Jedd: A BDD-based relational extension of Java

Sable Technical Report No. 2003-7

Ondřej Lhoták   Laurie Hendren

November 21, 2003

Contents

List of Figures

List of Tables

Abstract

In this paper we present Jedd, a language extension to Java that supports a convenient way of programming with Binary Decision Diagrams (BDDs). The Jedd language abstracts BDDs as database-style relations and operations on relations, and provides static type rules to ensure that relational operations are used correctly.

The paper provides a description of the Jedd language and reports on the design and implementation of the Jedd translator and associated runtime system. Of particular interest is the approach to assigning attributes from the high-level relations to physical domains in the underlying BDDs, which is done by expressing the constraints as a SAT problem and using a modern SAT solver to compute the solution. Further, a runtime system is defined that handles memory management issues and supports a browsable profiling tool for tuning the key BDD operations.

The motivation for designing Jedd was to support the development of whole program analyses based on BDDs, and we have used Jedd to express five key interrelated whole program analyses in our Soot compiler framework. We provide some examples of this application and discuss our experiences using Jedd.

1 Introduction

Binary Decision Diagrams (BDDs) [6] have been widely used for efficiently solving problems in model checking, and more recently we demonstrated that BDDs are very useful for defining compact and efficient solvers for whole program analyses like points-to analysis [4]. As BDDs have been in use for some time, there exist several excellent libraries providing efficient representations, algorithms and memory management techniques for BDDs, including two C-based libraries we have been using, BuDDy [9] and CUDD [17].

Based on our very positive experience with using BDDs for program analysis, we embarked on a project to express a number of key, interrelated whole program analyses for Java using BDDs inside our Java compiler framework, Soot [19]. We still wanted to use existing efficient C-based libraries, but now we required a clean and efficient interface between the Java code of our compiler and our BDD-based algorithms.

In developing our approach, it soon became apparent that a simple strategy of providing a Java wrapper to interface with a BDD library was not a good solution, for many reasons. First, we found that the interface provided by the existing BDD libraries is very low level, and as we attempted to express several complex interrelated analyses, understanding and maintaining our code became difficult. Moreover, programming at such a low level was error prone, and errors in our code led to either the BDD library aborting, or worse, to incorrect results. The implicit nature of the BDD representation made these errors difficult to track down. Furthermore, we found that it is quite difficult to match the memory management in Java with the reference counter based schemes employed in the BDD packages. Finally, we found that tuning a BDD-based algorithm requires profiling information about the size and shape of the underlying BDDs at each program step. We had previously developed some ad-hoc methods for visualizing this information, but a more automated approach was really needed.

Our solution, and the topic of this paper, was the development of: (1) Jedd, a language extension to Java, which provides a high-level way of programming BDD-based algorithms based on relations and operations on relations; (2) an associated translator which automatically translates Jedd to Java code that efficiently interacts with back-end solvers; and (3) run-time support for memory management, debugging and profiling of BDD operations. The key aspects of our approach, and the main contributions of this paper, are:

BDDs abstracted as relations: Rather than expose BDDs and their low-level operations directly, our Jedd language provides a more abstract data type based on database-style relations, and operations on those relations. In developing program analyses using BDDs, we have found that this is a more appropriate level of abstraction.

Static and dynamic type checking: When using a BDD library directly, there is very little type information to help the programmer determine if BDD operations are used in a consistent and correct fashion. In the Jedd approach, all operations on relations have static type rules which help to eliminate many programmer errors. Properties that cannot be checked statically are enforced by runtime checks.

Code generation strategy: We provide a strategy to convert the high-level relational operations into low-level BDD operations, and a mechanism for interfacing to several different BDD back-ends.

Algorithm for physical domain assignment: An important issue in programming with BDDs is how to assign physical domains of BDD variables to the problem being solved. When programming directly with BDDs, the programmer must explicitly make all of the assignments and ensure that BDD operations are applied to the correct physical domains, which can be quite a tedious process. Furthermore, a small change in physical domain mappings may require many changes in the program. When specifying a program using the Jedd language, the user specifies only the important mappings, and the translator completes a consistent mapping for the remainder of the program. The problem of assigning physical domains turns out to be NP-complete. We provide an algorithm to express it as an instance of the SAT problem, and we show that, using modern SAT solvers, the time to find a solution is very acceptable. In cases where no solution exists, we provide information back to the programmer to help them modify their program to make the problem solvable.

Run-time support for memory management: BDD solvers make use of reference counter memory management techniques to efficiently reclaim the BDD data structures. These require the programmer to explicitly manipulate the reference counts, which is error-prone and does not fit with the Java memory management model. Jedd frees the programmer from this task by automatically managing all reference counts, and freeing BDDs as soon as it is safe to do so.

BDD profiler: In our previous and current work with BDDs, we found that tuning the BDD-based algorithms required profiling the size and shape of the BDD data structures at each program point. Our Jedd system allows the user to automatically generate profiling information that can be browsed using any HTML browser, and which provides both counts of the number of operations applied, and graphical figures showing the size and shape of the underlying BDD data structures at each program point.

Proof of concept applications: In order to verify that our approach works, we have implemented several interrelated whole program analyses using the Jedd system. We found that the algorithms were quite easy to specify, compact, and that the resulting BDD solvers were efficient. We also found that the physical domain assignment algorithm worked well, ran in acceptable times, and provided good mappings of attributes to physical domains.

A high-level overview of the complete Jedd system is given in Figure 1. Jedd programs are written in our extension to Java, and are translated to Java programs using the `jeddc` compiler. The `jeddc` compiler is composed of a front-end (parser and semantic analysis) and a back-end (physical domain assignment and code generation). The physical domain assignment module calls an external SAT solver tool. The Java files produced by `jeddc` and other ordinary Java files are compiled using a Java compiler, producing class files with calls to the Jedd runtime library, which interfaces using JNI to a BDD package. A JVM is used to execute the classes along with the Jedd runtime. The runtime also includes a profiler, which writes profile information into a SQL database. When combined with CGI scripts accessing the database, an HTML browser can be used to navigate profiler views of BDD operations.

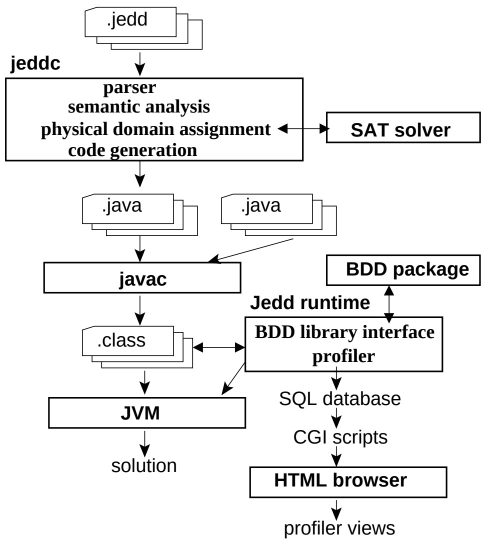


Figure 1: Overview of Jedd system

The remainder of this paper is structured as follows. In Section 2, we give an introduction to the Jedd language, along with some illustrative examples from our application of Jedd to program analysis. In Section 3, we explain the key aspects of the `jeddc` compiler, with a particular emphasis on how we handle code generation and physical domain assignment. In Section 4, we describe the important elements of the design of our runtime system and profiler, and in Section 5, we briefly report on our experiences with using Jedd to implement five interrelated whole program analyses in the Soot compiler framework. Finally, in Section 6, we discuss related work, and in Section 7, we conclude and suggest future work.

2 Jedd Language

In this section, we describe the Jedd language, and illustrate key concepts with examples. These examples are taken from extensions to the Soot framework that we have written in Jedd. These extensions perform interrelated whole-program analyses such as points-to analysis, call graph construction, and side-effect analysis in BDDs, and together they form a significant application of Jedd. Figure 2 shows an overview of the five main modules that have been implemented in Jedd and how they communicate with each other. In Figure 4, we show a simplified version of the core of the Virtual Call Resolution module to give an idea of what Jedd code looks like.

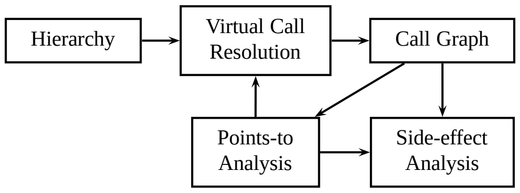


Figure 2: BDD-based analyses in Soot

The remainder of this section is structured as follows. In Section 2.1, we introduce a new data type, relations, and in Section 2.2, we describe the new operations provided for relations. The grammar for the extensions is given in Figure 5 and the type rules in Figure 7. In Section 2.3, we describe how objects can be extracted from relations back to Java.

2.1 Relations

Jedd extends the Java language with a new data type, database-style relations. Informally, a relation is just a set of tuples. For example, the top part of Figure 3 shows a relation that contains two tuples, each tuple contains values for the attributes *type*, *signature* and *method*. An ***attribute*** is just a named ***domain***, where a ***domain*** is a set of Java objects such as the set of all types in a program being analyzed, or the set of all methods. All tuples in a relation must have the same set of attributes, and we call the set of attributes for a relation its ***schema***. The relations in Jedd are high-level abstractions for BDDs, and there must exist some way of mapping the attributes of the higher-level Jedd relation to the underlying BDDs. A ***physical domain*** is a set of BDD variables used to represent an attribute of a relation.

type	signature	method
A	foo()	A.foo()
B	bar()	B.bar()

```
// declaring a relation with three attributes
<type, signature, method> implementsMethod;

// declaring a relation with explicit mappings
//    to physical domains
<type:T1, signature:S1, method:M1>
                    implementsMethodMapped;
```

Figure 3: Relations

```
1  <rectype, signature, tgttype, method> answer = 0B;
2  public void resolve( <rectype, signature> receiverTypes, <subtype, supertype> extend ) {
3    <rectype, signature, tgttype> toResolve = (rectype=>rectype tgtttype) receiverTypes;
4
5    do {
6        <rectype:T1, signature:S1, tgttype:T2, method:M1> resolved =
7             toResolve{tgttype, signature} >< declaresMethod{type, signature};
8         answer |= resolved;
9         toResolve -= (method=>) resolved;
10        toResolve = (supertype=>tgttype) (toResolve {tgttype} <> extend {subtype});
11   } while( toResolve != 0B );
12 }
```

(a)

type	signature
B	foo()
B	bar()

(b)

rectype	signature	tgttype
B	foo()	B
B	bar()	B

(c)

rectype	signature	tgttype	method
B	bar()	B	B.bar()

(d)

subtype	supertype
B	A

(e)

rectype	signature	tgttype
B	foo()	B

(f)

rectype	signature	supertype
B	foo()	A

(g)

rectype	signature	tgttype	method
B	foo()	A	A.foo()

Figure 4: Example of resolving virtual method calls (a) `receiverTypes` (b) `toResolve` in line 3 (c) `resolved` in first iteration (d) `extend` (e) `toResolve` in line 10 (f) result of composition in line 10 (g) `resolved` in second iteration

The bottom part of Figure 3 shows two different ways of declaring a relation in Jedd. The first example declares the `implementsMethod` relation which has three attributes. In this case no physical domain mapping was given for the attributes and it is left to the Jedd compiler to find a mapping. However, sometimes the programmer does want to expose the mapping of attributes to physical domains, and the declaration of `ImplementsMethodMapped` declares another relation, with the same schema as before, but with explicit mappings to the physical domains `T1`, `S1` and `M1`. The details of physical domains and the algorithm to perform the mapping of attributes to physical domains are described in Section 3.

Note that in our example, we have just used names for attributes (i.e. `type`, `signature` and `method`). These names must be defined by the Jedd programmer by defining Java classes that implement the interface `jedd.Attribute` which specifies the domain and the name. Similarly, each domain is defined by implementing the `jedd.Domain` interface, and each physical domain is defined by implementing the `jedd.PhysicalDomain` interface. Each domain specifies the maximum number of objects in it, and provides a mapping from Java objects to integers and vice versa. The integer is used to represent the object in BDDs. Jedd's type checker ensures that any use of an interface, domain or physical domain is a subclass of the correct interface.

Only relations with the same schema are assignable and comparable. Like other primitive Java types, relations are passed by value, not by reference. Jedd defines two constants, `0B` and `1B`, the empty relation and the full relation (containing all possible tuples), respectively. These constants have a special type that makes them comparable and assignable to any relation type, much like Java's `null` constant. Jedd also provides an easy way to create new tuples from Java objects. For example, in Soot, we use the following code to add a tuple to the `implementsMethod` relation:

```
void addMethod( Type newType, Signature newSig,
                SootMethod newMethod ) {
  implementsMethod |= new { newType=>type,
    newSignature=>signature, newMethod=>method };
}
```

The `new` expression constructs a relation of a single tuple with the Java objects `newType`, `newSignature`, and `newMethod` in attributes type, signature, and method, respectively. This relation of a single tuple is then added into the `implementsMethod` relation.

2.2 Operations on Relations

2.2.1 Set Operations and Comparison

The set union, intersection, and difference operations on relations viewed as sets of tuples are written in Jedd using the operators `|`, `&`, and `-`, respectively. These operations make sense only when their arguments have the same schema, and this is enforced by the static type checking. Jedd also defines the expected shorthand assignment operators `|=`, `&=`, and `-=`. In the example above, the `|=` operator is used to add the new tuple to the `implementsMethod` relation. The `==` and `!=` operators are used to compare relations for equality, an operation that takes only constant time in BDDs.

2.2.2 Projection and Attribute Operations

Jedd provides three operations on the attributes of a relation. A ***projection*** removes an attribute from the relation, along with the objects associated with the attribute in each tuple. Recall that relations are sets of tuples with no duplicates. Since removing an attribute from two tuples that differ only in that attribute makes the tuples equal, a projection may reduce the number of tuples in a relation. ***Attribute renaming*** substitutes one attribute for another, without changing the objects stored in tuples. ***Attribute copying*** adds a new attribute to a relation. In each tuple, the new attribute is mapped to the same object as the attribute being copied.

To illustrate how these operations are used, we will walk through the problem of resolving virtual method calls given the actual types of the receivers. Given a receiver type and a method signature, the algorithm must search for a class implementing a method with the signature, starting from the receiver type and moving up the class hierarchy. In Jedd, this is done for a relation of signatures and receiver types at once, rather than one signature and receiver type at a time.

The Jedd code for this algorithm is shown in Figure 4. It starts with the relation `receiverTypes`, with each tuple specifying a receiver type and a method signature at some call site. An example of such a relation is shown in Figure 4(a), specifying the receiver type B at two call sites with signatures foo() and bar(). Before starting to walk up the hierarchy starting from the receiver type, the algorithm first saves a copy of the original receiver type in each tuple using the attribute copying operation in line 3. In the resulting `toResolve` relation, each tuple contains the method signature and two copies of the receiver type (see Figure 4(b)). The next step will be to determine whether the current class implements a method with the required signature. Before explaining how to do this, we must pause to introduce the join and composition operations.

2.2.3 Join and Composition

The join and composition operations combine the information from two relations into a single relation. In addition to a pair of relations, they require a list of zero or more attributes from the left relation to compare with a corresponding list of attributes from the right relation. The new relation is constructed from all pairs of tuples from the two relations which match in the attributes being compared. Each such pair of tuples is merged into a single tuple in the final relation. The difference between a composition and join is in the attributes which are included in the final relation. A composition (denoted `<>`) projects away all of the attributes being compared. A join (denoted `><`) keeps the attributes being compared, but only those from the left relation, since their values are equal to those from the right relation. Although a composition is equivalent to a join followed by a projection, Jedd includes both operations because both are common, and a composition is implemented more efficiently than a join followed by a projection.

To see how these operations are used, let us return to our example. Recall that the `toResolve` relation contains, in each tuple, a method signature, and two copies of the receiver type, as shown in Figure 4(b). The next step is to determine whether the class of the receiver type implements a method with the signature. This is done using the join on line 7, which joins this relation with the `implementsMethod` relation in Figure 3, matching the current class (tgttype attribute) with the class implementing the method (type attribute of `implementsMethod`), and the method signature (signature attribute) with the method signature of the implemented method (signature attribute of `implementsMethod`). For each class and method signature being resolved, if the class implements a method with a matching signature, then the resulting relation `resolved` contains a tuple with the method signature, two copies of the receiver type, and the target method. In our example, the only match is type B and signature bar(), resulting in the `resolved` relation relation in Figure 4(c). In general, these are the method calls that we have just resolved by finding a method with the desired signature, so in line 8, we add them to our answer.

The T1, S2, T2, and M1 on line 6 are physical domains, indicating how the attributes are to be assigned to BDD variables. In this example, the programmer supplies them for the `resolved` relation, and the physical domain assignment algorithm discussed in Section 3.4 determines a reasonable assignment for all other expressions.

The next step is to remove the resolved call sites from the set of sites left to resolve. However, the `resolved` relation has the method attribute which `toResolve` lacks, so it must be projected away in line 9 before the resolved call sites can be subtracted. After doing this to our example, we are left with the `toResolve` relation in Figure 4(e).

The final step is to move up the class hierarchy by replacing each class in the tgttype attribute with its immediate superclass. This is done with a composition (in line 10) of the `toResolve` relation with the `extend` relation which has been passed in from the hierarchy, and encodes the immediate superclass (extends) relationship. In our example, as Figure 4(d) shows, B is a subtype of A. The tgttype attribute is matched with the subtype attribute in the extends relation, and a composition is used rather than a join because the attributes being compared (the subtype) are not needed; only the supertype attribute coming from the extends relationship is needed. The resulting relation has replaced each object in the tgttype attribute of `toResolve` with its immediate superclass, as shown in Figure 4(f). Before it can be assigned to `toResolve`, the supertype attribute must be renamed to tgttype to match the schema of `toResolve`. Finally, if the set of call sites to be resolved is not yet empty, the algorithm starts another iteration of the loop to resolve them. Figure 4(g) shows the call resolved in the second iteration. Together, the relations in Figures 4(c) and (g) show the final result: the targets of calling foo() and bar() with a receiver of type B are A.foo() and B.bar(), respectively.

2.2.4 Selection

We have not yet mentioned the common relational operation selection, which returns the subset of the tuples having specified objects in certain attributes. This is most easily implemented by constructing a relation containing the desired objects, and joining it with the relation of interest. Therefore, Jedd does not have a separate selection operation.

Added productions:

⟨*Type*⟩ ::= ‘<’ ⟨AttributePhys⟩ (‘`,`’ ⟨AttributePhys⟩)* ‘>’

⟨AttributePhys⟩ ::= ⟨Attribute⟩ | ⟨Attribute⟩ ‘`:`’ ⟨Attribute⟩

⟨Attribute⟩ ::= ⟨*ClassOrInterfaceType*⟩

⟨*UnaryExpressionNotPlusMinus*⟩ ::= ⟨RelExprJoin⟩

⟨RelExprJoin⟩ ::= ⟨RelExpr⟩ | ⟨Join⟩

⟨Join⟩ ::= ⟨RelExprJoin⟩ ⟨AttrList⟩ ⟨JoinSym⟩ ⟨RelExpr⟩ ⟨AttrList⟩

⟨AttrList⟩ ::= ‘`{`’ ⟨Attribute⟩ (‘`,`’ ⟨Attribute⟩)* ‘`}`’

⟨JoinSym⟩ ::= ‘>’ ‘<’ | ‘<’ ‘>’

⟨RelExpr⟩ ::= ⟨Replace⟩ | ⟨*PostfixExpression*⟩

⟨Replace⟩ ::= ‘`(`’ ⟨Replacement⟩ (‘`,`’ ⟨Replacement⟩)* ‘`)`’ ⟨RelationExpr⟩

⟨Replacement⟩ ::= ⟨Attribute⟩ ‘`=>`’ | ⟨Attribute⟩ ‘`=>`’ ⟨Attribute⟩ | ⟨Attribute⟩ ‘`=>`’ ⟨Attribute⟩ ⟨Attribute⟩

⟨*Literal*⟩ ::= ‘`new`’ ‘`{`’ ⟨LiteralPiece⟩ (‘`,`’ ⟨LiteralPiece⟩)* ‘`}`’ | ‘`0B`’ | ‘`1B`’

⟨LiteralPiece⟩ ::= ⟨*Expression*⟩ ‘`=>`’ ⟨AttributePhys⟩

Removed production:

⟨*UnaryExpressionNotPlusMinus*⟩ ::= ⟨*PostfixExpression*⟩

Figure 5: Jedd grammar productions

Added productions:

$\langle$*ArrayAccess*$\rangle$ ::= $\langle$*ClassInstanceCreationExpression*$\rangle$ '`[`' $\langle$*Expression*$\rangle$ '`]`'

$\langle$*ExplicitConstructorInvocation*$\rangle$ ::= $\langle$*ClassInstanceCreationExpression*$\rangle$ '`.`' '`this`' '`(`' $\langle$*ArgumentListOpt*$\rangle$ '`)`' '`;`'
  | $\langle$*ClassInstanceCreationExpression*$\rangle$ '`.`' '`super`' '`(`' $\langle$*ArgumentListOpt*$\rangle$ '`)`' '`;`'

$\langle$*ClassInstanceCreationExpression*$\rangle$ ::= $\langle$*ClassInstanceCreationExpression*$\rangle$ '`.`' '`new`' $\langle$*SimpleName*$\rangle$ '`(`' $\langle$*ArgumentListOpt*$\rangle$ '`)`'
  | $\langle$*ClassInstanceCreationExpression*$\rangle$ '`.`' '`new`' $\langle$*SimpleName*$\rangle$ '`(`' $\langle$*ArgumentlistOpt*$\rangle$ '`)`' $\langle$*ClassBody*$\rangle$

$\langle$*FieldAccess*$\rangle$ ::= $\langle$*ClassInstanceCreationExpression*$\rangle$ '`.`' IDENTIFIER

$\langle$*MethodInvocation*$\rangle$ ::= $\langle$*ClassInstanceCreationExpression*$\rangle$ '`.`' IDENTIFIER '`(`' $\langle$*ArgumentListOpt*$\rangle$ '`)`'

$\langle$*UnaryExpressionNotPlusMinus*$\rangle$ ::= $\langle$*ClassInstanceCreationExpression*$\rangle$

Removed production:

$\langle$*PrimaryNoNewArray*$\rangle$ ::= $\langle$*ClassInstanceCreationExpression*$\rangle$

Figure 6: Grammar transformations to keep Jedd grammar LALR(1)

2.3 Extracting Information from Relations

An important part of a language extension integrating relations into Java are facilities for extracting information from relations back to Java. Jedd provides two implementations of `java.util.Iterator` for iterating over the tuples of a relation. The first works on relations with a single attribute, and in each iteration returns the single object in each tuple. The second iterator works on relations of any size, and iterates over the tuples, returning each tuple as an array of objects. These iterators are used to implement a `toString()` method on relations, which is very useful for debugging Jedd programs. Without such a method, it would be very difficult to interpret the structure of a BDD to determine the relation it represents.

Jedd also provides a `size()` method, which returns the number of tuples in a relation. Jedd provides additional statistics about the BDD representations of relations as part of its profiling framework, which is described in Section 4.3.

3 Jedd Translator

We have implemented a translator which converts Jedd programs to Java programs. In Section 3.1, we discuss the key front-end issues, and in Section 3.2, we describe how the high-level relational operations are represented using lower-level BDD operations. A key part of the code generation algorithm is the physical domain assignment problem which is introduced in Section 3.3, and an algorithm based on SAT is provided in Section 3.4. In some cases, there exists no valid physical domain assignment, and in Section 3.5, we discuss how unsatisfiable core extraction is used to extract meaningful error messages.

3.1 Front-end

We implemented the Jedd to Java translator using Polyglot [15], a Java front-end intended for writing language extensions.

We used the Java grammar [8, ch. 19] as a starting point for a Jedd grammar. The productions that we added and removed to produce a grammar for Jedd are given in Figure 5. Non-terminals from the original Java grammar appear in italics. Unfortunately, Java's C roots make it difficult to write a clean LALR(1) grammar for it; some of the necessary workarounds are discussed in the introduction to the grammar itself. We found keeping an extension of the grammar LALR(1) to be difficult as well. Specifically, the grammar obtained by adding the productions in Figure 5 to the Java grammar is no longer LALR(1). The subexpressions in a join can be primaries, which in Java include class instance

$$\frac{a_i = a_j \Rightarrow i = j \quad a_i <: \texttt{jedd.Attribute}}{\texttt{new}\,\{o_1\texttt{=>}a_1,\ldots,o_n\texttt{=>}a_n\} : \{a_1,\ldots,a_n\}}\,[\text{Literal}]$$

$$\frac{x : T \quad a \in T \quad a <: \texttt{jedd.Attribute}}{(a\texttt{=>})x : T \setminus \{a\}}\,[\text{Project}]$$

$$\frac{x : T \quad a \in T \quad b \notin T \quad a,b <: \texttt{jedd.Attribute}}{(a\texttt{=>}b)x : T \setminus \{a\} \cup \{b\}}\,[\text{Rename}]$$

$$\frac{\begin{array}{c} x : T \quad a \in T \quad b,c \notin T \setminus \{a\} \\ b \neq c \quad a,b,c <: \texttt{jedd.Attribute} \end{array}}{(a\texttt{=>}b\ c)x : T \setminus \{a\} \cup \{b,c\}}\,[\text{Copy}]$$

$$\frac{x : T \quad y : T}{x \odot y : T \text{ where } \odot \in \{\texttt{\&}, \texttt{|}, \texttt{-}\}}\,[\text{SetOp}]$$

$$\frac{x : T \quad y : T \vee y \in \{\texttt{0B}, \texttt{1B}\}}{x \odot y : T \text{ where } \odot \in \{\texttt{=}, \texttt{\&=}, \texttt{|=}, \texttt{-=}\}}\,[\text{Assign}]$$

$$\frac{x : T \vee x \in \{\texttt{0B}, \texttt{1B}\} \quad y : T \vee y \in \{\texttt{0B}, \texttt{1B}\}}{x \odot y : \texttt{boolean} \text{ where } \odot \in \{\texttt{==}, \texttt{!=}\}}\,[\text{Compare}]$$

$$\frac{\begin{array}{c} x : T \quad y : U \quad U' = U \setminus \{b_1, \ldots b_n\} \quad T \cap U' = \emptyset \\ \{a_1,\ldots,a_n\} \subseteq T \quad \{b_1,\ldots,b_n\} \subseteq U \\ a_i = a_j \Rightarrow i = j \quad b_i = b_j \Rightarrow i = j \\ a_i, b_i <: \texttt{jedd.Attribute} \end{array}}{x\{a_1,\ldots,a_n\}\texttt{><}y\{b_1,\ldots,b_n\} : T \cup U'}\,[\text{Join}]$$

$$\frac{\begin{array}{c} x : T \quad y : U \quad T' \cap U' = \emptyset \\ T' = (T \setminus \{a_1,\ldots,a_n\}) \quad U' = (U \setminus \{b_1, \ldots b_n\}) \\ \{a_1,\ldots,a_n\} \subseteq T \quad \{b_1,\ldots,b_n\} \subseteq U \\ a_i = a_j \Rightarrow i = j \quad b_i = b_j \Rightarrow i = j \\ a_i, b_i <: \texttt{jedd.Attribute} \end{array}}{x\{a_1,\ldots,a_n\}\texttt{<>}y\{b_1,\ldots,b_n\} : T' \cup U'}\,[\text{Compose}]$$

Figure 7: Typing rules

creation expressions, which have an optional trailing class body enclosed in curly braces. A LALR(1) parser cannot distinguish this body from the attribute list following the subexpression in the join. Class instance creation expressions never have a relation type (which the join requires), so we can exclude them in this case. Therefore, prior to extending the Java grammar, we performed a series of language preserving transformations, removing class instance creation expressions from primaries, and adding them in all places where primaries can occur (except the join production that we added). These modifications are listed in Figure 6. The result is a LALR(1) grammar which extends Java in a natural way. The syntax and symbols for all operations are intuitive and easy to remember (the symbols for join and composition, `><` and `<>`, were inspired by $\bowtie$ and $\circ$, respectively, often used in relational database literature). Attribute manipulation operations (which change the type of expressions) use a cast-like syntax. No keywords and few new symbols were added.

Polyglot includes a complete semantic checker for Java. We extended this checker to infer the schemas of relational expressions from their subexpressions, and statically enforce the properties shown in Figure 7. The most important general properties are that no relation may have more than one instance of the same attribute, that operands of set and equality operations have compatible schemas, and that attributes mentioned in attribute manipulation, join, and composition expressions exist in the corresponding subexpressions.

3.2 Implementing Relational Operations in BDDs

In this section, we describe how relations are represented in BDDs, and how the relational operations are performed.

3.2.1 Representing Relations as BDDs

A BDD is a compact representation of a set of binary strings of a fixed length (or, equivalently, a function from $\{0,1\}^n$ to $\{0,1\}$). Jedd groups bit positions of these strings into *physical domains*. When a relation is represented in a BDD, each attribute is stored in a separate physical domain. The physical domains are defined and named by the user by implementing an interface included in the Jedd runtime library. The relative bit ordering of the various physical domains is also specified by the user. The assignment of the attributes of each relation to specific physical domains is subject to many constraints, and we leave the discussion of this important problem to Section 3.3. Once a physical domain assignment has been determined, Jedd ensures that each physical domain consists of enough bits to store the maximum number of objects that can be stored in each attribute assigned to it.

Each domain can convert objects in the domain to integers and vice versa. We use the binary representation of the integer to encode the object. To encode a tuple, we construct the BDD containing all strings such that for each attribute, the bits in the physical domain assigned to that attribute match the binary representation of the object stored in that attribute. Note that we have no requirement of the bits in physical domains not used by any attribute; these bits can be viewed to have a wildcard value. For example, suppose we want to encode the tuple {`o1=>A, o2=>B`}, where the binary representation of `o1` is 01, and the binary representation of `o2` is 10, `A` is assigned to the physical domain consisting of the first two bits, `B` is assigned to the physical domain consisting of the next two bits, and a third, unused physical domain exists, consisting of the last two bits. This tuple would be encoded by the BDD for the set of binary strings $\{0110??\} = \{011000, 011001, 011010, 011011\}$. Although this means that the BDD encoding of a single tuple can be a set of many strings, this does not affect the size of the BDD because BDDs represent such regular sets compactly. More specifically, the number of nodes in a BDD for a single tuple always equals the total number of bits in the physical domains used to encode the attributes.

The BDD for a relation of multiple tuples is simply the BDD for the union of the binary strings representing all the tuples. This means that the set operations on relations are implemented as the same operations on the sets of binary strings in the BDD, which are standard in BDD libraries. Similarly, relation equality is just BDD equality. However, for all these operations, the physical domain assignment must be the same for both their arguments.

3.2.2 Operations at the BDD level

Projection is implemented in BDDs using the universal quantification BDD operation on the physical domains assigned to the removed attributes. Conceptually, this operation takes all strings in the BDD, and creates new strings by replacing each bit of the physical domain with both 0 and 1. Therefore, each tuple in the original BDD will appear in the new BDD, but with a wildcard value for the physical domains projected away, indicating that they are not in use by the relation.

Attribute renaming requires no change to the underlying BDD. Only the mapping from attribute to physical domain needs to be updated, with the new attribute replacing the old.

To implement a ***join*** in BDDs, we must first carefully set up the physical domain assignment. The attributes being compared must be assigned to the same physical domains in the left and right relations. The remaining attributes must be assigned to physical domains not used by the other relation, or else their values will overwrite each other. Assuming we have such a physical domain assignment, the join itself is performed with an intersection operation on the sets of binary strings in the BDD. Since the attributes being compared are mapped to the same physical domain, the set intersection will find exactly those pairs from the two sets where these attributes match. The remaining attributes are stored in physical domains that are unused by the other relation, so they are represented there with a wildcard value. The set intersection of each object with the wildcard value just gives back the original object.

A ***composition*** is implemented in the same way as a join followed by a projection (set intersection followed by universal quantification), except that a special function of the BDD library is used that performs these two operations more efficiently in one step.

Due to the requirements of each operation on the physical domain assignment, it is sometimes necessary to change the physical domain assignment of a relation (that is, construct a different BDD representing the same relation, but under a different physical domain assignment). This is implemented using an operation called `replace` in BuDDy, and `SwapVariables` in CUDD, which constructs a BDD containing the same strings as the original BDD, but with the bits of each string permuted with a specified permutation. Jedd constructs the permutation required to move the bits of the old physical domain to the new physical domain, resulting in a BDD representing the same tuples, but in different physical domains.

3.3 Assigning Physical Domains to Attributes

One important problem when implementing algorithms using BDDs is deciding how to assign the attributes of each expression to physical domains of BDD variables. A programmer using a BDD library directly must perform this assignment by hand, and write the program directly in terms of the physical domains, rather than the attributes. For simple programs of several BDD expressions with two or three attributes, this is acceptable; however, for more complicated programs,[1] assigning a valid physical domain to each attribute of every subexpression is both tedious and error-prone. Furthermore, the physical domain assignment is closely related to the BDD variable ordering, and has significant effects on performance. Therefore, a tool like Jedd should relieve the programmer from having to specify physical domains for every expression, ensure that the physical domain assignment is a valid one, and provide a way to easily experiment with variations in the physical domain assignment without requiring all the code to be rewritten (as it would have to be if a BDD library were being used directly).

Jedd addresses these requirements in four ways. First, if the programmer does specify the physical domain assignment, Jedd checks that it is valid, and automatically inserts the required replace operations to implement the assignment. Although this is only a first step, it already makes programming less tedious and error-prone than when using a BDD library directly. Second, given a partial physical domain assignment for a small subset of expressions, Jedd contains an algorithm for automatically producing a reasonable assignment for the remaining expressions in the program. Should the programmer not be satisfied with specific parts of the automatically generated assignment, he can specify physical domains for those expressions explicitly, and re-run the automatic algorithm to find a reasonable assignment for the rest of the program. The physical domain assignment algorithm is discussed in detail in Section 3.4. Third, when the automated algorithm discovers that the programmer-specified partial physical domain assignment is inconsistent and no reasonable assignment exists, it reports the specific expression and attributes to which physical domains cannot be assigned. This makes it easy to locate the problem in a large project. Typically, the problem can be fixed by simply assigning the relevant attribute to a physical domain not already in use in the expression. Section 3.5 describes how Jedd determines the expression causing the problem. Fourth, Jedd provides a BDD profiler for visualizing the runtime costs of all BDD operations in terms of processing time and size and shape of the BDDs involved. This is particularly helpful in tuning the physical domain assignment and variable ordering for performance. The profiler is described in Section 4.3.

3.4 Physical Domain Assignment Algorithm

We call a physical domain assignment for a Jedd program *valid* if a BDD implementation using the assignment correctly computes the relational algebra expressions in the program. In order for a physical domain assignment to be valid, it is necessary and sufficient for it to satisfy the following constraints between attributes of expressions:

1. [conflict] All attributes of each expression must be assigned to *distinct* physical domains.
2. [equality] Each operation requires certain attributes of its operands to be assigned to the *same* physical domain, as described in Section 3.2.

A valid physical domain assignment can be found very easily. First, introduce a fresh physical domain for each attribute of each expression, satisfying the first requirement. Then, wrap each subexpression of a complex expression with a replace operation changing the physical domains to satisfy the second requirement. The resulting physical domain assignment is valid, but it requires many replace operations, slowing down program execution considerably.

We would like to minimize or at least reduce the number of replace operations, as well as give the programmer some control over where these operations take place. A convenient way to do this is to allow the programmer to specify physical domains for some small subset of expressions, and constrain the physical domain assignment not to contain any "unnecessary" replaces. This makes it possible for Jedd to construct a reasonable assignment with few replaces with very little input from the programmer, while giving the programmer the option to more completely specify a domain assignment for specific sections of the code.

[1]Our whole-program analyses contain 613 BDD subexpressions with a total of 1586 attributes.

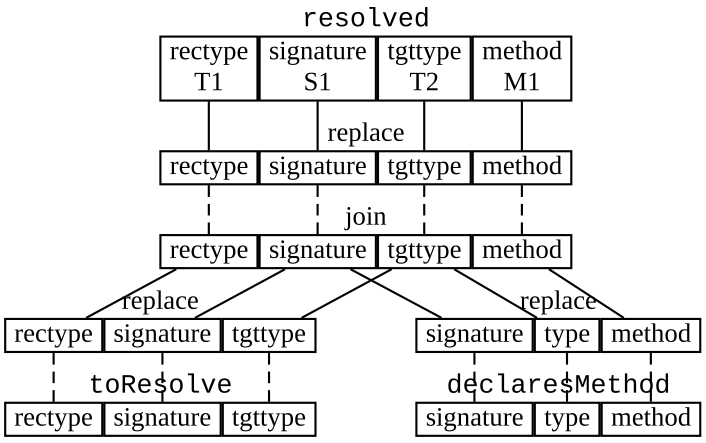


Figure 8: Example of physical domain assignment constraints

We need to formalize what we mean by "unnecessary" replaces. To do this, we first wrap all subexpressions with dummy replace operations as described above, so that the equality constraints can be satisfied. Then, for each attribute of each replace operation, we add an assignment edge from the attribute in the original subexpression to the attribute in the result of the replace. Intuitively, these assignment edges connect attributes which *should* be assigned to the same physical domain; if they are, the replace operation is unnecessary and can be removed. Because different replace operations have unpredictably different costs, we do not try to find an assignment having the minimum number of assignment edges with different physical domains; instead, we are satisfied with removing computation paths in which an attribute is replaced multiple times without reason. More precisely, we partition the graph formed by equality and assignment edges into connected components by potentially breaking some assignment edges, such that each component contains one attribute with a programmer-specified physical domain, and no conflict edge has both its endpoints in components with the same physical domain (or in the same component). Every attribute in a component is then assigned the same physical domain. This ensures that every replace operation has a reason, since replace operations only occur between attributes at the boundaries of components with different programmer-specified physical domains. Furthermore, this is consistent with the kind of behaviour the programmer likely expects: if an attribute is involved in a computation with other attributes for which physical domains have been specified, one expects it to be assigned to one of those domains.

The constraints produced from lines 6-7 of the example in Figure 4 are shown in Figure 8. Equality constraints are shown as solid lines and assignment constraints as dashed lines. Conflict constraints, which are not shown, are placed between all pairs of attributes within each expression. Replace operations have been wrapped around the subexpressions `toResolve` and `declaresMethod`, and around the entire join. In the absence of any other constraints, the graph would be split into four connected components (the first consisting of all rectype attributes, the second of all signature attributes, the third of all tgttype and type attributes, and the fourth of all method attributes), which would be assigned the physical domains T1, S2, T2, and M1, respectively. Since the input and output of each replace operation would then have the same physical domain assignment, no replacement would be necessary, so Jedd would remove them prior to generating Java code.

Proposition: The problem of partitioning the graph formed by equality and assignment edges into connected components by breaking assignment edges, such that each component contains one attribute with a programmer-specified physical domain, and no conflict edge has both its endpoints in components with the same physical domain (or the same component), is NP-complete.

Proof: The proof proceeds by a polynomial reduction of the NP-complete graph vertex k-colouring problem to the physical domain assignment problem. Given a graph to be k-coloured, we construct a Jedd program for which a physical domain assignment can be found if and only if the graph has a k-colouring.

Let $G = (V, E)$ be a graph for which a k-colouring is to be found. Construct a Jedd program from it as follows:

1. Declare attributes a, b, and c.

2. Declare $k+1$ physical domains $d_0 \dots d_k$.

3. For each vertex $v_i \in V$, declare a Jedd relation variable x_i with schema `<a, b>`, and no physical domains specified.

4. For each j with $1 \leq j \leq k$ and for each $v_i \in V$, add an assignment of a relation literal:
   x_i `= new {` `o1 => a:`d_j`, o2 => b:`d_0 `}`.

5. For each edge $(v_i, v_j) \in E$, add a statement computing x_i `{b} <> ((a=>c)` x_j`) {b}`.

Now, if G has a k-colouring assigning colour $C(i)$ to v_i, then the following partitioning satisfies the requirements: for each $v_i \in V$, create a connected component containing attribute a of x_i, attribute a of the literal with a assigned to $d_{C(i)}$, and attribute a or c of each composition having x_i as its left or right argument, respectively. For each composition with arguments x_i and x_j, an edge in the original graph ensures that x_i and x_j are assigned to distinct physical domains, so no conflict edges are violated.

Conversely, suppose the graph of equality and assignment edges formed from the Jedd program can be partitioned to satisfy the proposition. Then each attribute in the result of a composition must be in the same connected component as the corresponding argument of the composition (since this is the only equality/assignment path originating from it). This connected component must, in turn, contain an attribute with one of the physical domains $d_1 \ldots d_k$ assigned to it. Whenever an edge (v_i, v_j) exists in E, the conflict edge between the result attributes of the composition with x_i and x_j as arguments ensures that x_i and x_j are assigned to distinct physical domains. Therefore, the assignment of physical domains to attribute a of each x_i corresponds to a k-colouring of the v_i of G.

Therefore, G is k-colourable if and only if a partitioning satisfying the proposition exists for the constructed Jedd program, so the physical domain assignment problem is NP-hard.

Given a subset of the equality and assignment edges, it can be checked in polynomial time that:

1. all attributes (vertices) and equality edges are included in the subset,

2. each connected component of the graph formed by the subset of edges contains one attribute with a programmer-specified physical domain, and

3. no conflict edge has both its endpoints in components with the same physical domain (or the same component).

Therefore, the problem is in NP. Since it is also NP-hard, it is NP-complete. □

Several heuristics that we implemented to solve this NP-complete problem failed on common example programs. More importantly, an incomplete heuristic (which may fail to find a solution even when one exists) is undesirable for this problem. The case when Jedd fails to find a solution is precisely when the programmer very much wants to know whether a solution exists (and he should tediously look for it by hand) or does not exist (and he should modify the code so that a solution exists). Therefore, the potentially very high cost of an exhaustive search is justified, and our intuition told us that although the problem in general is NP-complete, typical instances would be relatively "easy" in some sense. However, we realized that implementing a smart exhaustive solver that would handle the easy cases efficiently would be difficult, and we would be duplicating much of the work that has been done on the boolean satisfiability (SAT) problem. We therefore encode the physical domain assignment problem as a SAT problem, and call a SAT solver to solve it for us.

Given a boolean formula over a set of variables, a SAT solver finds a truth assignment to those variables that makes the formula evaluate to true. We therefore encode the physical domain assignment problem into a boolean formula in such a way that we can recover a physical domain assignment from a truth assignment of its variables, and such that the formula evaluates to true exactly when the physical domain assignment satisfies our constraints. We construct the formula in conjunctive normal form because most SAT solvers require it, and it is easier to specify it directly in CNF than to construct an arbitrary formula and convert it to CNF later. A formula in CNF is a conjunction of disjunctions of literals, where each literal is a variable or a negated variable.

Let E be the set of all expressions of BDD type in the program. For each expression e, we use the notation e^a for attribute a of expression e. Let A be the set of all pairs e^a of expressions and attributes in the program. Let P be the set of all physical domains in the program.

The SAT formulation consists of two types of variables: attribute – physical domain variables, and flow path variables. A variable of the form $e^{a:p}$ indicates that attribute a of expression e is assigned physical domain p. To represent the notion of connected components in the SAT formula, we introduce *flow paths*, sequences of attributes of expressions with the following properties:

- the first attribute in the sequence is the only one with a programmer-specified physical domain,
- each consecutive pair of attributes on the flow path is connected by an equality or assignment edge,
- no attribute of an expression appears more than once on the flow path, and
- no other flow path ending with the same attribute exists whose attributes form a proper subset of the attributes of the flow path.

Intuitively, the flow paths represent, for each attribute of each expression, the shortest paths following equality and assignment edges to an attribute with a programmer-specified physical domain. We will require that at least one flow path ending at each attribute be active, indicating that the attribute, as well as all the attributes on the flow path, are in the same connected component. A variable of the form $\pi(e_0{}^{a_0:p_0}, e_1{}^{a_1}, \ldots, e_n{}^{a_n})$ indicates that the given flow path from attribute a_0 of e_0 to a_n of e_n is active; that is, all attributes along it are assigned physical domain p_0. We use Π to denote the set of all flow paths. The constraints are encoded in terms of these literals as follows.

1. Each attribute is assigned to some physical domain.

$$\bigwedge_{e^a \in A} \bigvee_{p \in P} e^{a:p}$$

2. No attribute is assigned to multiple physical domains.

$$\bigwedge_{e^a \in A} \bigwedge_{p,p' \in P, p \neq p'} \neg e^{a:p} \vee \neg e^{a:p'}$$

3. Any attribute with an explicitly specified physical domain is assigned that domain.

$$\bigwedge_{(e^a,p) \in SPECIFIED} e^{a:p}$$

4. For each conflict edge between e^a and $e'^{a'}$, a and a' must not be assigned to the same physical domain.

$$\bigwedge_{(e^a,e'^{a'}) \in CONFLICT} \bigwedge_{p \in P} \neg e^{a:p} \vee \neg e'^{a':p}$$

5. For each equality edge between e^a and $e'^{a'}$, a and a' are assigned the same physical domain.

$$\bigwedge_{(e^a,e'^{a'}) \in EQUALITY} \bigwedge_{p \in P} (e^{a:p} \vee \neg e'^{a':p}) \wedge (\neg e^{a:p} \vee e'^{a':p})$$

6. For each e^a, at least one flow path leading to it must be active.

$$\bigwedge_{e^a \in A} \bigvee_{\pi(e_0{}^{a_0:p_0}, e_1{}^{a_1}, \ldots, e^a) \in \Pi} \pi(e_0{}^{a_0:p_0}, e_1{}^{a_1}, \ldots, e^a)$$

7. When a flow path is active, all attributes on it are assigned the physical domain of the flow path.

$$\bigwedge_{\pi(e_0{}^{a_0:p_0}, e_1{}^{a_1}, \ldots, e_n{}^{a_n}) \in \Pi} \bigwedge_{0 \leq i \leq n} \neg\pi(e_0{}^{a_0:p_0}, e_1{}^{a_1}, \ldots, e_n{}^{a_n}) \vee e_i{}^{a_i:p_0}$$

3.5 Error Reporting

One challenge with using a black box such as a SAT solver in a compiler is in reporting errors to the user. When the SAT solver determines that no physical domain assignment exists, it only reports that the boolean formula is unsatisfiable. While this fact is useful for the programmer to know, it is not very helpful in determining the cause of the error.

To improve the error reporting, we took advantage of a new feature recently implemented in the zchaff SAT solver, unsatisfiable core extraction [21]. When the SAT solver determines that the boolean formula is unsatisfiable, it also outputs a small subset of the clauses (disjunctions) such that their conjunction is also unsatisfiable. Although the minimality of this core is not guaranteed, our experience has been that all the unsatisfiable cores found for the physical domain assignment problem were indeed minimal.

The physical domain assignment may not have a solution for one of two reasons. First, there may be an attribute of an expression with no path to any attribute for which a physical domain has been specified; that is, a component of the graph formed by equality and assignment edges may not have a physical domain specified for it. Jedd detects this case while constructing the input to the SAT solver, since it makes it impossible to construct the clause requiring at least one flow path leading to the attribute to be active (clause 6). Second, it may not be possible to partition the graph formed by equality and assignment edges in a way that respects all the conflict constraints. In this case, the following proposition gives us a way to report the source of the problem to the programmer.

Proposition: When the boolean formula produced for the physical domain assignment problem is unsatisfiable, every unsatisfiable core contains at least one clause of type 4 (conflict clause).

Proof: The key idea is that if clauses of type 4 are removed, the SAT formula ignores the requirement that conflict edges be respected by the partitioning of the graph. The proof then consists of two steps: showing that a partitioning can be found respecting all equality edges, and the technical steps to show that such a partitioning corresponds to a satisfying assignment for the remaining clauses of the SAT formula (which were explicitly designed to be satisfied exactly by such a partitioning).

That there exists a partitioning respecting all equality follows directly from the way in which equality edges are constructed. By construction, at least one of the endpoints of each equality edge is an attribute of a replace expression wrapping the argument of a relational operation, and has no other equality edges originating at it. These replace expressions are generated by Jedd and therefore have no programmer-specified physical domains. Each connected component of equality edges can therefore have no more than one vertex with a programmer-specified physical domain.

Let $G_E = (V = A, E = EQUALITY)$ be the graph with all attributes of expressions as vertices, and equality edges as edges. Let G' be the graph formed from G_E by adding assignment edges in the following way: as long as there is an assignment edge such that the connected components in G' of its endpoints do not contain attributes with different programmer-specified physical domains, add the edge to G', until no such edges are left.

Adding each edge preserves the property that no connected component of G' has more than one programmer-specified physical domain. In addition, every connected component of G' has at least one programmer-specified physical domain, because of the greedy construction of G'. For suppose to the contrary that G' contains a connected component c with no programmer-specified physical domain. In the original graph $G_{EA} = (V = A, E = EQUALITY \cup ASSIGNMENT)$ of all equality and assignment edges, there is a path from every attribute to some attribute with a programmer-specified physical domain. Therefore, there is such a path from an attribute of c. All edges of this path being in G' contradicts the definition of c, while an edge of this path missing from G' contradicts the definition of G'.

Having constructed a partitioning of the graph, we convert it to satisfying assignment of clauses 1, 2, 3, 5, 6, and 7 of the boolean formula. Let $e^{a:p}$ = true if and only if the connected component of e^a in G' has the physical domain p, and let $\pi({e_0}^{a_0:p_0}, {e_1}^{a_1}, \ldots, e^a)$ = true if and only if all the ${e_i}^{a_i}$ are in the same connected component of G'. This assignment trivially satisfies all clauses of types 1, 2, 3 and 7. Since G' contains all equality edges, it also satisfies all clauses of type 5.

To satisfy clauses of type 6, we must show that there is an active flow path from an attribute with programmer-specified physical domain to each attribute in the same connected component of G'. Let p be any such path in G', not necessarily an active flow path. If p violates the first requirement of being a flow path (that its first vertex is the only attribute with a programmer-specified physical domain), we can just take p to be the subpath of p starting at the last programmer-specified physical domain on p. p satisfies the second requirement, since all edges of G' are equality or assignment edges. If p violates the third requirement of being a flow path (that no vertex appears on it more than once), the section(s) between repeated vertices can be removed from it, yielding a p satisfying the requirement. Finally, the fourth requirement can only be violated by the existence of a flow path that is a subpath of p. Therefore, for each attribute in G', there is a flow path to it in G' from the attribute with programmer-specified physical domain of the same component. By construction of the assignment of active flow paths, this path is active. Therefore, all clauses of type 6 are satisfied.

We have constructed an assignment simultaneously satisfying all clauses of types 1, 2, 3, 5, 6, and 7. Therefore, every unsatisfiable core must contain a clause of type 4. □

It follows from the proposition that the small unsatisfiable core returned by the SAT solver will include at least one clause of type 4. From this clause, Jedd extracts the expression and the attributes to which physical domains could not be assigned, and even the physical domain(s) that were considered for assignment to the attributes. This information is reported to the programmer along with the position of the expression in the source file. The problem can be fixed by explicitly assigning a new physical domain to one of the attributes in the conflict constraint that cannot be satisfied.

4 Jedd Runtime

4.1 Backends

One of the benefits of expressing BDD algorithms in a language like Jedd is that we can execute these algorithms, without modification, using various BDD libraries as backends. This allows us to compare the performance of different backends on the same problem. In Jedd, we have already implemented interfaces to the BuDDy [9] and CUDD [17] libraries using JNI to call C code from Java, and we are experimenting with our own library written entirely in Java. Several researchers have suggested using zero-suppressed binary decision diagrams (ZDDs) [12] for our points-to analysis algorithms. We are therefore working on a backend for Jedd based on ZDDs, which will allow us to run all our algorithms using ZDDs without modification.

4.2 Memory Management Issues

All BDD libraries that we are aware of use a reference counting garbage collector to reclaim unused BDD nodes, since BDDs have a DAG structure. A disadvantage of this approach is that a programmer using the library in a C program is required to explicitly increment and decrement the reference count whenever BDDs are assigned or a reference to a BDD goes out of scope. In C++, it is possible to use overloaded assignment operators and destructors to relieve the programmer of much of this burden. Because this is not possible in Java, if we had implemented Jedd as a library rather than a language extension, we would have to require the programmer to explicitly manipulate reference counts like the C libraries do. This is yet another tedious and error-prone aspect of working with BDDs.

Since Jedd is an extension to the language, we can design it to update reference counts automatically, without any help from the programmer. For performance reasons, it is particularly important that the reference count be decremented as soon as possible after a reference becomes dead. When dead nodes are not freed in a garbage collection, fewer nodes remain for future computation, so garbage collection is required more frequently. In addition, BDD libraries use a cache to speed up the basic operations on nodes. Large numbers of unfreed obsolete nodes may pollute this cache. In general, we cannot rely solely on the Java garbage collector to determine when relations are unreachable, particularly short-lived temporary relations. This is because unlike allocations of Java objects, an allocation of a BDD node will not trigger a Java garbage collection when no more memory is available. It is very possible to allocate many large temporary BDDs in several iterations of a loop and have the BDD library run out of memory without a Java garbage collection ever being triggered.

A BDD can become dead in four ways. First, it may be the result of a subexpression of an expression becoming unreachable after the outer expression is evaluated. Second, it may be stored in a local variable or field, and it may be overwritten by another BDD. Third, the BDD may be stored in a local variable which goes out of scope. Fourth, the BDD may be stored in a field, and the object containing the field may become unreachable. For temporary values, the first two cases are the most common and therefore the most important.

To handle the first case, we implement the convention that each BDD operation decrements the reference count of its arguments and increments the reference count of its result before returning it. This convention is partly imposed by the requirement of the BDD libraries that any BDDs passed to library functions have non-zero reference counts.

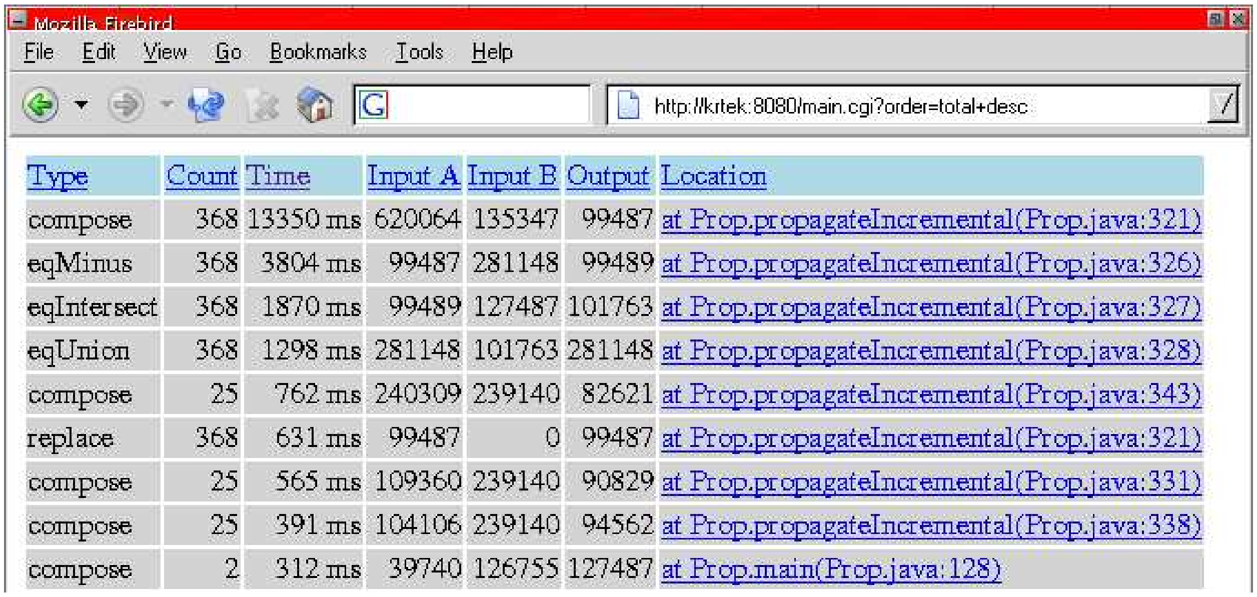
For a clean implementation of the remaining cases, we create a relation container object for each local variable and field. In the generated Java code, the variable/field points to its relation container throughout its lifetime; this is enforced by making the variable/field final. The BDD itself is stored as a private field in the relation container, and can be updated only through an assignment method which also updates the reference counts. This guarantees that in the second case, a BDD being overwritten has its reference count decremented immediately. To handle the third and fourth cases, the relation container object also decrements the reference count of any BDD stored in it in its finalizer, which is called when the relation container is garbage collected. In the fourth case (a field becoming dead), this happens in

Analysis	Relation		Phys.	Number of Constraints			SAT Problem			Solving
Component	Exprs.	Attrs.	Doms.	Conflict	Equality	Assign.	Variables	Clauses	Literals	Time (s)
Virtual Calls	46	127	5	184	173	70	1298	5600	11627	0.016
Hierarchy	172	344	5	242	442	143	3711	18140	37764	0.062
Points-to Analysis	247	561	8	637	802	259	7997	44405	93052	0.161
Side-effect Analysis	68	237	9	484	282	108	4441	41772	86482	0.165
Call Graph	89	340	8	929	442	187	7043	96514	197865	0.284
All 5 combined	613	1586	10	4902	2141	767	31083	273986	568597	4.607

Table I: Size of physical domain assignment problem

the same garbage collection [2] as the one in which the object containing the field becomes unreachable, which is the earliest time that it is safe to decrement the reference count. For the third case (a variable going out of scope), this ensures that the reference count will eventually be decremented, but this may be a significant amount of time after the variable goes out of scope. To improve on this, we perform a static liveness analysis on all relation variables, and at each point where a variable may become dead, we decrement the reference count of any BDD it may contain and remove the BDD from the container. In the face of exceptional interprocedural control flow, this is not always possible. We assume such control flow to be unusual, and fall back on the finalizer to decrement the reference count in such cases.

To summarize, Jedd manages BDD reference counts automatically without any help from the programmer. In all four cases, it frees BDDs as soon as it becomes safe to do so, so its performance should be no worse than that of a hand-coded reference counting solution.


Mozilla Firebird
File Edit View Go Bookmarks Tools Help
http://kitek:8080/main.cgi?order=total+desc

Type	Count	Time	Input A	Input B	Output	Location
compose	368	13350 ms	620064	135347	99487	at Prop.propagateIncremental(Prop.java:321)
eqMinus	368	3804 ms	99487	281148	99489	at Prop.propagateIncremental(Prop.java:326)
eqIntersect	368	1870 ms	99489	127487	101763	at Prop.propagateIncremental(Prop.java:327)
eqUnion	368	1298 ms	281148	101763	281148	at Prop.propagateIncremental(Prop.java:328)
compose	25	762 ms	240309	239140	82621	at Prop.propagateIncremental(Prop.java:343)
replace	368	631 ms	99487	0	99487	at Prop.propagateIncremental(Prop.java:321)
compose	25	565 ms	109360	239140	90829	at Prop.propagateIncremental(Prop.java:331)
compose	25	391 ms	104106	239140	94562	at Prop.propagateIncremental(Prop.java:338)
compose	2	312 ms	39740	126755	127487	at Prop.main(Prop.java:128)

Figure 9: Overall profile view

4.3 Profiler

A common problem when tuning any algorithm using BDDs is choosing an efficient *variable ordering*, the relative order of the individual bits of the physical domains. In complicated programs with many relations and attributes, a related problem is tuning the physical domain assignment, and the replace operations which it dictates. Specifically, we

[2] Here, we assume that the garbage collector collects all unreachable objects in each collection. However, even when this is not true in general, such as in a generational collector, it is very likely that the object containing the field and the relation container will be reclaimed in the same collection, since they are allocated close together: the latter is allocated in the constructor of the former.

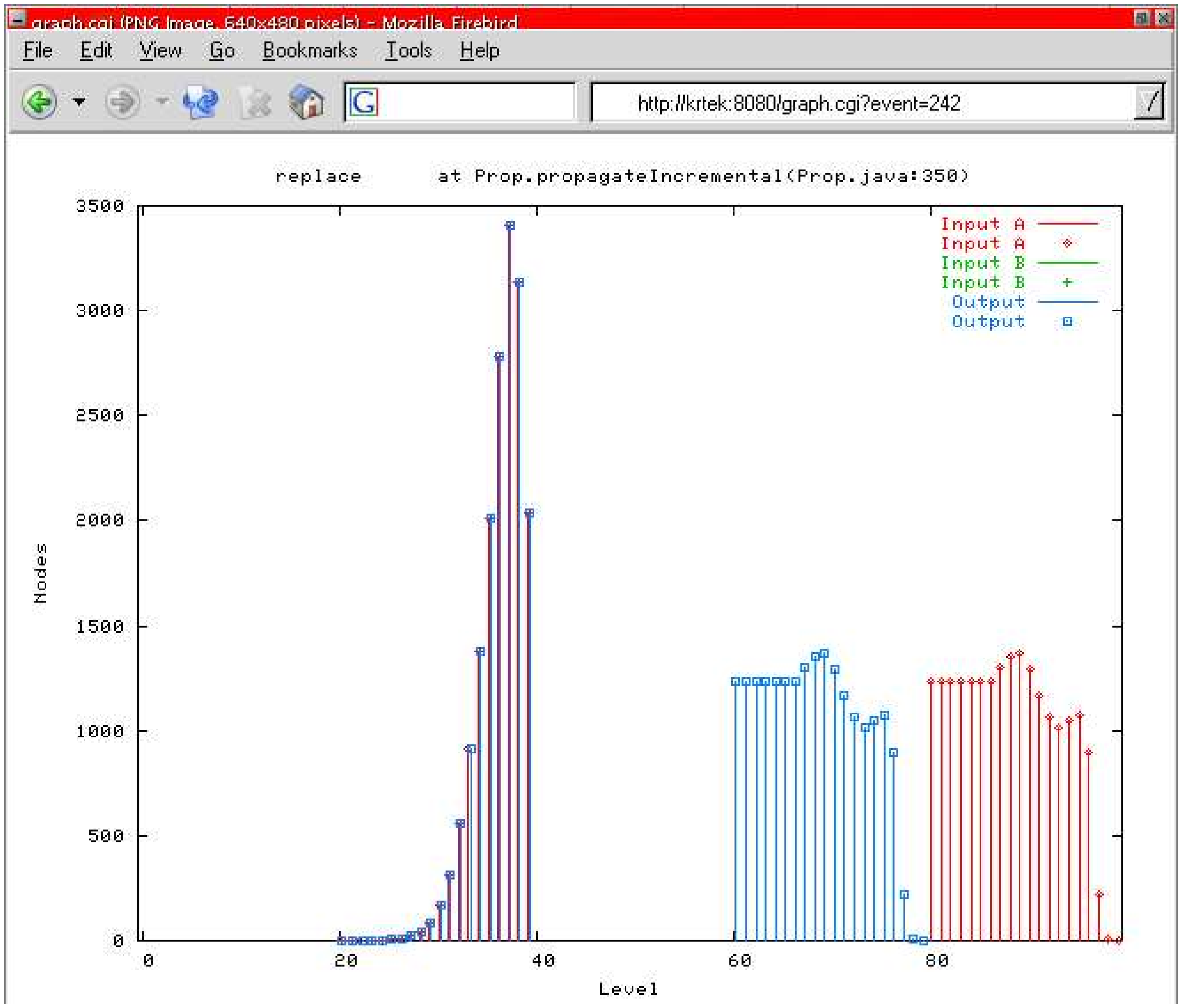


Figure 10: Graphical represenation of BDD in replace operation

are interested in removing those replace operations which are particularly expensive by modifying the physical domain assignment to make them unnecessary. For these tuning tasks, we need some insight into the runtime behaviour of our program. In particular, we want to know which operations are expensive in terms of time and BDD size (and therefore space), in order to either remove them, or make them cheaper by modifying the variable ordering. For tuning the variable ordering, knowing the shape of the BDDs involved in the operation is also useful. The shape of a BDD is the number of nodes at each level (testing each variable) of the BDD.

In the code generated by Jedd, relational operations are implemented as calls into the Jedd runtime library. The runtime library optionally makes calls to a profiler which records, for each operation, the time taken and the number of nodes and shape of the operand and result BDDs. This information is written out as an SQL file to be loaded into a database, which provides a flexible data store on which arbitrary queries can be performed to present the data to the user. Jedd also includes CGI scripts to provide access to the profiling data through a web browser. We use SQLite for the database and thttpd for the web server because of their ease of installation, but in principle, any SQL database and CGI-capable web server should work. The overall profile view shows, for each relational operation in the program, the number of times it was executed, the total time taken, and the maximum size of the BDDs involved (see Figure 9). Clicking on an operation brings up a detailed view with a line of information for each time the operation was executed. Clicking on a specific execution of the operation generates a graphical representation of the shape of the BDDs involved in the operation. Figure 10 shows an example of this graphical representation for a typical replace operation. In this case, the relation consists of two attributes, the first mapped to the physical domain ranging from levels 20 to 39 of the BDD, and the second being moved from the physical domain at levels 80 to 99 of the BDD to a different physical domain at levels 60 to 79.

5 Experience with Jedd

We have implemented in Jedd several test examples, our BDD points-to analysis algorithm [4], and a collection of interrelated whole-program analyses. Without Jedd, the latter would not have been feasible, since it would require us to assign physical domains by hand to the attributes of 613 subexpressions, with no automated way to verify that we had not made mistakes; in fact, we initially tried such an approach, and quickly gave up. Even the relatively short points-to analysis algorithm becomes much clearer when expressed using attributes rather than physical domains directly, and without the clutter of low-level replace operations. In general, we wrote the whole-program analyses without specifying any physical domains at all, and when it came time to compile, we assigned just enough attributes to physical domains to allow the physical domain assignment algorithm to assign the rest. In this process, Jedd's error reporting pointed us directly to the expressions that needed to have physical domains assigned by hand. The analyses themselves were easy to implement compared to pure Java implementations, mainly thanks to the compact representation provided by BDDs. For instance, the Java version of the side-effect analysis consists of 803 non-comment lines of code, mostly implementing data structures to compactly represent the large, highly redundant sets of side effects. In contrast, the Jedd version is only 124 lines. For now, this is just preliminary experience, but we hope that Jedd will enable the development of many other BDD-based program analyses.

We have found the zchaff SAT solver [13] to be more than fast enough for solving the domain assignment problem, even for significant programs such as the combination of all five program analyses, as shown in Table I. The first section of the table shows the number of relational expressions, attributes in these expressions, and physical domains. The second section lists the number of each type of constraint in the physical domain assignment problem. The third section gives the number of distinct variables, clauses, and literals in the resulting SAT formula. Finally, the fourth section shows the time taken by zchaff to parse and solve the formula on a 1833 MHz Athlon with 512 MB of RAM. To put the times into perspective, a complete build of Soot takes 5 minutes on the same machine, so 4.6 seconds to assign physical domains is very acceptable. The SAT encoding of the physical domain assignment problem was designed to be easy to understand rather than compact; it could easily be made smaller if the SAT solver ever became a bottleneck.

To measure the runtime overhead of Jedd compared to using a BDD library directly in C++ [4], we timed the C++ and Jedd versions of our points-to analysis algorithm on five benchmarks. Both versions used the BuDDy library as the backend. The timings are shown in Table II. The overhead varied from 0.5% to 4%, which we attribute to having to have the Java VM in memory, and to the internal Java threads that run even when not executing Java code.

Benchmark	Std. lib. version	C++	Jedd
javac	1.1.8	3.4 s	3.5 s
compress	1.3.1	21.7 s	22.4 s
javac	1.3.1	25.3 s	26.3 s
sablecc	1.3.1	25.4 s	26.1 s
jedit	1.3.1	41.1 s	41.3 s

Table II: Running time comparison of hand-coded C++ [4] and Jedd points-to analysis

6 Related Work

The relational data model based on relational algebra was proposed by Codd [7], and has since been used for many applications, particularly as the basis of relational databases.

Jedd is built on top of the BuDDy [9] and CUDD [17] BDD libraries, which provide a low-level interface to a BDD implementation. On top of its lowest-level interface, BuDDy implements *finite domain blocks*, a convenient way to group together BDD variables, much like the physical domains in Jedd. GBDD [14] is a C++ library providing an even higher-level relation-based abstraction.

Several small interactive languages have been developed for experimenting with BDDs. One example is BEM-II [11], designed for manipulating Arithmetic BDDs and solving 0-1 integer programming problems. Another is IBEN [2], which provides a direct user interface to BuDDy, as well as BDD visualization facilities.

The JNI interface allows Java code to use BDD libraries written in C through specially written wrappers. We have found it very convenient to use the wrapper generator Swig [1] to automatically generate these wrappers for us. However, others have chosen to write such wrappers by hand, resulting in JBDD [18], a Java interface to both BuDDy and CUDD, later extended and renamed JavaBDD [20]. Unlike Jedd, these JNI wrappers provide no abstraction over the underlying BDD libraries. They simply allow the library functions to be called from Java.

The RELVIEW system is an interactive manipulator of binary relations (each having exactly two attributes). One of its backends stores relations in BDDs [3].

Languages with support for binary relations have been around since the days of SETL [16] and are too numerous to list. Codd-style relations have also appeared in many languages, particularly those designed for interfacing with relational databases. Examples include the `<bigwig>` project [5] and recent work on extensions to C# [10]. In contrast to these languages whose primary goal is to provide access to relations, the primary focus of Jedd is to enable program analysis developers to exploit the compact data representation provided by BDDs, using relations as an abstraction to make programming with BDDs manageable.

7 Conclusions and Future Work

We have introduced Jedd, a high-level language extension to Java for expressing set-based algorithms so that they can be implemented using BDDs. The motivation for designing the language was to provide a convenient way of specifying program analyses so that they could be efficiently solved using existing BDD packages.

The approach presented is one of designing an appropriate language abstraction which: (1) provides the programmer with the correct abstraction of the problem, in this case a form of relations; (2) provides as much static type support as possible; (3) exposes only as much low-level detail as required (the programmer need only provide some of the key physical domain assignments); and (4) fits naturally as an extension of a general purpose programming language, Java.

Based on this language, we have defined and implemented a translator to generate Java code and an associated runtime system. Key parts of the translator are: (1) the high-level relational operations are translated into low-level BDD operations which can be provided by a variety of backend solvers; (2) the translator leverages the power of existing SAT solvers to automatically provide a complete assignment of attributes to physical domains (or provides a meaningful error message if no such assignment exists); (3) automatically supports a reference-count-based approach to memory management at the Java level, compatible with the approaches taken in the C-based solvers; and (4) provides support for debugging and profiling of the BDD-based operations.

We have used our system to program five key program analysis modules in the Soot compiler framework [19] and have found that it allows us to write compact programs which can be compiled in reasonable time, and that generated BDD-based code is about as efficient as the BDD solvers we coded by hand.

Our next steps are to experiment with more analyses written in Jedd and to integrate the Jedd-based analyses into the main Soot development trunk.[3] Another challenge will be to make Jedd compatible with generics when Java 1.5 is released.

Acknowledgments

This work was supported, in part, by NSERC.

[3]We are also planning on releasing a public version of Jedd in time for PLDI 2004.

References

[1] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, July 1996.

[2] Gerd Behrmann. The interactive BDD environment. `http://iben.sourceforge.net/`.

[3] R. Berghammer, B. Leoniuk, and U. Milanese. Implementation of relational algebra using binary decision diagrams. In *6th International Conference RelMiCS 2001*, volume 2561 of *LNCS*, pages 241–257, December 2002.

[4] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Inplementation*, pages 103–114. ACM Press, 2003.

[5] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Trans. Inter. Tech.*, 2(2):79–114, 2002.

[6] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[7] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[8] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[9] Jørn Lind-Nielsen. BuDDy, A Binary Decision Diagram Package. Department of Information Technology, Technical University of Denmark, `http://www.itu.dk/research/buddy/`.

[10] Erik Meijer and Wolfram Schulte. Unifying tables, objects, and documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages*, August 2003.

[11] S. Minato and F. Somenzi. Arithmetic boolean expression manipulator using BDDs. *Formal Methods in System Design*, 10(2/3):221–242, 1997.

[12] Shinichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277. ACM Press, 1993.

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th conference on Design automation*, pages 530–535. ACM Press, 2001.

[14] Marcus Nilsson. GBDD – A package for representing relations with BDDs. `http://user.it.uu.se/-˜marcusn/projects/rmc/docs/gbdd/index.html`.

[15] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *12th International Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 138–152, 2003.

[16] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets - an Introduction to Setl*. Springer, New York, 1986.

[17] Fabio Somenzi. CUDD: CU Decision Diagram Package. Department of Electrical and Computer Engineering, University of Colorado at Boulder,
`http://vlsi.colorado.edu/˜fabio/CUDD/`.

[18] Arash Vahidi. Arash's Java interface to BDDs.
`http://www.chl.chalmers.se/˜vahidi/bdd/bdd.html`.

[19] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, volume 1781 of *LNCS*, pages 18–34, 2000.

[20] John Whaley. JavaBDD – Java binary decision diagram library. `http://javabdd.sourceforge.net/`.

[21] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, May 2003.