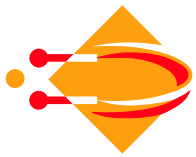

DDDLIB: A Library For Solving Quantified Difference Inequalities

Jesper Møller

IT University Technical Report Series **TR-2002-12**

ISSN 1600–6100 **February 2002**

ISSN 1600–6100

ISBN 87-7949-016-6

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web www.it.edu

DDDLIB: A Library For Solving Quantified Difference Inequalities

Jesper Møller

Department of Innovation, The IT University of Copenhagen
Glentevej 67, DK-2400 Copenhagen NV
Email: jm@it.edu

Abstract. DDDLIB is a library for manipulating expressions in a first-order logic over Boolean variables and inequalities of the form $x_1 - x_2 \leq d$, where x_1, x_2 are real variables and d is an integer constant. Expressions are represented in a semi-canonical data structure called difference decision diagrams (DDDs) which provide efficient algorithms for constructing expressions with the standard Boolean operators (conjunction, disjunction, negation, etc.), eliminating quantifiers, and deciding functional properties (satisfiability, validity and equivalence). The library is written in C and has interfaces for C++, Moscow ML and Ocaml.

1 Introduction

DDDLIB is a library for deciding functional properties of quantified difference inequalities which are expressions φ of the form

$$\varphi ::= \mathbf{0} \mid \mathbf{1} \mid b \mid x_1 - x_2 \sim d \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \Rightarrow \varphi_2 \mid \exists b.\varphi \mid \exists x.\varphi\,,$$

where b is a Boolean variable, x is a real variable, d is an integer constant, and $\sim \in \{\leq, <, =, \neq, >, \geq\}$ is a relational operator. $\mathbf{0}$ and $\mathbf{1}$ denote false and true expressions, and the symbols $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), and $\exists$ (existential quantification) have their usual meaning. We denote by $\varphi[\boldsymbol{v}/\boldsymbol{v'}]$ the substitution of all occurrences of variable v_i' in φ by v_i, for $i = 1, \ldots, n$. The problem of determining whether an expression φ is a tautology, denoted $\models \varphi$, is **PSPACE**-complete [11]. Expressions of this form occur in many areas of mathematics and computer science, some examples are logical formalisms for time, actions, events, and persistence [19, 6, 12, 3], reasoning with temporal constraints [16], and planning and scheduling [2, 9]. However, there are very few tools for performing quantifier elimination and validity checking for this logic efficiently; the primary focus has been on tools for either more expressive theories such as integers or reals with addition and order (e.g., Omega Library [10] and Redlog [7]), or less expressive theories such as quantified Boolean formulae (e.g., SATO [22] and BuDDy [15]).

This paper presents a library called DDDLIB for manipulating quantified difference inequalities. Expressions are represented in a graph data structure called DDDs [18], and the library implements a number of classical algorithms, such as Bryant's Apply algorithm [5] for combining expressions with Boolean operators, Fourier–Motzkin's method [8] for eliminating quantifiers, and Bellman–Ford's shortest-path algorithm [4] for determining satisfiability. The rest of the paper is organized as follows: Section 2 gives an overview of the Moscow ML interface, Sect. 3 is a short introduction to the implementation, and Sect. 4 presents an ML model checker for real-time systems.

2 Interface

This section gives an overview of the Moscow ML interface to DDDLIB (C++ and Ocaml interfaces are also available). Moscow ML is a functional programming language with good support for modeling mathematical problems. The Moscow ML interface utilizes the `Dynlib` structure in Moscow ML for dynamically linking with DDDLIB, so each Moscow ML function simply delegates the call to the corresponding C function in DDDLIB. Variables have type `var`:

```
type var
val RealVar : string -> var
val BoolVar : string -> var
```

Expressions have type `ddd` and are constructed as follows:

```
type ddd
datatype comp = EQ | NEQ | LEQ | GEQ | LE | GR
val False : ddd
val True : ddd
val BoolExpr : var -> ddd
val RealExpr : var * var * comp * int -> ddd
```

Boolean connectives and operators are defined as:

```
val Not : ddd -> ddd
val And : ddd * ddd -> ddd
val Or : ddd * ddd -> ddd
val Imp : ddd * ddd -> ddd
val Exists : var * ddd -> ddd
val Replace : ddd * var * var -> ddd
```

The following functions determine functional properties of an expression:

```
val Tautology : ddd -> bool
val Equivalent : ddd * ddd -> bool
```

3 Implementation

DDDLIB is based on (DDDs) [18] which are directed acyclic graphs with two terminal vertices, **0** and **1**, and a set of non-terminal vertices. Each non-terminal vertex $u = \alpha \to h, l$ denotes the expression $\varphi^u = (\alpha \wedge \varphi^h) \vee (\neg\alpha \wedge \varphi^l)$. The test expression α is a difference constraint of the form $x_i - x_j < d$ or $x_i - x_j \leq d$. A Boolean variable b_i is represented as $x_i - x_i' \leq 0$, where x_i and x_i' are real variables used only in the encoding of b_i. Figure 1 shows an example of a DDD.

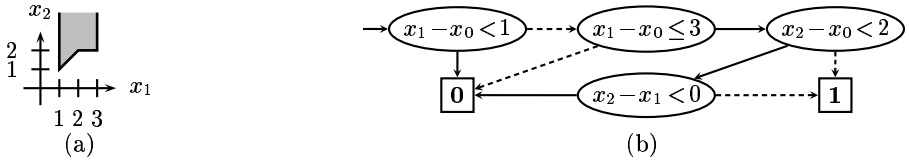

Fig. 1. The expression $\varphi = (1 \leq x_1 - x_0 \leq 3) \wedge \big((x_2 - x_0 \geq 2) \vee (x_2 - x_1 \geq 0)\big)$ as (a) an (x_1, x_2)-plot for $x_0 = 0$, and (b) a difference decision diagram.

As shown in [18], DDDs can be ordered and path-reduced, yielding a semi-canonical form, which makes it possible to check for validity and satisfiability in constant time (as for BDDs). The DDD data structure is not canonical, however, so equivalence checking is performed as a validity check. The operations for constructing DDDs are easily defined recursively on the DDD data structure. The function APPLY$(\oplus, u_1, u_2)$ combines two ordered DDDs rooted at u_1 and u_2 with a Boolean operator $\oplus$ (e.g., negation, conjunction, disjunction). APPLY is a generalization of the version used for BDDs [5] and has running time $O(|u_1||u_2|)$, where $|\cdot|$ denotes the number of vertices in a DDD.

The function EXISTS(v, u) is used to existentially quantify the variable v in a DDD rooted at u. The algorithm is an adoption of the Fourier–Motzkin quantifier-elimination method [8], removing all vertices reachable from u containing v, while keeping all implicit constraints induced by v among the other variables, for example $\exists x_1.(x_0 - x_1 < 1 \wedge x_1 - x_2 \leq 0) \equiv x_0 - x_2 < 1$.

EXISTS computes modified and additional constraints in polynomial time, but has an exponential worst-case running time since the resulting DDD must be ordered. The function PATHREDUCE(u) removes all redundant vertices in a DDD rooted at u, making it semi-canonical: an expression φ^u is satisfiable if and only if PATHREDUCE(u) $\neq \mathbf{0}$; an expression φ^u is a tautology if and only if PATHREDUCE(u) $= \mathbf{1}$. PATHREDUCE determines path feasibility using an incremental Bellman–Ford algorithm with dynamic programming, but has exponential worst-case running time.

4 Applications

In this section we show how to implement a symbolic model checker for δ-programs using the ML interface. A δ-program [17] is a general notation for modeling real-time systems, and consists of a set of commands of the form $\delta\boldsymbol{v}.\varphi$, where $\boldsymbol{v}$ is a vector of variables, and φ is an expression over $\boldsymbol{v}$ and $\boldsymbol{v'}$. A command $\delta(v_1,\ldots,v_n).\varphi$ nondeterministically assigns to each variable v_i any value v_i', for $i = 1,\ldots,n$, such that φ is satisfied. As we have shown in [17], an expression I holds invariantly for a δ-program with initial state φ_0 if and only if $\not\models \mathbf{pre}^*(\neg I) \wedge \varphi_0$, where

$$\mathbf{pre}^*(\neg I) = \mu X\Big[\neg I \vee \bigvee_{\delta\boldsymbol{v}.\varphi} \exists \boldsymbol{v'}.(\varphi \wedge X[\boldsymbol{v'}/\boldsymbol{v}])\Big], \tag{1}$$

and where $\mu X[f(X)]$ is the least fixpoint of $f(X)$.

It is straightforward to model timed systems as δ-programs. The key idea is to introduce a variable z interpreted as the common zero-point of all clocks in a timed system. A process in Fischer's protocol [13] can be modeled as follows:

$$\begin{array}{l}
\delta(a_i,b_i,x_i).\big(\neg b_i \wedge \neg a_i' \wedge b_i' \wedge x_i' - z = 0 \wedge \bigwedge_{j=1}^{N} \neg id_j\big)\\
\delta(a_i,b_i,x_i,id_i).\big(\neg a_i \wedge b_i \wedge a_i' \wedge \neg b_i' \wedge x_i - z \leq 10 \wedge x_i' - z = 0 \wedge id_i'\big)\\
\delta(a_i,b_i).\big(a_i \wedge \neg b_i \wedge a_i' \wedge b_i' \wedge x_i - z > 10 \wedge id_i \wedge \bigwedge_{j\neq i} \neg id_j\big)\\
\delta(a_i,b_i,id_i).\big(a_i \wedge b_i \wedge \neg a_i' \wedge \neg b_i' \wedge \neg id_i'\big)\\
\delta(z).\big(z' \leq z \wedge \big(\forall z''(z' \leq z'' \leq z) \Rightarrow \bigwedge_{i=1}^{N}(\neg a_i \wedge b_i \Rightarrow 0 \leq x_i - z'' \leq k)\big)\big),
\end{array}$$

where the last command is common for all processes and models the advancing of time. The initial state is given by $\varphi_0 = \bigwedge_{i=1}^{N}\big(\neg a_i \wedge \neg b_i \wedge \neg id_i \wedge x_i - z = 0\big)$, and the property that only one process is in the critical state can be expressed as $I = \bigwedge_{i=1}^{N}\bigwedge_{j\neq i}\neg\big(a_i \wedge b_i \wedge a_j \wedge b_j\big)$.

We can model a δ-program in ML as an initial set of states `phi0` of type `ddd`, and a list of commands `cmds` of type `var list * var list * ddd` (two lists of unprimed and primed variables, and an expression). Using the functions described in Sect. 2, Fischer's protocol can be modeled with less than 25 lines of ML code. We can use Eq. (1) directly to implement a symbolic model checker for δ-programs modeled in ML:

```
fun verify (cmds, phi0, I) =
   let val ReplaceL = ListPair.foldl (fn (v',v,d) => Replace(d,v',v))
       fun pre x = List.foldl (fn ((v,v',d),r) =>
             Or(r, List.foldl Exists (And(d, ReplaceL r (v',v))) v')) x
       fun lfp f =
          let fun f' x =
             let val y = f x in if Equivalent(x,y) then y else f' y end
          in f' False end
       val prestar = lfp (fn x => Or(Not I, pre x cmds))
   in not (Tautology(And(phi0, prestar))) end
```

We have used `verify` to check that Fischer's protocol guarantees mutual exclusion. Within 1 hour it is possible to verify $N = 15$ processes on a 1 GHz Pentium III PC. This is comparable with other real-time verification tools (e.g., UPPAAL [14] and KRONOS [21]). The size of $\mathbf{pre}^*(\neg I)$ is 370,501 DDD vertices.

5 Conclusion

This paper has presented a library for manipulation of quantified difference inequalities implemented using a data structure called DDDs. We have demonstrated the applicability of the library in symbolic model checking by given a backward reachability algorithm as a 10-line ML function that verifies whether a δ-program meets a given property. The library has also been used for symbolic model checking of event-recording automata [20], and verification of infinite-state systems [1]. DDDLIB is available at `http://www.it.edu/people/jm`.

References

1. P.A. Abdulla and A. Nylén. Better is better than well: On efficient verification of infinite-state systems. In *Proc. 15th IEEE Symposium on Logic in Computer Science (LICS)*, pages 132–140. IEEE Computer Society Press, June 2000.
2. J.F. Allen, H. Kautz, R.N. Pelavin, and J. Tenenberg, editors. *Reasoning about Plans.* Morgan Kaufmann, San Mateo, California, 1991.
3. Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings 5th Annual IEEE Symposium on Logic in Computer Science*, Philadelphia, Pensylvania, 1990. IEEE Computer Society Press.
4. R. Bellman. On a routing problem. *Quarterly of Applied Math.*, 16(1):87–90, 1958.
5. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
6. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
7. Andreas Dolzmann and Thomas Sturm. Redlog user manual. Technical Report MIP-9905, FMI, Universität Passau, D-94030 Passau, Germany, April 1999.
8. J.B.J. Fourier. Second extrait. In *Oeuvres*, pages 325–328. Gauthiers-Villars, 1890.
9. Mark S. Fox. *Constraint-directed Search: A Case Study of Job-Shop Scheduling.* Morgan Kaufmann Publishers, 1987.
10. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.
11. Manolis Koubarakis. Complexity results for first-order theories of temporal constraints. In *Proc. 4th Intl. Conference on Principles of Knowledge Representation and Reasoning*, pages 379–390. Morgan Kaufmann, 1994.
12. Robert A. Kowalski and Marek J. Sergot. A logic-based calculus of events. In *Foundations of Knowledge Base Management (Xania)*, pages 23–55, 1985.
13. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Comp. Systems*, 5(1):1–11, 1987.
14. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.
15. Jørn Lind-Nielsen. *BuDDy: Binary Decision Diagram package.* The IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, 1.9 edition, August 2000.
16. Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, 87(1–2):343–385, 1996.
17. J. Møller. Efficient verification of timed systems using backward reachability analysis. Technical Report TR-2002-11, IT University of Copenhagen, February 2002.
18. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proc. 13th International Workshop on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, September 1999.
19. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 31– 2 1977. IEEE Computer Society Press.
20. Maria Sorea. TEMPO: A model checker for event-recording automata. Technical Report SRI-CSL-01-04, SRI International, Computer Science Laboratory, 2001.
21. S. Yovine. Kronos: A verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1/2), October 1997.
22. Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, 1997.