DDDLIB: A Library For Solving Quantified Difference Inequalities

Jesper B. Møller

Department of Innovation, IT University of Copenhagen
jm@it.edu

Abstract. DDDLIB is a library for manipulating formulae 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. Formulae are represented in a semi-canonical data structure called difference decision diagrams (DDDs) which provide efficient algorithms for constructing formulae 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++, Standard ML and Objective Caml.

1 Introduction

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

$$\phi ::= \mathbf{0} \mid \mathbf{1} \mid b \mid x_1 - x_2 \sim d \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \exists b.\phi \mid \exists x.\phi\,,$$

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, respectively, and the symbols $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), and $\exists$ (existential quantification) have their usual meaning. We denote by $\phi[\boldsymbol{v}/\boldsymbol{v'}]$ the substitution of all occurrences of variable v_i' in ϕ by v_i, for $i = 1, \ldots, n$. The problem of determining whether a formula ϕ is a tautology, denoted $\models \phi$, is **PSPACE**-complete [11]. Formulae of this form occur in many areas of mathematics and computer science, some examples are logical formalisms for time, actions, events, and persistence [20, 7, 12, 3], reasoning with temporal constraints [17], and planning and scheduling [2, 10]. 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 [21] and Redlog [8]), or less expressive theories such as quantified Boolean formulae (e.g., SATO [25] and BuDDy [16]).

This paper presents a library called DDDLIB for manipulating quantified difference inequalities. Formulae 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 [6] for combining formulae with Boolean operators,

Fourier–Motzkin's algorithm [9] for eliminating quantifiers, and Bellman–Ford's shortest-path algorithm [5] for determining satisfiability. A preliminary version of DDDLIB has been used to implement a verification tool for infinite-state systems [1], and a symbolic model checker for event-recording automata [23]. Larsen et al. [14] have described a data structure similar to DDDs called clock difference diagrams (CDDs); however, they do not define an algorithm for eliminating quantifiers in a CDD, and the algorithm for determining satisfiability of a CDD is different. CDDs have been implemented in the tool UPPAAL [4].

The paper is organized as follows: Section 2 gives an overview of the Standard ML (SML) interface, Section 3 is a short introduction to the implementation, and Section 4 presents a model checker for real-time systems written in SML.

2 Interface

This section gives an overview of the SML interface to DDDLIB. SML is a functional programming language with good support for modeling mathematical problems. The current version of DDDLIB uses Moscow ML [22] which is a lightweight implementation of SML. This implementation supports dynamic linkage with C functions, so each SML function simply delegates calls to the corresponding C function in DDDLIB. Variables in DDDLIB have type `var`:

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

Formulae 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:

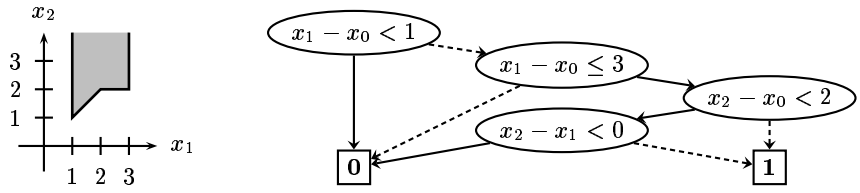
```
val Not : ddd -> ddd
val And : ddd * ddd -> ddd
val Or : ddd * ddd -> ddd
val Imp : ddd * ddd -> ddd
val Exists : var * ddd -> ddd
val Replace : ddd * var * var -> ddd
```

The following functions determine functional properties of a formula:

```
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



(a) (b)

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

vertex $u = \alpha \rightarrow h, l$ denotes the formula $\phi^u = (\alpha \wedge \phi^h) \vee (\neg\alpha \wedge \phi^l)$. 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. 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 $\textsc{Apply}(\oplus, u_1, u_2)$ 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 [6] and has running time $O(|u_1||u_2|)$, where $|\cdot|$ denotes the number of vertices in a DDD.

The function $\textsc{Exists}(v, u)$ is used to existentially quantify the variable v in a DDD rooted at u. The algorithm is an adoption of Fourier–Motzkin's quantifier-elimination algorithm [9], 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 $\textsc{PathReduce}(u)$ removes all redundant vertices in a DDD rooted at u making it semi-canonical, which means that a formula ϕ^u is satisfiable if and only if $\textsc{PathReduce}(u) \neq \mathbf{0}$. Similarly, ϕ^u is a tautology if and only if $\textsc{PathReduce}(u) = \mathbf{1}$. PathReduce determines path feasibility using an incremental Bellman–Ford algorithm with dynamic programming, but has exponential worst-case running time.

4 Applications

This section shows how to implement a symbolic model checker for δ-programs using the SML interface. A δ-program [19] is a general notation for modeling real-time systems, and consists of a set of commands of the form $\delta\boldsymbol{v}.\phi$, where $\boldsymbol{v}$ is a vector of variables, and ϕ is a formula over $\boldsymbol{v}$ and $\boldsymbol{v'}$. A command $\delta(v_1, \ldots, v_n).\phi$ nondeterministically assigns to each variable v_i any value v_i', for $i = 1, \ldots, n$, such

that ϕ is satisfied. 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. 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 progression of time. The initial state is $\phi_0 = \bigwedge_{i=1}^{N}\big(\neg a_i \wedge \neg b_i \wedge \neg id_i \wedge x_i = z\big)$, and the property that only one process is in the critical section 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 SML 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 a formula). Using the functions described in Section 2, Fischer's protocol can be modeled with less than 25 lines of SML code. As shown in [19], a formula I is invariant for a δ-program with initial state ϕ_0 if and only if $\not\models \mathbf{pre}^*(\neg I) \wedge \phi_0$, where

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

and where $\mu X[f(X)]$ is the least fixpoint of $f(X)$. We can use Eq. (1) directly to implement a symbolic model checker for δ-programs modeled in SML:

```
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
```

I 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 [15] and KRONOS [24]). The size of $\mathbf{pre}^*(\neg I)$ is 370,501 DDD vertices.

5 Conclusion

This paper has presented a library for manipulating quantified difference inequalities implemented using the data structure DDDs. I have demonstrated the applicability of the library in symbolic model checking of real-time systems by giving a 10-line SML function for verifying safety properties of δ-programs using backward reachability analysis. DDDLIB is available at `www.it.edu/research/ddd`.

References

1. P.A. Abdulla and A. Nylén. Better is better than well: On efficient verification of infinite-state systems. In *Proc. 15th LICS*, pages 132–140, 2000.
2. J.F. Allen, H. Kautz, R.N. Pelavin, and J. Tenenberg, editors. *Reasoning about Plans*. Morgan Kaufmann, San Mateo, California, 1991.
3. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. 5th LICS*, pages 414–425, 1990.
4. G. Behrmann, K.G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proc. 11th Conference on Computer Aided Verification*, LNCS 1633, pages 341–353, 1999.
5. R. Bellman. On a routing problem. *Quarterly of Applied Math.*, 16(1):87–90, 1958.
6. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
7. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
8. A. Dolzmann and T. Sturm. Redlog user manual. Technical Report MIP-9905, FMI, Universität Passau, D-94030 Passau, Germany, April 1999.
9. J.B.J. Fourier. Second extrait. In *Oeuvres*, pages 325–328. Gauthiers-Villars, 1890.
10. M.S. Fox. *Constraint-directed Search: A Case Study of Job-Shop Scheduling*. Morgan Kaufmann Publishers, 1987.
11. M. Koubarakis. Complexity results for first-order theories of temporal constraints. In *Principles of Knowledge Representation and Reasoning*, pages 379–390, 1994.
12. R.A. Kowalski and M.J. Sergot. A logic-based calculus of events. In *Proc. Foundations of Knowledge Base Management*, pages 23–55, 1985.
13. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Comp. Systems*, 5(1):1–11, 1987.
14. K.G. Larsen, J. Pearson, C. Weise, and W. Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.
15. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
16. J. Lind-Nielsen. *BuDDy: Binary Decision Diagram package*. IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, May 2001.
17. I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. *Artificial Intelligence*, 87(1–2):343–385, 1996.
18. J. Møller, J. Lichtenberg, H.R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proc. Computer Science Logic*, LNCS 1683, pages 111–125, 1999.
19. J.B. Møller. Simplifying fixpoint computations in verification of real-time systems. Technical Report TR-2002-15, IT University of Copenhagen, April 2002.
20. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57, 1977.
21. W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. *Comm. of the ACM*, 35(8):102–114, August 1992.
22. S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Owner's Manual*, June 2000.
23. M. Sorea. TEMPO: A model-checker for event-recording automata. In *Proc. Workshop on Real-Time Tools*, August 2001. Also as SRI Technical Report CSL-01-04.
24. S. Yovine. KRONOS: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, October 1997.
25. H. Zhang. SATO: An efficient propositional prover. In *Proc. Conference on Automated Deduction*, pages 272–275, 1997.