\mathcal{V}$ clearly is a subset of $\mathcal{V}$.

Rule V: The semantic rule for S-CDD's defines the **true** node to cover $\mathcal{V}$, whereas the *federation* covered by $\langle t, [I_1, T_1] \dots [I_n, T_n]\rangle$ cannot cover $\mathcal{V}$ as it would violate the reducedness rules of the CDD data structure, therefore the S-CDD cannot be included in the CDD.

[1] The rules are read as $[\![S - CDD \subseteq CDD]\!]$.

Rule VI: We consider the 3 sub cases separately,

1: If the type t_s of the S-CDD is smaller than the type t of the CDD, it means that the CDD implicitly contains a node with type t_s that covers $\mathbb{R}$, and the argument of Rule IV applies here. So for this type the S-CDD is included in the CDD as the corresponding type for the CDD is non existing thus forming an $\mathbb{R}$-cover. So we traverse further down the S-CDD to check if the child node of the S-CDD is covered by the CDD node $\langle t, [I_1, T_1] \dots [I_n, T_n] \rangle$.

2: If the two types are identical, we use the semantic set description for the S-CDD and CDD to prove the semantics for the *inclusion test*:
Showing S-CDD $\subseteq$ CDD is equivalent to show that S-CDD $\cap$CDD = S-CDD
From the semantic definition of which *Zone/federation* the S-CDD and CDD describes respectively, we need to prove the following:

$$(I_s \cap [\![T_s]\!]) \cap \bigcup_{i=1}^{n} \{I_{i_t} \cap [\![T_i]\!]\} \quad = \quad I_s \cap [\![T_s]\!] \tag{5.2}$$

Which can be rewritten as:

$$I_s \cap [\![T_s]\!] \cap (I_{1_t} \cap [\![T_1]\!] \cup \dots \cup I_{n_t} \cap [\![T_n]\!]) \quad = \quad I_s \cap [\![T_s]\!] \tag{5.3}$$

$$\Updownarrow$$

$$(I_s \cap (I_{1_t} \cup \dots \cup I_{n_t})) \cap ([\![T_s]\!] \cap ([\![T_1]\!] \cup \dots \cup [\![T_n]\!])) \quad = \quad I_s \cap [\![T_s]\!] \tag{5.4}$$

$$\tag{5.5}$$

The next step taken, is to remove all successors of the CDD node, which interval intersected by I_s equals $\emptyset$, when removing the intervals, the child nodes referred by these are also removed, as these only matter for the set of valuations described if the edge leading to it has a non empty intersection with I_s.

$$I_s \cap \bigcup_{p \in \{1 \dots n\}} \{I_p | I_p \cap I_s \neq \emptyset\} \cap [\![T_s]\!] \cap \bigcup_{p \in \{1 \dots n\}} \{[\![T_p | I_p \cap I_s \neq \emptyset]\!]\} \quad = \quad I_s \cap [\![T_s]\!] \tag{5.6}$$

From the syntactic definition of the CDD, we know that the intervals of each CDD-node must form an $\mathbb{R}$-cover $\bigcup_{i=1}^{n} \{I_i\} = \mathbb{R}$, so the following must hold:

$$I_s \quad \subseteq \quad \bigcup_{p \in \{1 \dots n\}} \{I_p | I_p \cap I_s \neq \emptyset\} \tag{5.7}$$

So it is known that for eq. 5.6 to hold it is sufficient for the following to hold:

$$[\![T_s]\!] \cap \bigcup_{p \in \{1 \dots n\}} \{[\![T_p]\!] \,|\, I_p \cap I_s \neq \emptyset\} \quad = \quad [\![T_s]\!] \tag{5.8}$$

That is the question is back to:

$$[\![\langle t_s, I_s, T_s \rangle \subseteq \langle t_2, [I_1, T_1] \dots [I_n, T_n] \rangle]\!] = \bigwedge \{[\![T_s \subseteq T_p]\!], p \in \{1 \dots n\} \,|\, I_s \cap I_p \neq \emptyset\} \tag{5.9}$$

$$\text{for } t_s = t_2.$$

which were our semantic definition.

3: If the type t_s of the S-CDD is larger than the type t of the CDD, then the S-CDD implicitly contains a node with type t that covers $\mathbb{R}$. Therefore rule **VI**(2) can be reused here, but now all successors of the CDD node need to be examined, as all intervals intersected by $\mathbb{R} \neq \emptyset$. To the semantic for this case is:

$$[\![\langle t_s, I_s, T_s\rangle \subseteq \langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = \bigwedge_{p \in \{1 \dots n\}} \{[\![T_s \subseteq T_p]\!]\},\ for\ t_s > t$$

Completeness of the *inclusion test*

For semantic rules to be complete we need to prove that all syntactic correct inputs are covered by some semantic rule.

To do that we take the cross product between the syntactic legal input from the S-CDD (2 possible syntactic inputs), and the CDD (3 syntactic inputs), yielding that there should be 6 rules to cover all possible inputs, see table below.

S-CDD
true
$\langle t_s, I, T_s\rangle$

$\times$

CDD
false
true
$\langle t, [I_1; T_1] \dots [I_n; T_n]\rangle$

$=$

Input:	S-CDD	CDD
1	**true**	**true**
2	**true**	**false**
3	$\langle t, I, T_s\rangle$	**true**
4	$\langle t, I, T_s\rangle$	**false**
5	**true**	$\langle t, [I_1, T_1] \dots [I_n, T_n]\rangle$
6	$\langle t, I, T_s\rangle$	$\langle t, [I_1, T_1] \dots [I_n, T_n]\rangle$

Input 1 through 5 is trivially covered by rule 1 through 5 in the semantic definition of the *inclusion test*.

Input 6 is covered by rule 6 in the semantic definition, but as the rule here is subdivided, we consider if the three sub rules together covers all possible legal inputs. The sub rules covers the cases where:

$$t_s < t\ ,\ t_s = t \text{ and } t_s > t$$

As these three rules trivially covers all possible situations of the types, we conclude that all possible legal input to the *inclusion test* is covered by a corresponding semantic rule yielding completeness.

5.5 Union

The semantics for the *union* between a S-CDD and a CDD is as follows, the rules are read as $[\![S - CDD \cup CDD]\!]$:

Rule I: $[\![\mathbf{true} \cup \mathbf{true}]\!] = \mathcal{V}$

Rule II: $[\![\mathbf{true} \cup \mathbf{false}]\!] = \mathcal{V}$

Rule III: $[\![\langle t_s, I, T_s\rangle \cup \mathbf{true}]\!] = \mathcal{V}$

Rule IV: $[\![\langle t_s, I, T_s\rangle \cup \mathbf{false}]\!] = [\![\langle t_s, I, T_s\rangle]\!]$

Rule V: $[\![\mathbf{true} \cup \langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = \mathcal{V}$

Rule VI: $[\![\langle t_s, I, T_s\rangle \cup \langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = [\![\langle t_s, I, T_s\rangle]\!] \cup [\![\langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!]$

To prove soundness of the *union* operation we prove each of the preceding rules:

Rule I: If both the existing CDD accepts all valuations, and a S-CDD that accepts all valuations is unioned, clearly all valuations is accepted by this union. $[\![\mathbf{true} \cup \mathbf{true}]\!] = [\![\mathbf{true}]\!] \cup [\![\mathbf{true}]\!] = \mathcal{V} \cup \mathcal{V} = \mathcal{V}$

Rule II: The same argumentation holds as for **Rule I** only, here the CDD covers $\emptyset$, but the S-CDD covers all valuations, so clearly now all valuations is covered. $[\![\mathbf{true} \cup \mathbf{false}]\!] = [\![\mathbf{true}]\!] \cup [\![\mathbf{false}]\!] = \mathcal{V} \cup \emptyset = \mathcal{V}$

Rule III: When a CDD accepting everything is unioned with a S-CDD covering the *federation* F_1 is unioned, the result is a CDD accepting. $[\![\langle t_s, I, T_s\rangle \cup \mathbf{true}]\!] = [\![\langle t_s, I, T_s\rangle]\!] \cup [\![\mathbf{true}]\!] = [\![\langle t_s, I, T_s\rangle]\!] \cup \mathcal{V} = \mathcal{V}$

Rule IV: If an empty CDD, is unioned with a S-CDD accepting some valuation, the resulting CDD must also accept the same valuation, and nothing else. $[\![\langle t_s, I, T_s\rangle \cup \mathbf{false}]\!] = [\![\langle t_s, I, T_s\rangle]\!] \cup [\![\mathbf{false}]\!] = [\![\langle t_s, I, T_s\rangle]\!] \cup \emptyset = [\![\langle t_s, I, T_s\rangle]\!]$

Rule V: If an S-CDD accepting all valuations is unioned with some CDD,the resulting CDD have to accept everything. $[\![\mathbf{true} \cup \langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = [\![\mathbf{true}]\!] \cup [\![\langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = \mathcal{V} \cup [\![\langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = \mathcal{V}$

Rule VI: If a general S-CDD is unioned with a general CDD, the resulting CDD should accept the union between the two CDD data structures. $[\![\langle t_s, I, T_s\rangle \cup \langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!] = [\![\langle t_s, I, T_s\rangle]\!] \cup [\![\langle t, [I_1, T_1] \dots [I_n, T_n]\rangle]\!]$

The completeness of the *union* operation can be shown in a similar way as the completeness proof of the *inclusion test*.

5.6 Semantics of Backtrace

In this section we provide some semantic proofs of the *backtrace* algorithm.
To summarize how the *backtrace* algorithm works, we try recursively to find nodes that covers exactly the same zone as the corresponding sub S-CDD we union with, we do this bottom up starting at the **true** node. The Algorithm for the *backtrace* algorithm is provided in section 4.4.4 on page 42

To prove the soundness of the *backtrace* algorithm we need to prove two things:

1: First we need to prove what kind of nodes can be used for the *backtrace* algorithm, and that reusing these nodes does not violate the semantics of the union, nor the reducedness properties of the CDD data structure. To prove this the following need to be proven.

- Restricting to search among nodes with only one child node is semantically correct.

- That the found CDD nodes describes the semantically correct valuation. That is the valuation described by the sub S-CDD.

- That the reuse of these nodes does not violate the reducedness properties of the CDD, data structure.

2: That these found nodes cannot be altered due to other union operations.

5.6.1 Reuse of nodes

We will prove that it is not allowed to reuse a CDD node if it has more than one child node not being the **false** node.

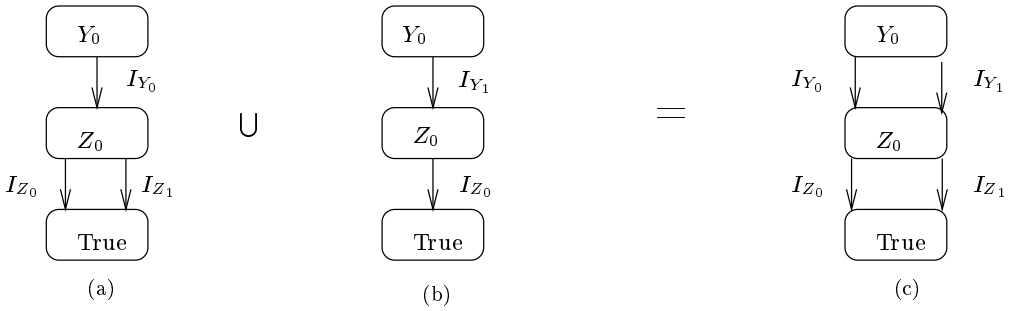


Figure 5.2: (a) is a CDD, (b) is a S-CDD and (c) is the union of (a) and (b), when reusing node Z_0 even though it have more than one child different from the **false** node

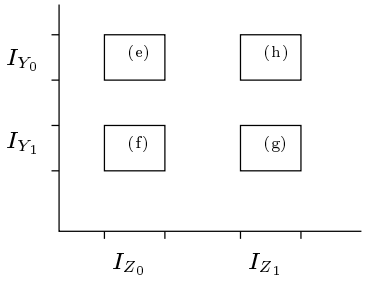


Figure 5.3: (e) and (f) are the area covered by the CDD in figure 5.2 (a), and (e),(f),(g) and (h) denotes the area covered by the CDD in figure 5.2(c). The (h) area should have been omitted as it denotes the area implicitly added by reusing node Z_0 in figure 5.2

In figure 5.2 we union the CDD (a) with the S-CDD (b), in this union we allow reusing node Z_0. The *federation* described by (a) is depicted as box (e) and (f) in figure 5.3.
The *Zone* described by the S-CDD in figure 5.2(b) is depicted as box (g) in figure 5.3, thus the union of the the CDD and the S-CDD should provide a *federation* describing box (e),(f) and (g). But reusing node Z_0 as in this union forms a *federation* describing all four boxes in figure 5.3.

Thus we conclude that it is not allowed to reuse a CDD node if it has more than one child node, and none of these are the **false** node. Note that this is a special case that only applies when

performing union of a CDD and a S-CDD as this would **not** be the case if both were CDD's as two CDD node could be isomorphic even though they have more than one child node.

From this proof we get that we only need to search among nodes with one child node, as using others implicitly will lead to inconsistency.

Also note that not being allowed to reuse a node with more that one child only apply during the *backtrace* algorithm. During the *reduction* phase all nodes are allowed to be reused.

5.6.2 Semantics of found CDD nodes

In this subsection we will argue that it is semantically correct to reuse a single stringed part of a CDD. A single stringed part of a CDD is a CDD node from which there is only one trace to the **true** node. The nodes are found as an exact match to the corresponding S-CDD node, and it is thus trivially to prove that they cover the same *Zone*, as we only search for nodes with one child as described in the previous subsection. Here we get that the found subpart of the CDD actually describes the same *Zone* as the S-CDD that we union with.

5.6.3 Reduction of CDD

As no nodes are added or modified by the *backtrace* algorithm, no reducedness properties can be violated by running this algorithm. The properties might be violated when the following *union* operation uses the found *backtrace* node, and does not construct new nodes for itself - this might lead to sub-optimal sharing.
If the *union* did not use the CDD trace found by the *backtrace* operation, it would construct the exact trace itself, as this is the only unique trace describing the valuation, after this trace would have been constructed the *reduction* phase would reduce the newly created trace to the one that would have been found by the *backtrace* operation, so using the CDD node found by the *backtrace* operations does not violate the reducedness properties of the CDD.

5.6.4 Change nodes

After the *backtrace* algorithm has run, and before the matching[2] *union* operation reaches the node where the *backtrace* algorithm stopped, other *union* requests might reach the node. To prevent these *union* operations from altering the found *backtrace* path, we prove that, if a node has two parents, or two different paths leading to it, then it is not allowed to change any successor nodes of the node having two parent nodes, as this would change the valuation of both paths.

We prove this by contradiction.

In figure 5.4 are three CDD's, (c) is the union of (a) and (b). Here we change the Y_0 node, even though it have two parents (X_1 and X_0).

Before the union $[\![X_0]\!]$ describes:

$$[\![X_0]\!] = I_{X_0} \cap [\![Y_0]\!] \Rightarrow [\![X_0]\!] = I_{X_0} \cap I_{Y_0} \cap [\![Z_0]\!]$$

[2]The matching *union* operation is the *union* operation started by the *backtrace* algorithm, and holds the same unique 'id' as the *backtrace* path found.

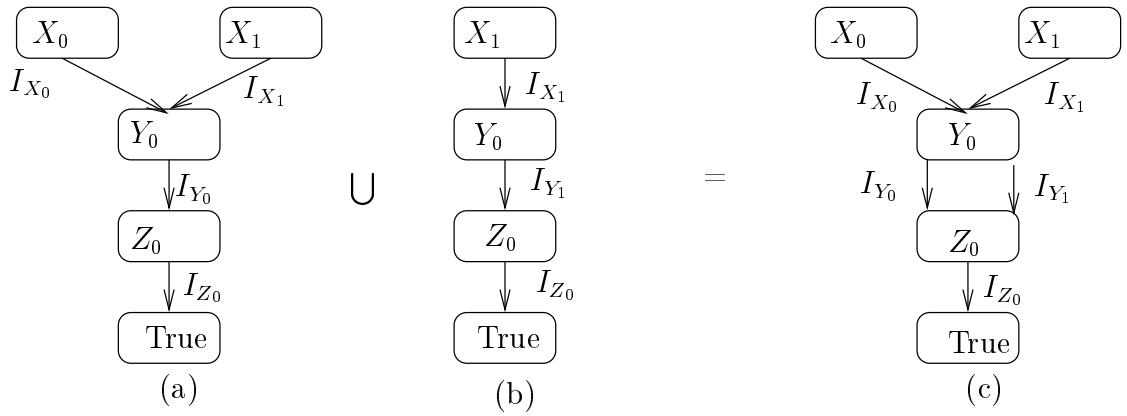


Figure 5.4: (a) is a CDD, (b) is a S-CDD, (c) denotes a CDD that is the union of (a) and (b). Note that $I_{Y_0} \cap I_{Y_1} = \emptyset$

The union of figure 5.4(a) and 5.4(b) is performed by adding an extra successor from node Y_0 to Z_0 with interval I_{Y_1}.

After the union $[\![X_0]\!]$ describe:

$$[\![X_0]\!] = I_{X_0} \cap [\![Y_0]\!] \Rightarrow [\![X_0]\!] = I_{X_0} \cap (I_{Y_0} \cup I_{Y_1}) \cap [\![Z_0]\!]$$

As $[\![X_0]\!]$ still should describe the same *federation* then, it must hold that:

$$I_{X_0} \cap I_{Y_0} \cap [\![Z_0]\!] = I_{X_0} \cap (I_{Y_0} \cup I_{Y_1}) \cap [\![Z_0]\!] \Rightarrow I_{Y_0} = I_{Y_0} \cup I_{Y_1}$$

This is clearly not possible as this implies that $I_{Y_1} \subseteq I_{Y_0}$ and this is a contradiction to ; $I_{Y_1} \cap I_{Y_0} = \emptyset$, and that neither of the intervals are allowed to be $\emptyset$ according to the definition of CDD's. From this proof we get that when a correct sub part of the CDD is found, it can safely be used as it will not be changed later. This is easily ensured by increasing the *ref_ count* of the nodes which will be reused.

To summarize what these three proofs provide:

- It is not allowed to change a node if it has more than one child node and none of these are the **false** node. This allows to conclude that in the *backtrace* algorithm we need only to search for nodes with one child, as using other nodes would possible introduce inconsistency.
- We have argued that the semantics of reusing a single stringed part of a CDD found using the *backtrace* algorithm, describes the same *Zone* as the corresponding part of the S-CDD yielding that the *backtrace* algorithm is sound.
- It is not allowed to change a node, that is changing the $I(node)$ set, if the node has more than one parent. This allows us to conclude that if we have found a single stringed part of a CDD via *backtrace* search from the **true** node, then this part cannot be altered by another *union* operation, as it will have more than one parent: the previous and the one using the *backtrace* algorithm.
  If a *union* operation on, say X_0 of figure 5.4(a), wants to change node Y_0, it has to make a copy of Y_0 and change the copy.

5.7 Reducing CDD's

Whenever a S-CDD is added to a CDD, some nodes may hold redundant information and must be reduced, section 3.1.1 described which rules must apply for the CDD being reduced, to summarize these rules:

- The CDD has maximum sharing: $\forall n, m \in \mathcal{N} \mid type(n) = type(m)$. whenever $succ(n) = succ(m) \Rightarrow n = m$
- All intervals are maximal: Whenever $n \xrightarrow{I_1} m, n \xrightarrow{I_2} m$, then $I_1 = I_2 \vee I_1 \cup I_2 \neq \mathcal{I}$

To ensure that these properties holds, a *reduction* is performed whenever a S-CDD has been added to the CDD. This is done by checking for violations of the prior two mentioned rules, and if any violations are found, they are corrected as follows:

- If a violation to the first rule is found, one of the nodes n, m is chosen for deletion (say 'n'), and all other nodes pointing to this node is redirected to point at the remaining node (node 'm').
- If a violation to the second rule is found, the two consecutive intervals is substituted by an new successor pointing to the same node, the successor is given the interval $I_1 \cup I_2$.

In this section we prove that these operations does not alter the semantics of the CDD.

- If node $m \in \mathcal{N}$ and $n \in \mathcal{N}$ is syntactically equivalent, the *federation* they cover is trivially the same. Therefore it is trivially allowable to interchange semantic equivalent nodes.
- The *federation* covered by a node $n \in \mathcal{N}$ is described by:
  $[\![\langle t, [I_1, T_1] \dots [I_n, T_n] \rangle]\!] = \bigcup_{i=1}^{n} \{I_{i_t} \cap [\![T_i]\!]\}$,
  if T_k and T_{k+1} is the same node, they describe the same *federation* say $[\![T_k]\!]$
  then $(I_k \cap [\![T_k]\!]) \cup (I_{k+1} \cap [\![T_k]\!]) = (I_k \cup I_{k+1}) \cap [\![T_k]\!]$.
  The interval $I_k \cup I_{k+1} \in \mathcal{I}$, is now called I_k, then the *federation* covered by n may be rewritten as:
  $[\![\langle t, [I_1, T_1] \dots [I_n, T_n] \rangle]\!] = \bigcup_{i \in \{1 \dots n\} \setminus k+1} \{I_i \cap [\![T_i]\!]\}$, which is equal to
  $[\![t, [I_1, T_1] \dots [(I_k \cup I_{k+1}) \cap T_k], [I_{k+2}, T_{k+2}] \dots [I_n, T_n]]\!]$ as stated.

5.7.1 Canonical

Unfortunately the reducedness properties of CDD's is not as nice as for BDD's. A reduced BDD make a canonical representation of a binary formula, whereas a reduced CDD does *not* make a canonical representation of a *federation*. To see why a reduced CDD does not make up a canonical representation, consider figure: 5.5, here the three reduced CDD's of (a), (b), and (c) represent the same *Zone*, namely the one presented in figure 5.5(d).
As can be seen none of the nodes in the CDD's of figure 5.5 does violate any of the reducedness rules.

That CDD's are not canonical may result in that the CDD's build in this project may not always take up the same number of CDD node as the order in which the S-CDD's is added might have an effect on the construction/reduction.

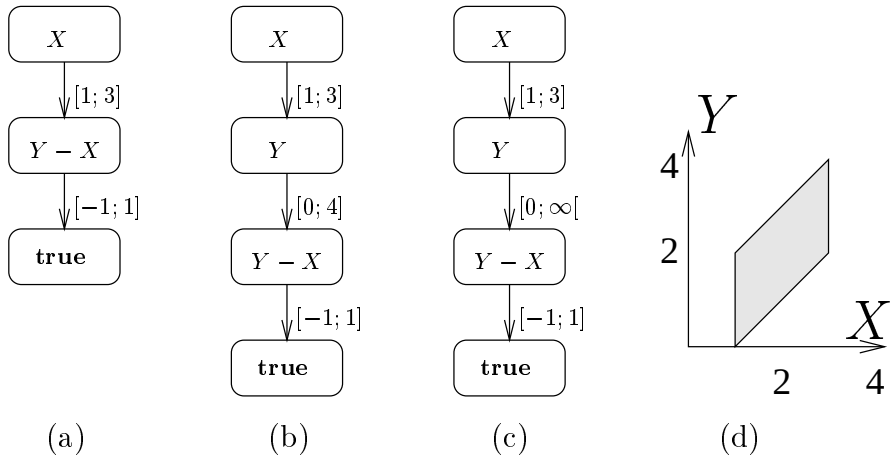


Figure 5.5: Reduced Ordered CDD's does not make a canonical representation for a constraint system. The three CDD's in (a), (b), and (c) represent the same *Zone*, namely the one presented in (d)

Cost Benefit Analysis

Chapter 6

In this chapter we give a cost benefit analysis for the different operations that is performed on the distributed CDD data structure. A purpose in our design were to minimize the number of messages send for each operation performed. First we analyzes the worst case number of messages send for each of the implemented operations in a Uppaal environment, thereafter the worst number of messages needed to explore a single state in Uppaal is analyzed, and finally it is analyzed how many messages can be saved by introducing groups. After having handled the number of messages needed for doing various operations and verifying states in Uppaal an analysis of the added memory overhead is provided.

6.1 Operations

The operations designed is the following:

- Inclusion test
- Backtrace
- Union
- Reduction

The following subsection describe the number of messages needed to run a single one of these operations. Figure 6.1 might help in understanding the analysis.

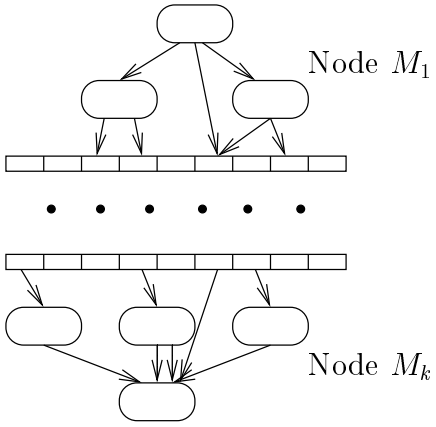


Figure 6.1: A test setup consisting of k computer nodes, each generating new states for exploration. Only the upper computer node can initiate *Inclusion Test* and *Union*. And only the bottom computer node can initiate *Backtrace* and *Reduction*

6.1.1 Inclusion Test

Whenever Uppaal discovers a new state it checks whether the state has been explored before performed through an *inclusion test*. The worst case scenario is that a computer node M_i different

from the top computer node M_1 discovers the state, then an *inclusion request* has to be send to the top computer node - that is one message. Then the top computer node M_1, initializes the *inclusion test*. Now the worst case scenario is that the *inclusion request* has to propagate to the bottom computer node M_k, which might succeed or fail. To propagate to the bottom computer node $k-1$ messages is needed. This gives a worst case of k message for a single *inclusion test*.

6.1.2 Backtrace

Whenever an *inclusion test* fails on a computer node, two actions takes place: First, the failed state is send to the *Waiting* list on a computer node - that is one message. Second, a *backtrace request* is send to the bottom computer node - that is two messages. The *backtrace request* can in worst case propagate to the top computer node, which adds additional $k-1$ messages. That is from the point were an *inclusion test* fails to the *union* is performed $k+1$ messages is send.

6.1.3 Union

Whenever a *backtrace* operation fails on a computer node, a union request is send to the top computer node - that is one messages. When the top computer node M_1 receives the *union request* it locates the handle and initializes the *union* operation. Most *union requests* must propagate to the bottom computer node of the local group to reach the **true** node, which adds additional $k-1$ messages. Even if the *backtrace* algorithm propagated to a computer node M_i, where $i \neq k$, the request might still propagate to the bottom computer node, as the CDD node reached by *backtrace* is only used if the S-CDD is to be unioned with the **false** node. That is in the worst case scenario, from a *backtrace* operations stops, to the *union* is performed k messages is send.

6.1.4 Reduction

When a *union* requests terminates on a computer node, most often M_k of the local group. This computer node has collected reduction information which may result in a reduction request to the previous computer node. Such a reduction request may propagate all the way to the top computer node of the local group. This gives that a *reduction* operations in worst case sends $k-1$ messages. Worst case in this sense is only the worst case of messages send, the higher the requests goes, the better sharing in the CDD data structure is given, yielding a better memory utilization.

6.2 State Exploration

In this section we use the prior operation costs to calculate the worst case number of messages needed to be send to explore a state found by Uppaal. First Uppaal initializes an *inclusion request* - which worst case took k messages if this request failed on the bottom computer node. If the request failed in the bottom computer node, the state is send to the *Waiting* list on a computer node - that is one additional message. But the *backtrace* requests need not be send, as it is already on the last node so the worst case number of messages so far is $k+1$ messages.

Then the *backtrace* request might propagate to the top computer node, which adds additional $k-1$ messages, if this is the case, the *union* request need not be send to the top computer node as it is already there. Now the *union* request need to propagate to the bottom computer node, which adds $k-1$ messages, the backtrace can only be used by nodes, that is to be unioned with the **false** node.
Finally the *reduction* operation adds additional $k-1$ messages to the total number of messages. To summarize the total worst case number of messages needed to be send to verify a single state is: $4k-2$ messages.

From the cost benefit analysis of the *union* operation it can be seen that the number of messages the union request sends worst case is unaffected of the *backtrace* operation. And if the *backtrace* operation is disabled the worst number of messages send for each discovered state is reduced to:

- Send *inclusion* request to top computer node - 1 message
- Propagate *incursion* request to bottom computer node - $k-1$ messages
- Send state to a *Waiting* list on another computer node - 1 message
- Send *union* request to top computer node - 1 message
- Propagate *union* request to bottom computer node - $k-1$ messages
- Propagate *reduction* request to top computer node - $k-1$ messages

Which sums up to $3k$ messages.

The purpose of the *backtrace* operation were not to save overall used memory, but to save temporary memory in form of call stacks, and temporarily not reduced CDD nodes. Further more the *backtrace* operation were made to save runtime, but as argued in the previous, $O(k)$ more messages has to be send, and it might not even help the *union* operation terminate earlier, so the *backtrace* algorithm is not implemented - and thus not tested. Another argument for not implementing the *backtrace* operation is that most of the advantage gained by the *backtrace* operation is solved by the *temporary hash* table, described in section 4.4.2. That is, it is expected that the little runtime advantage gained by the *backtrace* operation is lost due to the great number of message needed to be send, worst case. Further more the *union* request which must follow a *backtrace* must be delayed until the *backtrace* has finished it work, this has the disadvantage, that the state has to be stored until the *backtrace* has terminated, and that *inclusion test* might fail because of this delay.

The lowest number of messages which need to be send for a single state is $k-1$ messages. This is the case when the *inclusion test* succeed. That is when a new state is found at the top computer node at a group, and this request propagates to the bottom computer node - which is $k-1$ messages.

In an extremum the lowest number of messages send is zero messages. This is the case when the *inclusion test* succeed at the top computer node. But as this case were rarely seen in our preliminary tests, we argue that the best case complexity for a single state is $k-1$ messages.

6.3 Groups

If the number of groups used is different from the number of computer nodes used, then the same cost benefit analysis holds. In the analysis of the groups the number k is the number of computer nodes in the largest group. The current distributed Uppaal uses one computer node in each group, worst case this leads to 1 message. Namely sending the *inclusion* request to the correct computer node. No requests must propagate as all operations are done locally.

6.4 Memory Overhead

As a consequence of adding communication CDD nodes as described in section 4.3, the distribution of the data structure adds some memory overhead, which a single processor implementation would not have. In this section we describe how large this overhead is, as well as how this overhead can be removed by using distributed shared memory.

For this analysis some syntax is needed. Let $t_{i_{start}}$ be the first type located in computer node M_i, and let $t_{i_{end}}$ be the last type located on computer node M_i. And let $nr(t : \mathcal{T})$ return the number of nodes with type t. Let the computer nodes range from M_1 to M_k. Each computer node except the top computer node holds a *client communication array*, which takes up 1 word of memory. Each computer node except the bottom computer node holds a *server communication array* which also takes up 1 word of memory. That is the total number of words used for communications nodes in a group is:

$$\sum_{i=2}^{k}(nr(t_{i_{start}})) + \sum_{i=1}^{k-1}(nr(t_{i_{end}}))$$

Again if the extreme "one computer node per group" the memory overhead added is:

$$\sum_{i=2}^{1}(nr(t_{i_{start}})) + \sum_{i=1}^{0}(nr(t_{i_{end}})) = 0$$

So no memory overhead is added as expected.

From this analysis it can be seen that the fewer computer nodes participating in a group, the less becomes the memory overhead added by communication CDD nodes. Also it can be seen that the higher the number of types handles by one computer node, the less becomes the relative memory overhead used on communication nodes, compared to "real" CDD nodes.

6.4.1 Distributed Shared Memory

A way to avoid the memory overhead introduced by the communication nodes is to use distributed shared memory. Then the node when receiving a state from Uppaal could chose either:

1. Do the *inclusion test* and *union* itself in the memory of all other nodes.
2. Or when sending messages, add the pointer to the CDD node on the receiving computer node, in the request.

But as we have chosen to distribute the CDD data structure using the MPI interface, to offer a more common interface to conform to the portability of Uppaal, we cannot avoid the need for communication nodes.

Summary

The cost of each of the four operations implemented is linear in the number of computer nodes participating in the operation, which cannot be done more optimal when a horizontal distributing approach has been chosen. Though the cost for verifying a single explored state takes the cost of $4k - 2$, but by disabling the *backtrace* operation the cost were reduced to $3k$ messages. This number of messages might introduce a considerable overhead in runtime. This is taken from the fact that analyzing a single state in Uppaal is a small task, and adding $3k$ messages for each state explored might multiply the verification time for each state. But as the purpose of this project is to allow verification of larger models and to measure memory overhead from distributing, we do not consider this result obstructing for further investigation.

Implementation

Chapter 7

In this chapter we describe what action has been taken during the implementation to decrease the amount of memory copied, and to optimize the runtime of the operations. Besides describing these actions, the interface to Uppaal is described.

7.1 Uppaal interface

This section describes the interface between Uppaal and the implemented CDD data structure made in this project. The overall functionality of Uppaal is depicted in the pseudo code of figure 1.3. To repeat:

- While the *Waiting* list is not empty
    - Take a state from the *Waiting* list, and search whether this states fulfills the property given.
    - Then check whether this state has been explored before
    - If not find all successors to the state and put these into the *Waiting* list.

The version of Uppaal we interface use a *Passed-*/*Waiting*List(PWList) interface. So to interface Uppaal this PWList interface has to be implemented. This interface consists of two functions with the following description:

tryPut(state *ps) Whenever Uppaal finds a new state it calls *tryPut*, which must examine whether the state '*ps*' is included in the *Passed* list. If the state is included the function returns immediately, without any action. If the state is not included, '*ps*' is inserted into the *Passed* list and also into the *Waiting* list for further exploration.

bool tryGet(state *ps) Whenever Uppaal has found all successors for a state, it calls *tryGet* to get a new state. *tryGet* then redirects the pointer given to an element in the *Waiting* list. If an element is found in the *Waiting* list the function must return `true`, and if no more elements are available in the *Waiting* list `false` must be returned to signal Uppaal that all states has been searched.

In our implementation the *Waiting* list is implemented as a FIFO linked list, therefore the *tryGet* call becomes simple. Simply redirect the '*ps*' pointer given to point to the first entry in the linked list.
The *tryPut* call becomes somewhat more complicated. The discrete state is hashed to a group participating in the

Before any of the *tryGet*/*tryPut* calls returns it is checked whether any incoming messages exists, being an *inclusion*/*union*/*reduction* request. If any incoming messages exists these are processed before returning.

If the *inclusion test* initiated succeeds somewhere, the state is simply discharged. If the *inclusion test* fails, two messages are send. First a *union* request is send to the top node of the local group, then the state itself is send to a computer node that will put this state in its *Waiting* list for further exploration.

7.2 Use Pipelining

Whenever a computer node receives a request for a CDD operation, either an *inclusion* request or a *union* request, the state need to be included. To avoid copying the state at each computer node, the following buffer management is implemented.
Whenever a message arrives, the respective operation is called keeping the state in the message buffer, then whenever the request need to be propagated to the next computer node, the same buffer is reused to send the request, so that the state need not be copied. In this way several memory copy operations are saved for each operation performed on the CDD data structure. This can be done as the *union* and *inclusion* operations are non destructive to the state.

7.3 Hash lists

Many of the operations in the implementation has to search for matching items. E.g. whenever a handle into the CDD data structure should be found, a list of discrete states with corresponding handles must be searched for a match to find the correct handle. And during the reduction phase whenever a node is changed, it is be examined whether there exist a node which is syntactic equivalent with the newly created node, so that this node can be reused/shared.
To reduce the complexity of the search for a matching item, both the list of discrete states, and all CDD nodes are inserted into a hash data structure. As more than one node might hash to the same bucket, bucket overflow is handled by adding the nodes to a linked list of nodes at each hash buckets.

This implies that either nodes cannot be changed, otherwise the node must be removed from the hash list. If is should be possible to remove an arbitrary element from a hash list, then the hash list should either be doubly linked - which waste memory. Otherwise the hash list has to be searched to find the previous element in the hash list, to redirect its '*next pointer*' to point to the next element in the list - which wastes time. Therefore it has been decided that is should not be possible to change a CDD-node (it is already not allowed to change the size without moving it, because of the segregated memory layout, see section 4.5).

7.4 Distributed Garbage Collection

In our first implemented approach, the '*ref_ count*' were updated at each *union* and *reduction* request, but this led to a very large runtime penalty for each operation performed, as the increasing of a '*ref_ count*' of one node, should result in an incrementation of the '*ref_ count*' of each of the successors to the node. This might result in an exponential number of nodes which should have their '*ref_ count*' incremented and some nodes '*ref_ count*' should even be incremented by more than once.
The reason that the '*ref_ count*', should be kept correct, has two purposes. First when the '*ref_ count*' is decremented to zero, no nodes references the node, and the node can be deleted. Secondly a node may not be changed if its '*ref_ count*' is larger than one, due to the *backtrace* proof. But as nodes is never changed anyway, due to the discussion in the previous section, the second purpose of the '*ref_ count*' is not an issue anymore.

This section describes a distributed garbage collection scheme which delete unused nodes in a more time efficient way, by only deleting nodes when space is needed.

7.4.1 Mark and Sweep

The chosen garbage collection schema chosen is a mark and sweep algorithm, modified to work in a distributed fashion. To help understand the algorithm refer to figure 7.1. First the '*ref_ count*' of all nodes not deleted yet (that is which '*used*'-flag is true), is set to 0. This is done by traversing all the linked list from each of the hash buckets. In the mark phase nodes referenced by a discrete handle are traversed recursively to update their '*ref_ count*' to 1. This is done by the following algorithm:

```
 1: void Mark(cdd_handle ∈ 𝒩)
 2: begin
 3:       if (cdd_handle.ref) = 0 then
 4:             cdd_handle.ref_count := 1
 5:             foreach (I, m) ∈ succ(cdd_handle)
 6:                   Mark(m)
18: end
```

This algorithm ensures that the '*ref_ count*' of all nodes referenced by a discrete handle is set to 1. Finally the *sweep* phase is initiated. It scans the linked lists for each hash bucket, and delete all nodes (by setting their '*used*'-flag to *false*) that has '*ref_ count* = 0', and therefore not referenced.

To distribute this algorithm, the *sweep* phase, notices which elements in the communication is used, and sends a message to the next computer node, with this information. Then the next computer node uses the used entries in its communication layer, as handles into the CDD data structure and deletes all elements not referenced by a used communication entry. That is the same algorithm is used, but the used entries in the communication layer is used as handles.

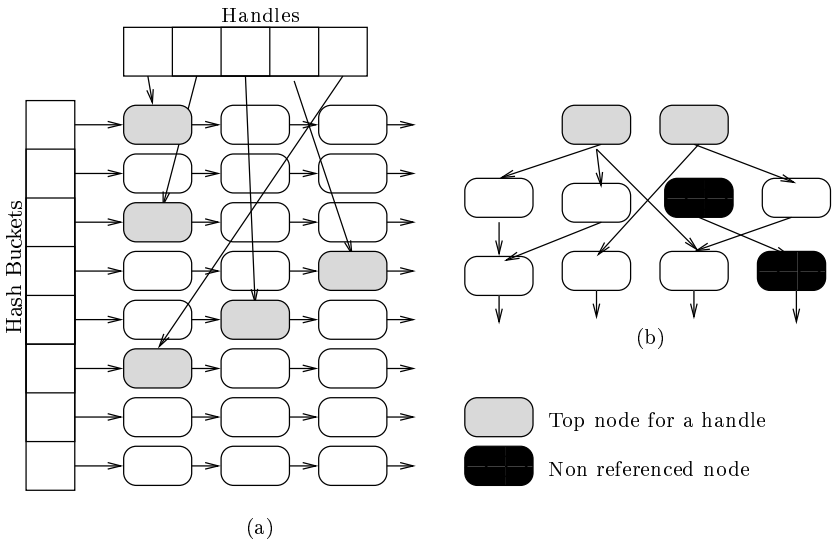


Figure 7.1: (a) show how all nodes is placed into a hash data structure, mentioned in the previous section, it also shown how the handles for the discrete states reference these CDD nodes. (b) Show a simple CDD data structure with two handles, and some unreferenced nodes.

Test

Chapter 8

A number of tests are conducted to test the performance of the implemented system. This chapter describes the tests, how the test setup is and our expected results. The actual results are presented with an analysis of these. Finally the analysis leads to a conclusion of the results archived. But first we consider what the consequences are for only distributing the symbolic part and not the discrete part of the states in Uppaal.

8.1 Limitations of the Implementation

The implementation was designed and implemented with the expectation, that it would be the symbolic part of the state space that would account for the most memory used in the verification. Therefore the discrete part was designed to be located on the computer node holding the handles into the CDD, as this gives faster access to the handle.

After our distributed implementation of the CDD data structure, and our preliminary testing begun, we found that the part of the *Passed* list that took up the most memory is the discrete part, which we had not focused on distributing. This disallows us to verify models that not already could be verified on a single computer node. In the *Future Work* section 10.2 we discuss how the discrete part can be distributed by modifying the current implementation. We are still able to test how large a memory reduction we obtain from using a single distributed CDD instead of several CDD's, one on each computer node, or in groups. The memory savings upon using the different node representations can also be tested.

8.2 Purpose of the test

The purpose of this project is as stated in section 2.2 to investigate how much memory can be saved using global sharing when storing the *Passed* list in verification of timed automata. Furthermore the relationship between synchronization overhead/memory usage when using a hybrid CDD model, where the number of CDD's are ranging between one and the number of computer nodes used, is to be tested

Furthermore we investigate how much memory can be saved using the alternative node representations described in section 4.5.1.

All tests should show the memory savings as well the encountered runtime penalty. When testing the distributed versions we also display the number of send messages, as this indicates the synchronization overhead.

The tests performed are as follows:

Memory representation The first test performed measure the memory savings vs. time penalty of the three node representations implemented.

Distribution The second test measures the time penalty introduced by distributing the data structure, together with the memory usage.

Groups The purpose of the third test is to show how much memory can be saved by global sharing compared to making groups. Furthermore it should show how large the time penalty is for the memory saved.

8.3 Premises

In this section we describe the premises under which the tests are conducted, both the hardware platform and the software used.

8.3.1 Hardware platform

Development and performance tests are conducted on a cluster of seven homogeneous dual 733 MHz Pentium III Coppermine workstations running on Asus CLS motherboards with ServerWorks LE chipset. The computer nodes are interconnected by a 100 Mbit Ethernet LAN, connected by a Cisco System Catalyst 3500 Series switch. Each computer node is equipped with 2 GB memory.
The software configuration were as follows: Debian GNU/Linux kernel 2.4.17, configured with SMP. The `gcc` compiler used were version 2.95.2.

8.3.2 Software premises

The used version of Uppaal is 3.3.20, interfaced using a combined *Passed-/Waiting* list.

For performance tests the following modes are used:

Dacapo which simulates the `Dacapo` protocol. It is the smallest model used, when verified on a single computer node, it explores 53967 states, in 5.65 seconds.

Buscoupler which is a simulation of bus coupling. When verified on an single computer node is explores 5288096 states, in 1701 seconds.

Fisher This is a model of the `Fischer` protocol, with 6 processes each trying to access the critical section. When verified on a single computer node, it explores 55674 states in 493 seconds.

The models used can be obtained by contacting Institute for Computer Science at Aalborg University.
Uppaal can be downloaded from `http://www.uppaal.com`.

Finding a model that it could be advantageous to distribute among several computer nodes is not possible as we only distribute the symbolic part of the *Passed* list, meaning the the memory usage of the computer node that holds all the discrete states becomes a bottleneck memory wise. Thus we will not be able to verify larger models as stated in the purpose.
The models is chosen, as these has been used for reference models in numerous Uppaal articles. The number of explored states documented in the previous list, is the number of states found by Uppaal and given to our interface through the *tryPut* call. That is, the number represent the number of *inclusion test* performed, whereas the number of *unions/reductions* is somewhat lower.

Tests on single computer nodes are deterministic and are thus only performed once.
Tests on multiple computer nodes are not deterministic, as two messages send from two different computer nodes to the same destination might arrive in any order. Therefore tests on multiple computer nodes are performed 5 times, and averaged.

Whenever tests is conducted on more than a single computer node the communication is done using the Message Passing Interface (MPI), the MPI interface used is LAM/MPI[1].

As distributing the verification adds some non-determinism, the number of verified states varies a little. In our tests the number of verified states varied less that 2%, and are documented.

Types Ordering

The chosen ordering of the variables can be important for the possible sharing in the CDD data structure, why the variable ordering is presented here. The chosen ordering is shown in a DBM, as the entry (X_i, X_j) and (X_j, X_i), where $i \neq j$ is represented by the same type, the entries is equal over the diagonal. Variables of the type $(X_j - X_j)$ should always be zero, and are thus never represented, and are therefore not given a type. The variable ordering chosen as follows:

	X_0	X_1	X_2	X_3	X_4	X_5
X_0	-	15	14	13	12	11
X_1	15	-	10	9	8	7
X_2	14	10	-	6	5	4
X_3	13	9	6	-	3	2
X_4	12	8	5	3	-	1
X_5	11	7	4	2	1	-

This variable ordering is chosen from the idea, that the greatest sharing is located where the condition is only based on the value of a single clock, and therefore the types of the form (X_j, X_0), is located at the bottom were sharing is possible. Further work could examine which variable ordering in the CDD data structure is best. The variable ordering is dependent on the model verified, but some guidelines could possible be stated, from such experiments. These experiments can be conducted on a single processor CDD implementation, as the results would map directly to a distributed implementation.

8.3.3 Measured data

Whenever the memory usage of our implementation is to be measured, only the memory usage in the CDD data structure is measured. The memory usage of the *Waiting* list, and the discrete state space is **not** measured, as these are not the focus in this project. When the memory usage is documented, the memory used when the verification is finished is measured.

Whenever the time is measured, the total runtime of Uppaal is measured in seconds. That is, the constant overhead of initialization/finalization is also measured, which is negligible.

The CPU-load is measured as the percentage of the run time that each computer node is active, and thus not blocking to receive a message, when its *Waiting* list is empty.

[1] www.lam-mpi.org

The memory load is presented as the percentage of the CDD that the specific computer node holds.

8.4 Test description

In the following subsections the test setup for each test are described. It is described what is measured, how many computer nodes participate in the test, and how many groups the test is performed with.

8.4.1 Node Representation

The first test performed is to measure how much memory can be saved using the alternative node representations described in section 4.5.2. This test is only performed on a single computer node, as it is assumed that the relative memory saving is only dependent on the model size. We test all three previously mentioned models. The results will be presented in a table showing the runtime and the memory usage for each node representation.

8.4.2 Distribution

To measure the overhead by distributing the CDD data structure, a series of tests is conducted, first on a single computer node, then on two, three and four computer nodes. The total memory usage is measured in each test to measure the memory overhead introduced by the communication layers.

The primary purpose of the tests is to measure the time overhead introduced by communication, and secondly to measure the load balancing between the different computer nodes. By load balancing we mean both processor utilization and memory utilization. The optimal result would be that each computer node always use 100% CPU time, and the memory usage being distribution evenly.

8.4.3 Groups

The primary purpose of this test, is to determine how much space savings that can be obtained using global sharing, and what runtime overhead is introduced by the communication. This test uses 4 computer nodes, first configured in four groups, then configured in two groups, and finally configured in a single group. The relative memory usage and space requirements are compared, with the total runtime for each configuration. This should allow us to measure the characteristics of using groups compared to a single distributed CDD.

8.5 Expected Results

This section is used to describe which expectations we have for the results, these expectations are used in the analysis of the results. These expectations are written before the actual tests are conducted, but after some preliminary tests were performed, which allow us to take communication overhead into considerations during the discussion of the expected results.

8.5.1 Node Representation

During our preliminary tests on some small models, our observations were that compared to the ordinary node representation the second node representation saved between 5% and 10%, the third node presentation saved between 15% and 20% of memory. We expect that the relative memory saving for the second node representation is somewhat constant as a function of the model size as the relative number of **false** intervals is expected to be constant. The third node representation might save even more memory as we expect that the nodes get more intervals and thus more integers to represent compared to the constant overhead, to the node given by the node header, which is 8 bytes. This representation is very dependent on the specific model, as the values of the integers is different between models.

The time penalty introduced for the second node representation is expected to be rather minimal, as only checks on bits is introduced. The time penalty for the third node representation is expected to be somewhat larger, as integers has to be packed/unpacked in all computer nodes participating in an *inclusion/union*. Additional the CDD nodes contains pointers that not necessarily is word aligned, thus compromising the portability as this does not work on most RISC architectures. Having pointers that are not word aligned means that the CPU has to fetch two words to get the pointer.

8.5.2 Distribution

The distribution of the data structure is expected to place a considerable overhead to the verification, as the number of messages send is large for each state verified. We also expect the top computer node to have a larger load than the others, mainly due to the problem of not distributing the discrete part of the state space. The cause is that the top computer node has to search for the correct handle, before initiating any operations on the CDD.

The memory usage is expected to increase slightly, as the communication nodes also takes up space.

Clearly the more types a computer node holds the fewer communication nodes exists compared to the number of CDD nodes in total - so distributing models with a large number of types is more feasible (memory wise) than distributing models with few types.
If models with a few types should be verified distributed, other methods may be used, or each computer node, could participate in a group for itself, which totally eliminated the need for communication nodes - but also disables global sharing.

The time complexity for the used algorithms are expected to range between the best and worst case complexity presented previously in chapter 6.

8.5.3 Groups

We expect that the effect of having global sharing will outrange the memory used for communication layers.
Compared to a single distributed CDD, the memory usage for more than one group is expected to be larger as more CDD data structures has to be constructed which cannot share states.

When using groups the runtime is expected to decrease with the number of computer nodes participating in the groups. This decrease in runtime is expected as the number of send messages

is reduced.

8.6 Results and Analysis

In the following section we present the test results and the analysis of these.

8.6.1 Node Representation

First we present the results for the three different node representations:

Dacapo:

Representation	Size in bytes	Time in sec	Size index	Time index
1	302.596	5.65	1.00	1.00
2	279.232	5.81	0.92	1.03
3	225.866	5.96	0.75	1.05

Buscoupler:

Representation	Size in bytes	Time in sec	Size index	Time index
1	12.602.672	1701	1.00	1.00
2	12.435.932	1803	0.98	1.06
3	10.193.851	2517	0.80	1.48

Fisher - 6:

Representation	Size in bytes	Time in sec	Size index	Time index
1	6.510.968	493	1.00	1.00
2	6.482.704	453	1.00	0.92
3	5.614.071	414	0.86	0.84

8.6.2 Analysis of Node Representation

In this subsection we analyze the results for the three different node representations.

Representation 1

This representation is the standard representation used by the current version of Uppaal, thus used for referencing the two other representations.

Representation 2

The results for the second representations, where edges to leading to **false** are omitted, shows only a modest memory savings for small models, whereas for larger models the savings are negligible. The increase in runtime using representation 2 is only 2%, but as the savings are negligible we conclude that there is no point in using this representation. Our expected results for this representation were that the memory savings would be constant as a function of the model size. This expectation was not correct, as it seems that for large models we obtain large

nodes as there is a lot of sharing, meaning that the relative number of intervals leading to **false** is reduced.

Representation 3

For all models tested this representation saves between 14% and 25%, smaller for larger models, but as stated in the expected results, will this representation be very dependent upon the specific model, as the clock values of this model influences greatly upon the memory usage. The increase in runtime is as expected greater than for the other representations being an increase of 50%. For the Fisher protocol this representation is surprisingly faster than the others, as the total number of nodes in the CDD is the same, the only explanation is as the nodes becomes smaller due to the compactness of the third representation, there will be lesser cache misses leading to increased performance.

8.6.3 Distribution

First we present the memory usage for each CDD-type in each of the models in a non distributed CDD, in figure 8.1, 8.2, and 8.3.

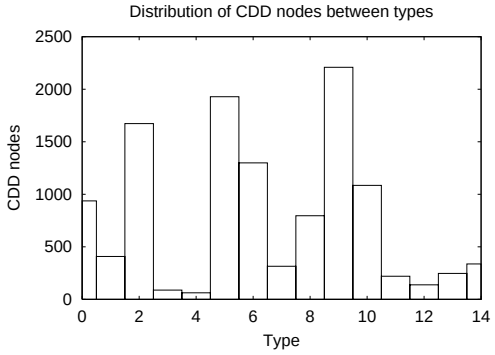


Figure 8.1: The distribution of CDD nodes between the different types of the `Dacapo` model

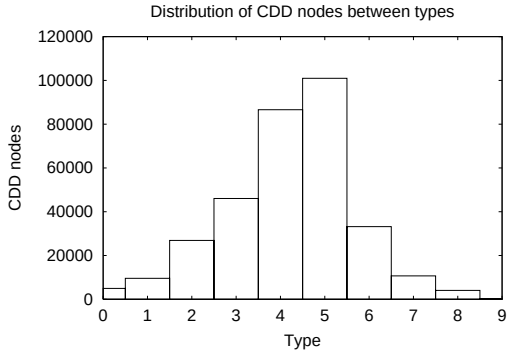


Figure 8.2: The distribution of CDD nodes between the different types of the `Buscoupler` model

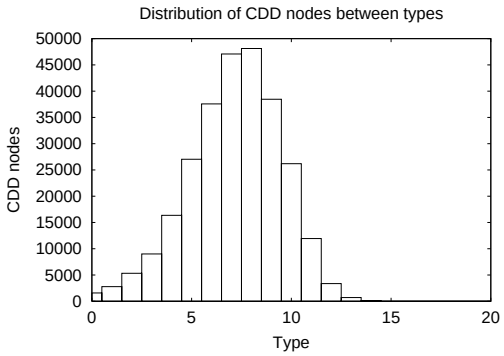


Figure 8.3: The distribution of CDD nodes between the different types of the `Fischer - 6` model

The results for the distributed tests are presented in tables, one table for one to four nodes, each model represented separately. Node 1 is always the top computer node.

The following 4 tables holds the results for verifying the `Dacapo` model on one to four nodes. For each node, the runtime, memory usage, CPU Load, Memory load, and the number of send messages is documented:

Dacapo one Node

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	5.65	302.508	100%	100%	0

Dacapo two Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	49.14	166888	100%	54%	96.117
Node 2	49.14	142164	60%	46%	55.861
Total	49.14	309.052	-	100%	151.978

Dacapo three Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	50.3	83880	100%	27%	121.308
Node 2	50.3	173972	66%	56%	97.880
Node 3	50.3	52812	40%	17%	61.523
Total	50.3	310.664	-	100%	280.711

Dacapo four Node

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	47.7	82360	100%	26%	115.087
Node 2	47.7	91864	65%	29%	94.919
Node 3	47.7	120372	37%	38%	89.373
Node 4	47.7	22172	26%	7%	44.190
Total	47.7	316.768	-	100%	343.569

The following four tables documents the results for the `Buscoupler` model, the same results as for the `Dacapo` model is presented:

Buscoupler 1 Node

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	1701	12.602.672	100%	100%	0

Buscoupler two Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	9532	7.210.612	100%	54%	12.623.045
Node 2	9532	6.142.372	61%	46%	6.354.126
Total	9532	13.352.984	-	100%	18.977.171

Buscoupler three Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	9652	1.746.032	100%	13%	14.623.045
Node 2	9652	9.670.352	64%	72%	10.388.128
Node 3	9652	2.014.660	35%	15%	5.764.121
Total	9652	13.431.044	-	100%	30.775.294

Buscoupler four Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	9572	538.184	100%	4%	16.523.948
Node 2	9572	3.094.560	62%	23%	12.312.429
Node 3	9572	9.149.140	45%	68%	7.354.264
Node 4	9572	672.732	26%	5%	4.723.539
Total	9572	13.454.616		100%	40923180

The final four tables in this section document the results archived when verifying the `Fischer` model on one to four computer nodes:

Fischer 1 Node

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	493	6.510.968	100%	100%	0

Fischer 2 Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	4183	5.483.360	100%	81%	187.221
Node 2	4183	1.286.220	63%	19%	70.538
Total	4183	6.769.580	-	100%	257.759

Fischer 3 Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	4213	3.015.220	100%	44%	187.678
Node 2	4213	3.769.032	71%	55%	135.287
Node 3	4213	68.528	12%	1%	72.846
Total	4213	6.852.780	-	100%	395.811

Fischer 4 Nodes

Node	Time Usage	Memory Usage	CPU Load	Memory Load	Send Messages
Node 1	4451	1.041.532	100%	15%	174.606
Node 2	4451	4.513.312	72%	65%	137.584
Node 3	4451	1.319.280	23%	19%	138.964
Node 4	4451	69.436	4%	1%	60.527
Total	4451	6.943.560	-	100%	511.681

8.6.4 Analysis of the Distribution Results

In the following we discuss the results archived during the tests of the distributed data structure. The main focus is based on CPU/Memory load, and time penalty and memory overhead. Finally the number of messages send, is compared to the cost-benefit analysis.

Time Penalty

The time penalty added by distributing the data structure, were almost unaffected by the number of nodes used. We expect that the explanation for this is that the maximum number of messages send by a single computer node (the top node) only increase slightly. Besides that more computer nodes participating in the exploration of states. The time penalty were between 560% and 900% which is an acceptable cost as the focus in this project were to save memory at a certain runtime penalty. The memory savings will be discussed later in section 8.6.6.
Besides the implementation is not optimized for runtime, only memory wise. E.g. it would be possible to lower the total runtime by packing several requests in the same message. This would decrease the total number of messages send, which we expect to account for some of the time penalty. How such requests should be packed is discussed in the *Future Work*, section 10.4.

Memory Overhead

The memory overhead added by distributing the data structure were between 4.3% and 6.7%. Models with larger symbolic state space often have a larger number of clocks and therefore the relative memory overhead is decreased. We find the memory overhead found acceptable, as the number of clocks in the used models is relatively small. This overhead is not expected only to be due to the communication layer, but also that the CDD might be build differently, as the distribution adds some non-determinism to which order states are unioned with the CDD, as CDD's are not canonical - see section 5.7.1.
Additionally this memory overhead could be reduced to a minimum, by implementing the distribution in a shared memory environment as discussed in section 6.4.1, a protocol for this is discussed in 10.6

CPU/Memory Load

During the tests the amount of used memory on each computer node, used for the CDD data structure were measured. The memory load were not evenly distributed, the reason is that the number of CDD nodes in each layer is not evenly distributed. This can be seen from the box diagrams in figure 8.1, 8.2, and 8.3.

This call for dynamic memory load balancing, which we discuss in *Future Work*, section 10.5. The CPU load is also not evenly distributed. The top node is always the most loaded node, this is due to the fact that it is this node that has to find the handle into the CDD data structure. We expect that the reason that the CPU load decrease in the lower computer nodes, is that *inclusion tests* may fail earlier, and thus not burden the lower computer nodes.

Average Complexity

Previously we have stated that the worst case complexity is $3k$ messages for each state explored with k being the number of nodes participating, whereas the best case complexity for each state is $k-1$ messages. First we note that the top node always is the node that account for the largest number of send messages. The reason for this is that the top node is responsible for sending states that has failed the *inclusion test* to a node - which then put it into its *Waiting* list. This might seem as a bottleneck but with the current version where the discrete part is not distributed, then this is the only possible approach as the top node holds the discrete part which also has to be stored in the *Waiting* list. Alternatively the discrete part could be included in all messages introducing an overhead in the message size.

The tests shows an average complexity in the number of send messages per verified state between $1.5k$ for `Dacapo` and $2.3k$ for `Fischer`, `Buscoupler` being in the middle with a complexity of $1.93k$. These results is as we expected between the best case and worst case complexity. `Dacapo` has the lowest complexity which we expect is due to an observation that almost all *inclusion tests* that failed, did fail on the top computer node, leading to a limited number of *inclusion* requests send to other computer nodes. This together with an observation of the *reduction* only reaching the top computer node in a limited number of *reductions* gives the complexity of 1.5. `Fischer` has the highest complexity as the opposite case here is the case being that almost all *inclusion tests* fails on another computer node than the top computer node. The ordering of the types could influence on where the *inclusion test* fails, why such a reordering may change the average complexity.

8.6.5 Groups

The results for using groups is presented in the tables below, one for each model with time, send messages, and memory usage. The number of used computer nodes is four.

Dacapo:

Nb of groups	Time in sec	Total used memory in bytes	Total nb of sent messages
1	37.7	316.772	343.569
2	25.2	554.352	171.548
4	8.2	904.212	34.720

Buscoupler:

Nb of groups	Time in sec	Total used memory in bytes	Total nb of sent messages
1	9562	13.454.616	40.923.180
2	5570	22.573.832	22.703.471
4	1620	45.033.084	3.365.242

Fischer:

Nb of groups	Time in sec	Total used memory in bytes	Total nb of sent messages
1	4451	6.943.560	511.681
2	2721	10.634.604	276.864
4	834	21.420.432	36.758

8.6.6 Analysis of the Group Results

In this section we discuss the results archived from the tests using groups, where there are more than one CDD to represent the *Passed* list, these CDD's may be located on a single computer node or span several computer nodes.

Memory Usage

The memory usage increases as expected when using more than one CDD. The maximum would be that each of the new CDD's takes just as much space up as the single distributed CDD. Thus for the test results the extra memory usage for using 4 groups instead of one can maximum be of a factor four. The tests report the extra memory usage for the three models to be between 2.9 and 3.3 times as much memory for using four groups instead of a single group. This is surprising to see that the sharing between different handles in the CDD is so large. The test using two groups instead of a single group reports an increased memory usage of between 1.5 and 1.75 times as much memory as the single group version. The increased runtime seems to be linearly dependent upon the number of groups participating, the less computer nodes in each group the lower the runtime is.

The memory usage when using groups is larger than expected, thus it might not be an attracting alternative to use groups instead of a single distributed CDD, as the memory usage cannot be estimated before the actual verification has been carried out, and therefore the optimum number of groups cannot be estimated, which might lead to a non terminating verification if it runs out of memory.

Using the CDD data structure in a distributed environment might waste memory if each computer node holds a separate CDD, as the sum of the memory consumption of these CDD's will account for more memory than a single CDD.

Time Penalty

The time penalty seems to be linearly dependent on the number of groups used in storing the CDD, but we cannot conclude this linearity as we only conduct tests with a limited number of computer nodes. The more nodes participating in the group the larger the time penalty. We had expected the difference in runtime of two and one group to be larger, but from the results we conclude that one group is best for large models where memory usage is critical. If memory usage is non critical, we recommend using the DBM data structure, as the CDD data structure adds a runtime overhead, as reported in [4].

8.6.7 Additional results

[4] report that the inclusion test using CDD's is better than the inclusion test using DBM's as CDD's describe *federations* whereas DBM's describe *Zones*. We have made the same observation, e.g. for the `Buscoupler` model - using the DBM structure requires exploration of 7.2 million states. When using our implementation of the CDD data structure the model could be verified in 5.3 million explorations.
This improvement is conducted even though the DBM implementation of Uppaal used the priority *Waiting* list, and we only implemented the *Waiting* list as a linked list.

To see that the implementation of the *Waiting* list as a priority queue has advantages over the linked list implementation, can be seen from the verification of the of the `Dacapo` model. Here the DBM Uppaal version with its priority *Waiting* queue verified the model in 45005 state explorations, where our implementation explores 53957 states.

8.7 Summary

This section summarizes the results archived during the tests, and repeat the main conclusion.
For the node representations up to 20% of memory were saved, at a maximum runtime cost of 50%, which we consider acceptable as the runtime were not considered during the design of these node representations. Memory representation 3 is only designed for running on non RISC architectures, as pointer may not be word aligned, but this could be extended to run on RISC architectures at a certain runtime cost. This runtime cost is expected as the pointer converting has to be performed in software.
The thesis we tried to test were whether memory could be saved by taking advantage of global sharing. The results archived during the group test, showed that nearly the same symbolic states has to be explored for all discrete states, which lead to a memory saving of 70% when building a single CDD compared to building 4 separate CDD's. But the runtime penalty introduced were up to 900%.
In the test of the distributed CDD we observed that the memory overhead introduced by communication nodes, and building the CDD differently were acceptable, ranging from 4.3% to 6.7%. The runtime penalty introduced by the distribution, were constant to the number of computer nodes participating in the verification.
The worst case complexity for each verified state were calculated to $3k$ messages, k being the number of computer node participating. The measured complexity were between $1.5k$ and $2.3k$, which might indicate that the *inclusion test* often failed on the upper computer nodes, or that a large number of *inclusion tests* succeeded.
The memory were not evenly distributed, which indicate the need for a dynamic memory load balancing. The CPU load were affected by this non even distribution of memory, but mostly from the fact that the top computer node holds all discrete states, as the number of discrete states is large, resulting in a large runtime penalty for searching for the CDD handle.

Conclusion

Chapter 9

In the conclusion we analyze the results archived, which knowledge can be used, and which improvements are possible to archive further memory savings.

Although it were discovered that the problem of storing the *Passed* list in Uppaal were not storing the symbolic part of the states, the results archived are satisfying, as purpose were to investigate how much memory could be saved by global sharing.
The global sharing archived were almost linear to the number of computer nodes used for the CDD data structure, the argument for this has to be that for almost all discrete states, the same symbolic states are explored. This fact, could be used to make an even more compact representation of the *Passed* list, were the discrete and the symbolic part of the *Passed* list is combined in the same data structure.
The memory overhead added by the communication layers was between 4.4% and 6.8% using four computer nodes which seems reasonable. The relative memory savings for a single distributed CDD compared to four groups on four machines, was up to 70%.
The runtime penalty introduced were between 560% and 900%, which is acceptable taken into considerations that we have not optimized with respect to runtime. If the distributed data structure were redesigned to also take runtime optimizations into account, the runtime penalty might be reduced, and if the great memory savings could be kept, the data structure might prove interesting. Possible runtime optimization is discussed in section 10.4.

Although the archived results might not be used for the current Uppaal version, as the runtime penalty added is to large and models with large symbolic parts does not exist. Some of the ideas might be used in the design of other distributed DD data structures. Some of the ideas might also be used to design a distributed decision diagram(DD) data structure for storing the discrete part of the *Passed* list. If the sharing in such a distributed data structure could be as great as for the symbolic part, designing such a DD data structure could allow the verification of very large models. But before such an algorithm can be implemented a throughout design of the influences of combining the discrete and symbolic part of the *Passed* list in the same data structure has to be investigated. The possibly sharing in the discrete part also has to be investigated. In the next chapter we give an example on how the discrete part could be stored in a DD data structure.

The alternative node representations designed and implemented showed that the used memory usage could be reduced up to 20%, at an acceptable cost in runtime being at most 50%. If the design of the node representations were designed to optimize runtime as well as memory usage, the runtime might decrease.

As this project makes a contribution to computer aided verification, the data structures and algorithms designed must be semantically sound and complete, why we have conducted several semantic proofs. First we proved that adding the communication layer to the CDD does not change the semantics of the CDD. This were used to simplify the proofs of the distributed algorithms, as it made a direct mapping between the non distributed algorithms and the distributed. The semantic proof eased the design of the distributed algorithms, which are very similar to the non distributed. The only exception is the *union* algorithm, which had to be extended with a *reduction* algorithm - as reduction can only be performed bottom up, and union must be performed top down.

The *backtrace* operation designed in section 4.4.4, were not implemented, and thus not tested. The reason for not implementing this operation were that the *backtrace* algorithm worst case

added $k-1$ messages to the number of messages needed to explore a single state in Uppaal, which we considered too expensive, for further arguments refer to section 6.2. The results showing that the sharing possible from global sharing being so large, the *backtrace* might prove worse, as the number of CDD nodes having only one successor might be very small, this can also be seen from the diagrams showing the distribution of CDD nodes for each type, as there is a relatively small number of these nodes.
Even though it might prove that the *backtrace* algorithm could reduce the runtime on some kinds of models, which could be investigated. This investigation could easily be conducted on a single processor version of the CDD data structure, as the results would map to the distributed implementation.

The group test showed that the size of each of the CDD's located in groups almost had the same size as the single CDD. This indicates that is advantageous to use a single CDD, as the size of several CDD's using groups is almost as large as single CDD times the number of groups.

The complexity of the distributing the CDD, indicated that is it not advantageous to distribute them, if they can be located on a single node. As all the timed automata model we have encountered used most memory representing the discrete part. An approach to improve both the runtime and memory usage, would be to locate the symbolic state space on a single computer node, and distribute the discrete part. This would allow global sharing between all discrete states, and reduce communication overhead.

If a model with a symbolic state space that cannot be located on a single computer node, the distribution approach described in this project might prove useful. Although this approach does not scale to more computer nodes than the number of types in the CDD.

In this chapter we discuss which improvements to the existing implementation can be made, to archive further memory/runtime savings.

10.1 CDD Implementation of Waiting List

In the current implementation only the symbolic part of the *Passed* list is represented as a CDD. We have not conducted tests showing how much memory is saved by using CDD's compared to the use of DBM's for storing the *Passed* list, as mentioned in the purpose. A single processor implementations of Uppaal based on the CDD data structure have shown 42% memory savings compared to storing the *Passed* list as shortest path reduced DBM's. Therefore it is not unlikely that storing the *Waiting* list as a CDD also could save some memory in the representation of the *Waiting* list. Besides saving memory in the representation another advantage could be archived: When Uppaal takes a state out of the *Waiting* list, it expects a DBM, which has been shown to be equal to a S-CDD. If the two S-CDD's in figure 10.1 is inserted into the *Waiting* list in the shown order, only the S-CDD of figure 10.1(b) will be represented and thus verified by Uppaal, this might lead to a faster termination of Uppaal, as fewer states might be examined.

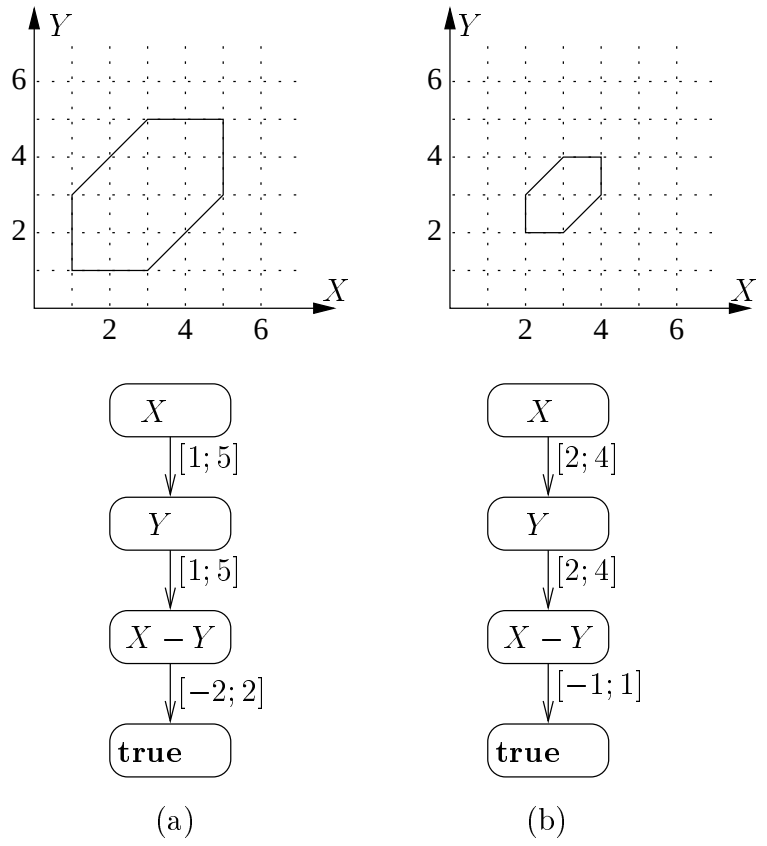


Figure 10.1: Example on how representing the *Waiting* list as a CDD might lead to fewer examined states for Uppaal

In the current implementation of Uppaal the *Waiting* list is implemented as a priority queue where states with a larger *Zone* are verified first, followed by states with smaller *Zones*, this leads to the same reduction as for the *Waiting* list implemented as CDD, only in the DBM priority queue elements must be inserted correctly leading to a linear search, when using CDD for storing the *Waiting* list this priority queue could be implemented at a lower runtime cost.

10.2 Distributing the Discrete part

The current implementation of the distributed CDD does not distribute the discrete part of the state space, as stated previously. This section comes with some guidelines for how such a distribution could be done using the current implementation. The current implementation stores all discrete states associated with handles into the CDD on the top computer node. The problem with distributing the discrete part is that we need it to find the handle into the CDD, as the symbolic state must be unioned with the handle matching the discrete part.

A possible distribution approach is described in the following algorithm:

- A computer node receives a state from Uppaal
- It finds the computer node responsible for this discrete part via a hash algorithm, and sends the state to this node.
- The computer node responsible for the discrete part receives the state and uses the discrete part to find a handle into the CDD, and sends the symbolic part along with the handle, and a machine ID to the top node. Note that this computer node has to synchronize with the top computer node on the handle, when it receives a discrete part it has not seen before.
- The top computer node receives the symbolic state along with the handle which it uses to find the node for which it must perform an *inclusion test* of the state. The *inclusion test* is performed distributed as in the current implementation.
- If the *inclusion test* succeeds no further action has to be taken, if it fails a *union* request is send to the top node, and the state is send to the node responsible for the state using the machine ID.
- The node responsible for the state receives the state that failed the *inclusion test* and inserts it in its *Waiting* list for further exploration.

The synchronization on the handle between the computer node responsible for the discrete part and the top node, can be performed by letting the node holding the discrete part send a unique id that it generates itself, meaning that each time the top computer node receives a state it can find the handle from the unique id. This will work as all nodes holds separate discrete parts, meaning that no synchronization on handles need to be performed between other nodes.

This way to distribute the discrete part will add a complexity of sending one additional message, leading to a new complexity of $3k + 1$ instead of $3k$. We do not expect this extra message to increase the overall runtime much, but additional test needs to be performed to verify this thesis. The memory used to store the discrete part will in this way be evenly distributed, given a uniform hash algorithm.

10.3 Representing the Discrete Part as MTIDD

During this project we realized that the problem of storing the *Passed* list in formal verification of timed automata, is not storing the symbolic part, but storing the discrete part, we have thought of a way to storing the discrete part of the *Passed* list. The discrete part of the *Passed* list is in this project implemented in a hash list, that is a number of hash buckets, each containing a linked list of discrete states, with each discrete state a pointer is provided, which is a pointer to the handle in the CDD data structure which should be used.
In this section we describe how the discrete part of the *Passed* list possible could be implemented using a Multi Terminal Integer Decision Diagram (MTIDD). , which possible could save both memory as well as reduce the runtime.

We first describe how the discrete part could be stored using a MTIDD, then we argue why this possibly could save memory, and finally we argue how this method of storing the discrete part could save some time.

10.3.1 Representation

The discrete part of an Uppaal state consists of two parts:

Location A timed automata model, consists of a number of processes possibly synchronizing with each other. Each of these processes can be in a number of different locations. E.g. if a process simulates a train gate, it might have three states: *open*, *closed*, and *active*. The location part of the discrete part consists of a vector designating which location each of the processes is currently in. In the current Uppaal implementation each processes is allowed to have 64K different locations. This information is stored as an array of integers.

Variables Besides locations, synchronizations channels, and clocks, timed automata models are also allowed to hold integer variables. The second part of the discrete state in Uppaal states is an array holding the value of each of these integer variables.

The discrete part is currently stored as an array of integers. For each location vector, the number of different variable assignments may be very large. In the current implementation the location vector is stored once for each variable assignment, this redundant information could be deleted representing the discrete part as a MTIDD.
That is the only information needed to be stored is an array of integers. For this purpose a MTIDD could be used, then each layer in the MTIDD, represent one entry in the array representing a location or an integer. Each node holds a list of assignments to the location/variable and a pointer to follow if this assignment is true. When the lowest layer has been reached, the bottom node holds a handle into the CDD data structure. An example on a MTIDD data structure holding the information:

- $\{1, 1, 1, 45, 46\} \rightsquigarrow handle_1$
- $\{1, 1, 2, 45, 46\} \rightsquigarrow handle_2$
- $\{1, 1, 1, 45, 47\} \rightsquigarrow handle_3$
- $\{1, 1, 1, 44, 46\} \rightsquigarrow handle_4$

- $\{2, 1, 1, 45, 46\} \rightsquigarrow handle_5$
- $\{1, 1, 1, 45, 48\} \rightsquigarrow handle_6$

Is presented in figure 10.2(a)

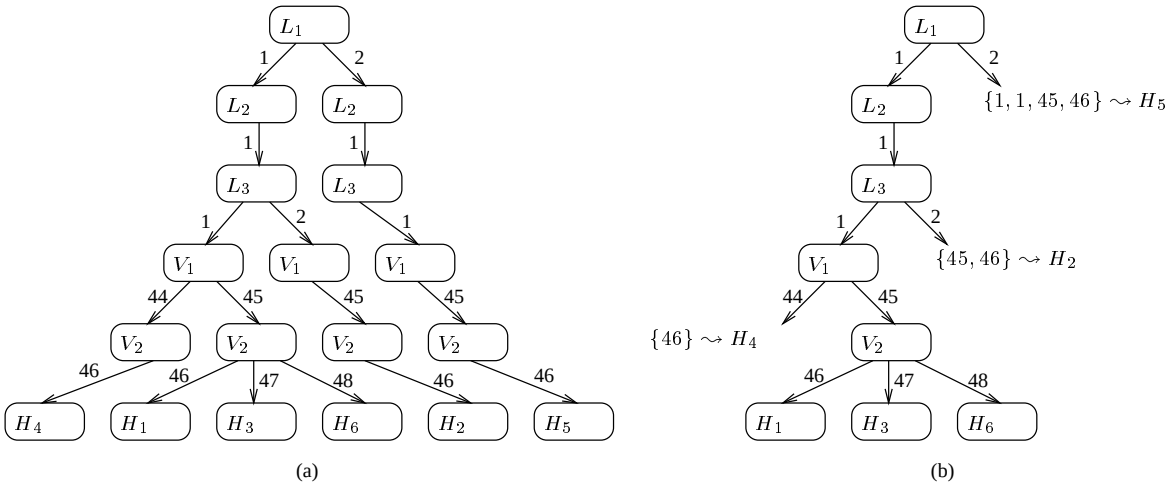


Figure 10.2: (a) Multi Terminal Integer Decision Diagram, used for representing the discrete part of the *Passed* list of Uppaal. (b) MTIDD representation of the discrete part using optimizations, using tagged architecture. In the figure L_1, L_2, and L_3 designates locations, V_1 and V_2 designated integer variables, and $H_1 \ldots H_6$ designated handles into a CDD data structure.

10.3.2 Saving Memory

The reason that it is expected that the MTIDD data structure might save memory in the representation of the discrete part of the *Passed* list, is the same as the reason that other XDD data structures save memory, namely by only storing information once. But where most XDD data structures share data in the bottom elements, the MTIDD data structure used is only allowed to share information at the top of the data structure. E.g. the representation of the first three integers as $\{1, 1, 1\}$ is only stored once, although used for four handles. One disadvantage of this representation is that a certain overhead is added as each node need to keep additional information. This problem occurs when the MTIDD holds a subpart of only a single string, then this representation might double the memory usage, as it can be compared to represent an array as a linked list. If when constructing a sub MTIDD with only a single terminal node, an array is inserted with the values of the remaining variables and a handle, some memory could be saved. This memory optimization is showed in figure 10.2, to known whether the successor is a MTIDD node or an array, a tagged architecture can be used, as the two least significant bits are not used in pointers, these can be used to represent whether the child following the pointer is a MTIDD node or an integer array. It would even be possible to use our alternative node representation 3 to use a minimum number of bytes to store variables, which possibly could lead to further memory saving.

10.3.3 Saving Time

The reason it is believed that representing the discrete part in a MTIDD might save some time, is that the amount of memory compared is expected to decrease greatly. E.g. for verifying *Buscoupler*, which is a relatively small model, over 1.5 million discrete states of size 160 bytes exists. The *Buscoupler* model holds 16 processes so the location vector consists of 16 (16bit) integers, and the model holds 27 (32 bit) integer variables. That is the MTIDD holds $16+27 = 43$ layers, and only 43 MTIDD nodes, of different size, has to be traversed.
In the current implementation using a hashed linked list, the hash data structure holds 17609 buckets. Then in the worst case $\frac{1.5*10^6}{17609} = 85$ discrete elements of size 160 bytes need be searched. Therefore the amount of memory comparison needed using the MTIDD data structure might easily be somewhat smaller than the amount of memory comparison needed for the current implementation, and it is expected to be even better for larger models.
If the size of the MTIDD nodes is large, it takes more runtime to find the handle, but a greater sharing is accomplished, and if the MTIDD nodes is small the time for finding the handle is small, but so is the sharing. That is either is runtime saved compared to the ordinary implementation, or else memory is saved.

10.3.4 Distributing

As the discrete part of the *Passed* list is the part that takes up the main part of the memory usage, it would be interesting to distribute this MTIDD data structure, if it makes a more compact representation of the discrete part. Figure 10.2(a) show that the representation takes up the main part of the memory in the bottom part of the tree. This means that using a horizontal distribution approach is not suitable, the vertical distribution approach, or the groups approach described in section 4.2.2 on page 31 might show to be more appropriate as traces never merge, therefore the maximum number of messages send can be reduced to less than the number of computer nodes used.

10.4 Pack Messages

In the current implementation, some of the time penalty introduced is expected to come from the large number of messages send. In this section we describe how this number of messages can be decreased without changing the functionality/semantics of the implemented system. The idea is taken from the current distributed Uppaal version, which pack a number of states before sending them to another computer node[3]. That is, each computer node holds a buffer for each of the other computer nodes, and whenever a messages should be send to another computer node, it is placed in the corresponding buffer and when a certain number of messages has been added to a buffer it is send. The same approach might be used in the distribution of the CDD. Each computer node only propagates requests to the lower computer node (for *inclusion test* and *union*), and to the upper computer node (for *reduction*). Therefore, for each request type a single buffer is kept, and send whenever it is full. To optimize this scheme a little *union* and *reduction* request buffers, should be send before *inclusion test* requests, which would possible decrease the number of *inclusion test* which fail, and thereby reduce the number of explored states.

10.5 CPU/Memory Load

Another problem our implementation suffers from, is that the load distribution both with respect to CPU usage and memory usage is far from even. To solve this two approaches has to be implemented, first and most important, the discrete part of the *Passed* list should be distributed, and secondly dynamic memory load sharing should be implemented.
An approach for distributing the discrete part of the *Passed* list has been discussed is the previous. For implementing dynamic memory load sharing, it should be possible to move a layer of the CDD data structure from one computer node to another, and still keeping all successors correct, to be able to implement this, a design and possible a semantic analysis has to be performed.

10.6 Distributed Shared Memory

In the cost benefit analysis we argue that the memory overhead introduced by the communication nodes could be removed by using distributed shared memory.
In the last two project we have investigated the capabilities of the Scalable Coherent Interface (SCI) network technology. The SCI technology offers a hardware based distributed shared memory environment. The following discussion is based of the use of SCI for distributing the CDD data structure.
The distributed shared memory interface offered by SCI, has very low latency, and very high bandwidth (100-200MB/s) in writes, for reads the bandwidth is lowered to (4-5MB/s). Therefore the protocol designed should be based on writes and not reads. Of the two ideas mentioned in the cost benefit analysis section 6.4.1, the idea where the global pointers is written to the next node, which is then signaled to continue the request, might be the best approach, when SCI is used. If the node discovering a state should make the *inclusion test* and the *union* itself, the operations should fetch large amounts of memory from the other nodes, at the low speed. Therefore the same ideas as in this project should be used, only should the communication protocol be altered, and the communication layers could be omitted.

Bibliography

[1] Alur and Dill. Automata for modelling real-time systems. *Lecture Notes in Computer Science, LCNS 443*, 1990.

[2] R. Stallman B. Lewis, D. Laliberte and the GNU Manual Group. *GNU Emacs Lisp Reference Manual.*, 1995.

[3] Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Juli 2000. Springer-Verlag.

[4] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification, LCNS 1633*, pages 341–353, 1999.

[5] R. Bellmann. Dynamic programming. *Princeton University Press*, 1957.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Cimputers, 8(C-35):677-691*, 1986.

[7] W. T. Comfort. Multiword list items. *Communications of the ACM,7(6)*, June 1964.

[8] Kim G. Larsen et al. Clock difference diagrams. *BRICS Report Series publications*, 1998.

[9] Paul Pettersson Kim G. Larsen and Wang Yi. Uppaal: Status & develoments. -, 1997.

[10] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction, 1997.

[11] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[12] Amit Narayan, Jawahar Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned robdds;a compact, canonical and efficiently manipulable representation for boolean functions. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 547–554. IEEE Computer Society Press, 1996.

[13] Thies Rauhe Niels Klarlund. Bdd algorithms and cache misses. *BRICS Report Series publications*, 1996.

[14] Karsten Strehl and Lothar Thiele. Symbolic model checking using interval diagram techniques. *Technical Report 40, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich.*, 1998.

[15] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[16] Sergio Yovine. Kronos: a verification tool for real-time systems. -, 1997.

Union/Reduction Example

Appendix A

This appendix gives an example on first a *union* operation building a *call stack*, and using a *temporarily hash* table for a more efficient construction. After the *union* has constructed the union of a S-CDD and a CDD, a *reduction* is performed. The example run on two nodes, and whenever a message is send from one computer node to another the message content is also shown.

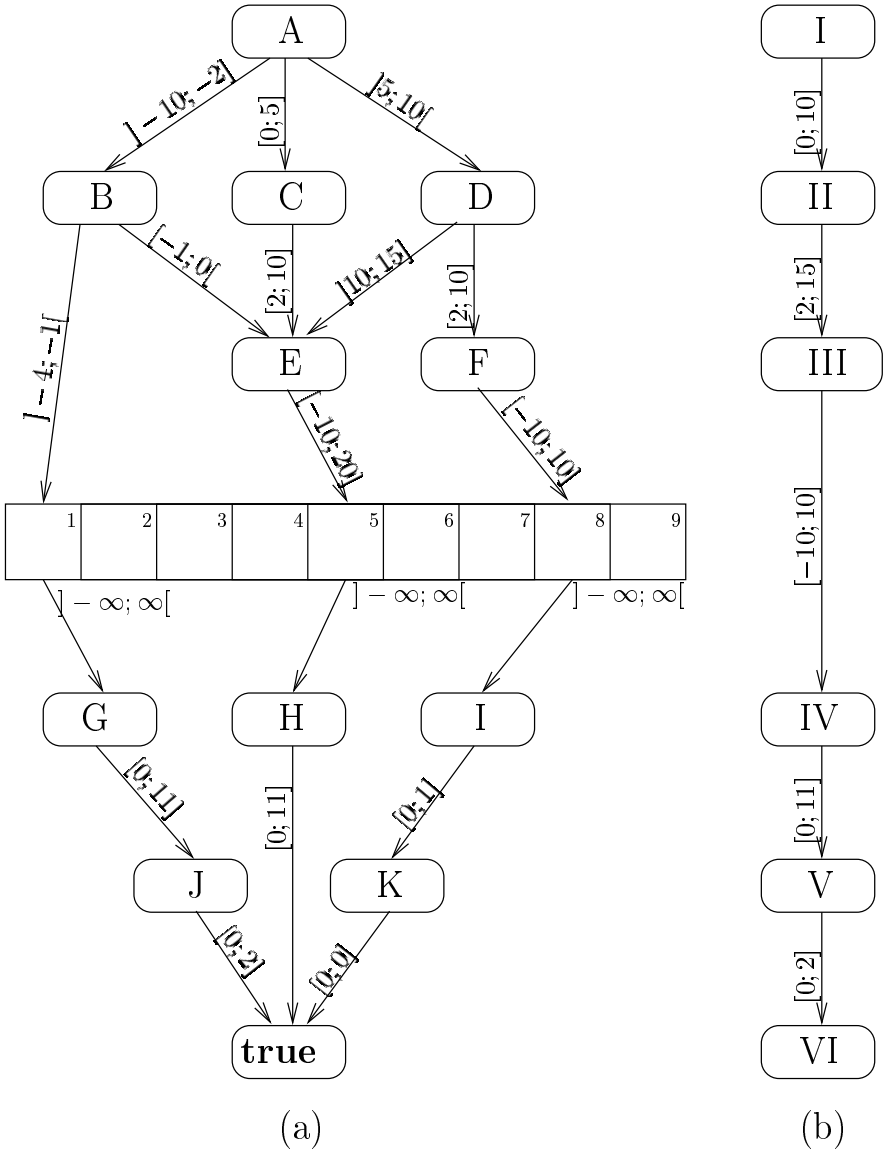
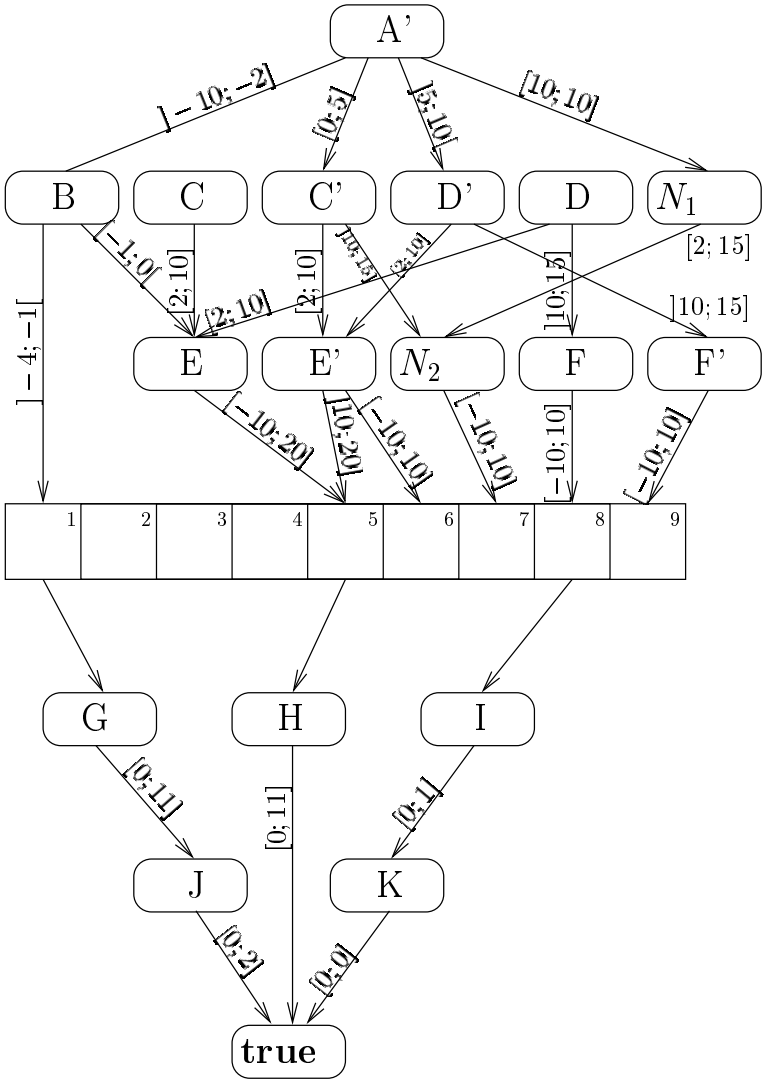


Figure A.1: The two *X*DD data structures that are going to be unioned in this example.

Figure A.1 show a CDD and an S-CDD which are going to be unioned, and thereafter reduced. The *backtrace* algorithm is not used, at this makes it easier to illustrate the *reduction* operation. During the operations it is assumed that all CDD nodes is referred by other CDD nodes not shown, so that these nodes **cannot** be deleted.
Figure A.2 show the situation after the *union* has finished its operations at the top node. To



Union Request = $\{(6,5),(7,\textbf{false}),(9,8)\}$
Call stack = A',C',D',N_1,E',N_2,F'
Temp hash = $\{(A \cup I = A'),(C \cup II = C'),(D \cup II = D'),(\textbf{false} \cup II = N_1),$
$(\textbf{false} \cup III = N_2),(\textbf{false} \cup III = N_2),(E \cup III = E'),(F \cup III = F')\}$

Figure A.2: Show the example after the union has finished on node one, and send a *union request* to the next node.

the right of the CDD, the *call stack* and the *temporary hash* table is shown. The idea is that for all nodes which need to be changed is copied (and in the example renamed from X to X'), and this copy is then changed to satisfy the requirements. To see the function of the *temporary hash* table, note that the only successor to node N_1 is the union between **false** and III. The successor of C', with interval $]10;15]$ is also the union between **false** and III, thus the node N_2 is reused, without repeating the union. Node E' is also reused by using the *temporary hash* table.
The message send to the bottom computer node is the following: $\{(6,5),(7,NULL),(9,8)\}$, has

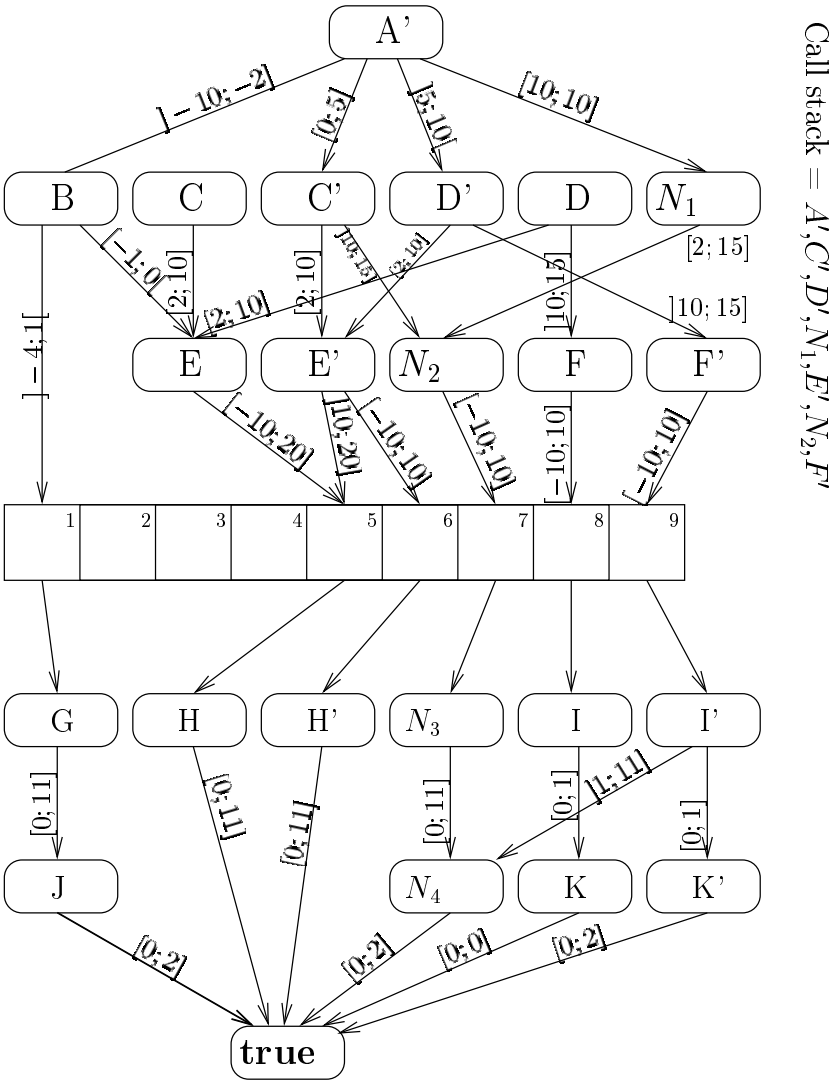


Figure A.3: Show the situation after the *union* on the bottom node, has reached the bottom in it's recursion, and just before it recursion starts build CDD nodes from bottom up.

the following meaning. Union S-CDD (type IV^1) with the CDD-node referred to by communication array entry **5**, and place the result in communication array entry **6**. The $(7,NULL)$, means union S-CDD (type IV) with **false**, and let communication array entry **7** refer to it.
Figure A.3 show the situation after the bottom node has performed it's union, but before it start recognizing that there is a possible sharing. (This situation does actually never exist, except on the call stack, but is shown as it gives a better understanding of the situation).

[1]First type on next node

Whenever the bottom node builds a node in its *union* operation, it checks whether there exist

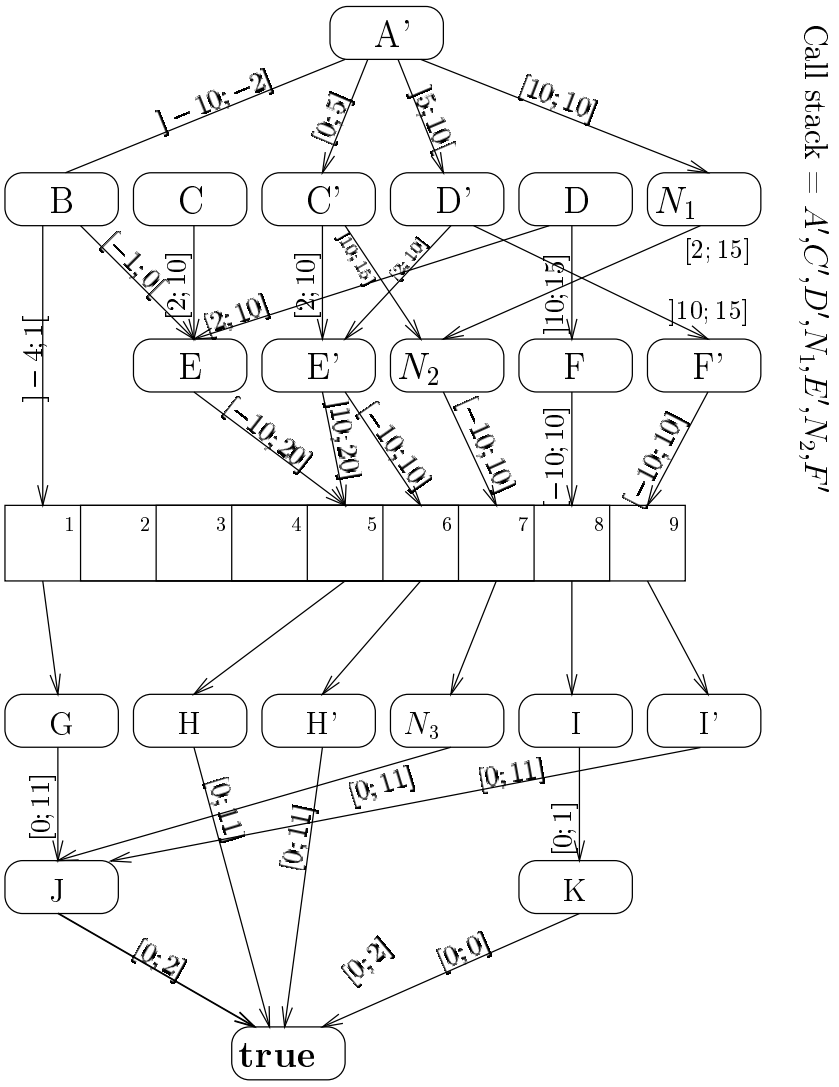


Figure A.4: An intermediate situation during the *reduction* on the bottom node

an equivalent CDD node, and if it does it use this node instead.
E.g. whenever the nodes K', and N_4 is created, the runtime system recognizes that these nodes are equal to the already existing node J. N_4 and K' is thus never created, but all references to these nodes, is redirected to J. This result in two *neighboring* intervals point to the same node (both successors from I' points to J), and are thus merged and made to point to the same node, namely J. The situation after this is depicted in figure A.4. This lead to that the *union*/*reduction* operation recognizes that node N_3 and I' is equal to the existing node G, therefore N_3 and I' is deleted and all references to these nodes is redirected to node G, the same holds for node H' which is equal to node H. As this reaches the communication array, the *reduction* request $\{\langle 6 = 5\rangle, \langle 7 = 1\rangle, \langle 9 = 1\rangle\}$ is send to the upper computer node, the situation after the full *reduction* of the bottom node is shown in figure A.5. As can be seen the pointer from **6**, **7**, and **9** is kept, these pointers are kept if future *inclusion*/*union* request is propagated from the top computer node to the bottom computer node using these references, before the *reduction* request update the nodes at the top node.

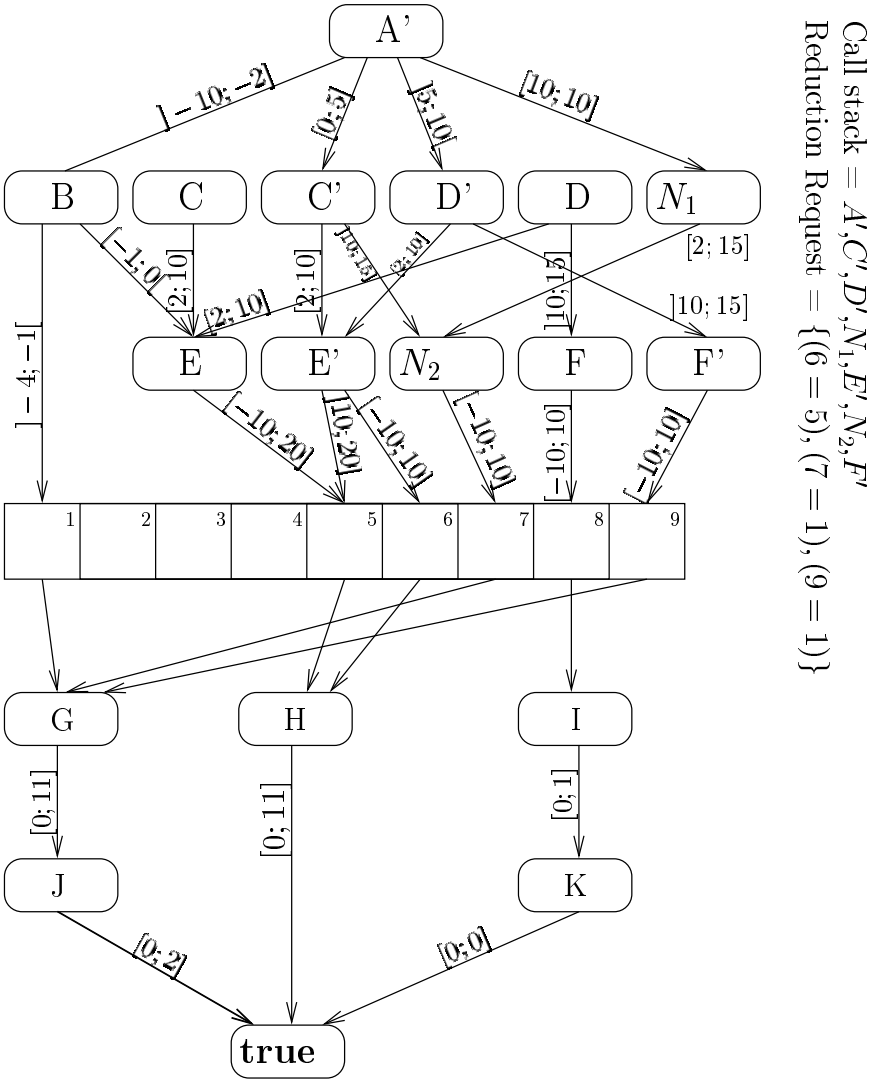


Figure A.5: The situation after the full *union/reduction* has finished on the bottom node.

When the *reduction* request reaches the top node, it parses the message, and redirects all

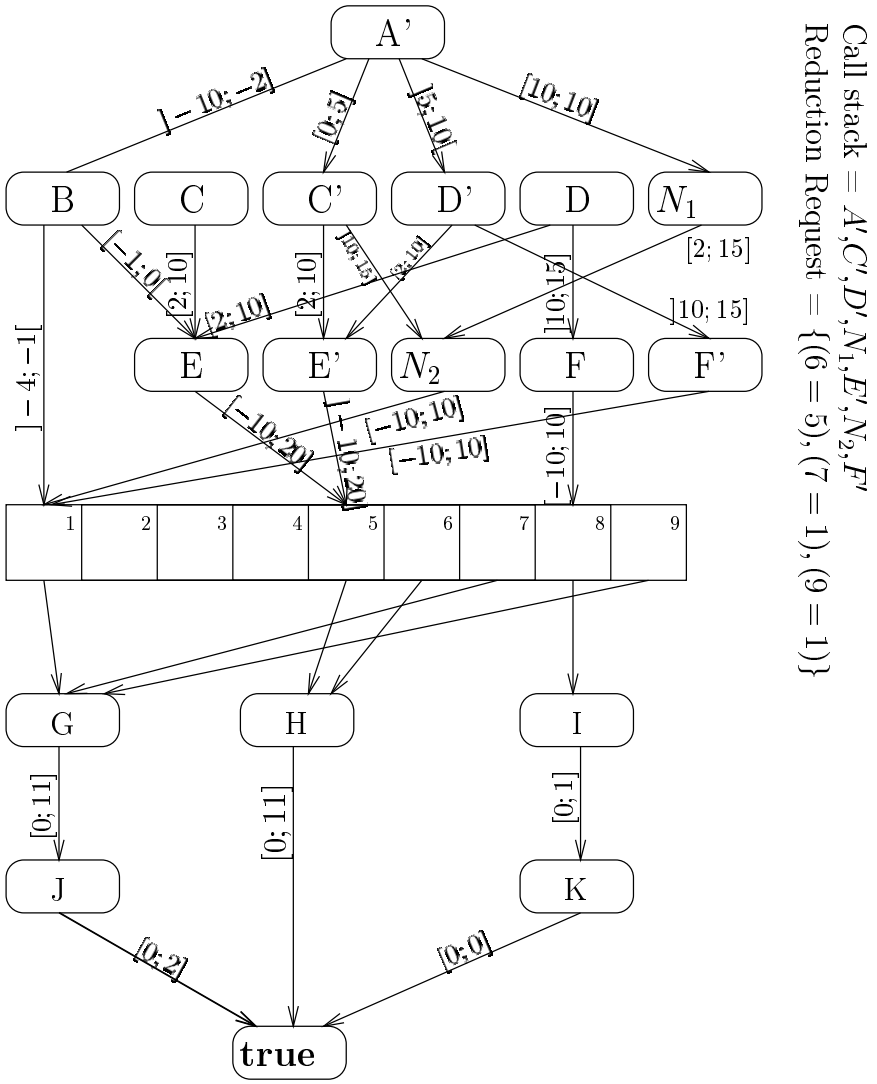


Figure A.6: Show an intermediate situation of the *reduction* on the top computer node.

pointers, referring to **7** and **9** to point to **1**. And all pointers from **6** to point to **5**, all nodes in the call stack is search for such pointers to redirect. The situation after this is done is shown in figure A.6. This *reduction* result in that CDD node N_2 and F' become equivalent. Node E and E' also become equivalent, and are reduced. Therefore all successors previous referring node F' is redirected to point to N_2. The same is the case with successors referring E' which is redirected to point to E. This *reduction* lead to further reduction, figure A.7 show the final CDD after the *union*/*reduction* has finished.

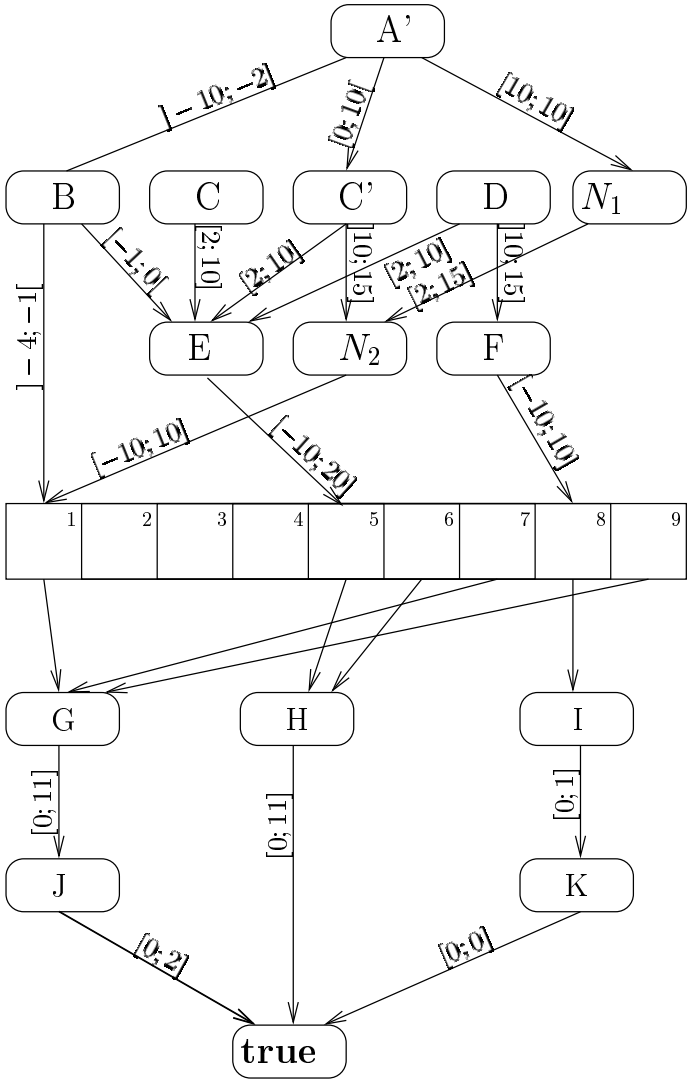


Figure A.7: Show the final situation after the S-CDD and the CDD from figure A.1 is unioned/reduced