5-20-2012

Improving the JMLE Tool's Constraint Solving on Sets

Katherine Marie Veil
Dickinson College

Improving the jmle Tool's Constraint Solving on Sets

By

Katherine Veil

Submitted in partial fulfillment of the requirements
for departmental honors in Computer Science
Dickinson College, 2011-2012

Tim Wahls, Advisor
Grant Braught, Reader
David Richeson, Reader

May 15, 2012

The Department of Mathematics and Computer Science at Dickinson College hereby accepts this senior honors thesis by Katherine Veil, and awards departmental honors in Computer Science.

Tim Wahls (Advisor) Date

Grant Braught (Reader) Date

David Richeson (Reader) Date

Tim Wahls (Department chairperson) Date

Department of Mathematics and Computer Science
Dickinson College

May 15, 2012

Abstract

Improving the jmle Tool's Constraint Solving on Sets

by

Katherine Veil

The jmle tool executes JML specifications (formal specifications for Java classes) by translating them to constraint programs. This tool has useful applications such as prototyping and specification testing; however, it uses backtracking, which takes exponential time. To decrease jmle's running time, we can add new constraint handling rules that remove elements from the domains of variables, thus reducing the search space for backtracking. The goal of our project is to improve the running time of jmle by adding rules concerning properties of and relationships between sets. For example, we have treated the size of a set as a finite domain variable instead of as a single integer, so infeasible areas of the search space can be more quickly eliminated. We have also bounded sets from both above and below with collections of their elements. The addition of our rules has dramatically decreased the running time of some specifications in jmle. However, each new rule adds some overhead in running time, so it is important to add rules that are beneficial enough to outweigh this cost. We conclude that the careful addition of constraint handling rules can have a positive impact on jmle's performance.

Acknowledgments

I would like to thank Professor Tim Wahls, my advisor, for being a constant source of help and guidance throughout this research project. I would also like to thank Professor Grant Braught and Professor David Richeson, my readers, for their feedback and help shaping this paper.

Contents

1. Introduction

Over the course of the year I have investigated how to improve the jmle tool's constraint solving abilities, particularly in regard to constraints on sets. The jmle tool executes JML specifications (formal specifications for Java classes) by translating them to constraint programs. This tool has several useful applications such as prototyping and specification testing, so it is advantageous to try to improve its performance. Constraint satisfaction in jmle is done by a combination of domain pruning and backtracking, which takes exponential time. The domain pruning currently performed decreases the search space and improves performance; however, more pruning is always desirable because it decreases the necessary amount of backtracking. To achieve more pruning we can improve jmle's constraints on properties of sets. Our two main approaches have been treating sizes of sets as finite domain variables instead of single integers, and creating a constraint to bound sets from above and below with collections of their elements.

2. Background

To begin we will give an overview of constraint programming, and then we will proceed to the Java Constraint Kit (JCK), a set of Java libraries to handle constraint solving. Next we will cover the Java Modeling Language (JML), a formal specification language for Java, before describing the jmle tool, which executes JML specifications using a constraint solver. We will then discuss finite domain variables and constraints as well as bounds constraints, and finish by relating these ideas to constraints on sets.

2.1. **Constraint programming.**

Constraint programming is a type of declarative programming in which the programmer uses constraints to specify characteristics of and relations between variables. It is different from imperative programming (for example, in the C-based languages) in that the programmer specifies the result, not the procedure. In a constraint satisfaction problem (CSP), a set of variables must be assigned values that satisfy a set of constraints. A constraint solver keeps a list, called the constraint store, of constraints which must be true simultaneously. It uses a set of constraint handling rules to simplify the constraints on each variable, adding and removing constraints from the store. Frequently search is needed to assign values to each variable. This search is typically implemented as backtracking; variables are assigned values until the constraint store becomes false, indicating a contradiction. At this point the most recently assigned variable is given a new value and the search continues. Backtracking has exponential time complexity, so solving a constraint program can take a considerable amount of time. A CSP is solved when a

variable assignment that satisfies all the constraints is found; if there is no variable assignment that can satisfy all of the constraints, the problem is not satisfiable (Marriott & Stuckey, 1998).

2.2. **Java Constraint Kit.**

The Java Constraint Kit, or JCK, is a set of constraint handling libraries for Java. It consists of three parts: a high-level syntax for writing constraint handling rules, an interactive GUI for visualizing constraint solving, and a search engine (Abdennadher, Kramer, Saft, & Schmauss, 2002). Using JCK it is possible to translate a constraint program into pure Java. This makes it much simpler to integrate constraint programming components into a larger Java application; there is no need for the user to install other constraint handling applications or to fit executables from different languages together.

The constraint handling rules in JCK, called Java Constraint Handling Rules (JCHR), have their own syntax, specified in Abdennadher et al. (2002). There are three parts to each rule: the guard, the head, and the body. The guard is a condition that must be true for the rule to execute, or fire, written as `if(`*condition*`)`. The head is a collection of constraints that must be in the store for the rule to fire, and the body is a collection of constraints that will enter the store as the rule fires.

There are three types of constraint handling rules: simplification, simpagation, and propagation. In a simplification rule, all head constraints are removed from the store and replaced by the body constraints. Figure 2.1 is an example of a simplification rule. A simpagation rule looks like a simplification rule, except that it includes the special symbol `&/&` between two head constraints. Head constraints before the symbol are kept in the constraint store, and head constraints after the symbol are removed and replaced by the

```
if (org.jmlspecs.jmlexec.runtime.ObjUtil.ground(N1) &&
    org.jmlspecs.jmlexec.runtime.ObjUtil.ground(N2))
   { mult(N1, N2, N3) } <=>
   { N3 = org.jmlspecs.jmlexec.runtime.MyNumber.mult(N1, N2) } mult1;
```

FIGURE 2.1. Example simplification rule with the JCHR syntax. The three variables N1, N2, and N3 are numerical, and the constraint `mult(N1, N2, N3)` means that N1 × N2 = N3. If N1 and N2 are ground (meaning we know their values), then the `mult` constraint can be removed and N3's value can be computed by a static method.

body constraints. The third type of rule is a propagation rule, where the head constraints remain in the constraint store and are joined by the body constraints. Figure 2.2 is an example of a propagation rule. Propagation rules are dangerous because constraints are added to the store, but no constraints are removed; hence, the same rule could potentially fire repeatedly, filling the store with redundant constraints. The type of rule is indicated by the symbol between the head and body of the rule (`<=>` for simplification and simpagation, and `==>` for propagation).

The guard and body of a rule may contain calls to static methods in Java classes. There are several utility classes in jmle containing static methods for use in constraint handling rules. These methods must be called with their fully qualified name, which sometimes

```
if (org.jmlspecs.jmlexec.runtime.JMLTool.isSetMapOrRel(St1))
   { intersection(JC1, JC2, St1, JC3) } ==>
   { isSubset(JC3, JC1, St1) && isSubset(JC3, JC2, St1) } intersection1;
```

FIGURE 2.2. Example propagation rule with the JCHR syntax. The three variables JC1, JC2, and JC3 are sets of type St1, where JC1 ∩ JC2 = JC3. The method in the guard, `isSetMapOrRel(St1)`, verifies that St1 is an appropriate type. Every time this rule fires, two `isSubset` constraints are added to the store: JC3 ⊆ JC1 and JC3 ⊆ JC2. This example rule fires repeatedly and fills up the constraint store, so it is no longer in use in our rule set.

can make the rules difficult to read. The constraint handling rules in Figures 2.1 and 2.2 exhibit calls to static methods.

2.3. **Java Modeling Language.**

The Java Modeling Language, or JML, is a formal specification language for Java. By providing preconditions and postconditions in JML syntax, programmers can guarantee the behavior of their code in a high-level but unambiguous way (Chalin, Kiniry, Leavens, & Poll, 2006). This is an example of the Design by Contract philosophy, where rights and responsibilities of pieces of software are stated explicitly to ensure program correctness (Hunt & Thomas, 2000).

Figure 2.3 is an example of a square root function specified in JML. The specification is contained in the special `/*@...*/` comments above the method declaration. The first line of the specification declares that this method may not have side effects; it may only assign to its local variables. The second line is the precondition, requiring in this case that the parameter is nonnegative. The third line is the postcondition, which ensures that the return value truly is the square root of the parameter.

```
/*@ assignable \nothing;
    requires x >= 0;
    ensures \result * \result == x;
  */
public static double sqrt(double x) {
    // compute square root
}
```

FIGURE 2.3. Example JML specification for a square root function.

2.4. **jmle.**

jmle is a tool for executing JML specifications. It works by translating a JML specification into a constraint program. The JCK-generated Java constraint solver runs the constraint program, creating a Java implementation of the original JML specification. The ability to execute a specification is useful for several reasons: it aids understanding and development of the specification, helps the developer verify that it was written correctly and completely, and can be used as a prototype or a test oracle (Krause & Wahls, 2006). There have been many recent performance and usability enhancements to jmle (Keating, Kostrubiak, & Wahls, 2011). Even so, it is still in need of optimization. One way to improve performance is to prune the search space through which the solver must backtrack, which is the goal of this project.

2.5. **Finite domain variables and constraints.**

We will use finite domain variables in our effort to reduce the search space for backtracking. A finite domain variable is a variable that can take on values in a finite, known set, typically a subset of the integers (Marriott & Stuckey, 1998). Finite domain constraints use relationships between finite domain variables to remove possible values from their domains. This is useful because it decreases the search space for the constraint solver, and therefore can reduce the amount of backtracking that occurs.

There are many techniques used to solve constraints over finite domains. If the variables in constraint relations are integers, we can use a method called bounds consistency to reduce their domains during backtracking (Marriott & Stuckey, 1998). In bounds consistency, the minimum and maximum of each related domain is checked, possibly narrowing the values each variable could take. For example, with the inequality $x < y$

where $x \in [4..7]$ and $y \in [1..6]$, we can prune the domains to $x \in [4..5]$ and $y \in [5..6]$, reducing the search space significantly. If an unsatisfiable state is reached, such as $x < y$ where $x \in [4..7]$ and $y \in [1..3]$, we can immediately backtrack from this point instead of continuing down the failing branch of the search tree.

There exist other methods, such as arc and node consistency, which are polynomial time but only work on constraints of one or two variables. Unfortunately hyper-arc consistency, their generalization for 3 or more variables, is NP-hard and will not be described here (Marriott & Stuckey, 1998, p. 97).

Arc, node, and bounds consistency are already implemented in jmle (Keating et al., 2011). In my project I aim to extend the usefulness of finite domain constraints beyond simply finite domain variables by to using them in constraints on sets.

2.6. **Bounds constraints on sets.**

We have looked to the Mozart-Oz constraint system (Haridi & Franzen, 2008) for inspiration of how to apply finite domain constraints to properties of sets. In Mozart-Oz, set constraints are of three different forms: lower bound, upper bound, or cardinality. A lower bound constraint stores a subset of the set, collecting all elements that are known to be in the set. An upper bound constraint does the opposite; it stores a superset, collecting all elements that could possibly be in the set. A cardinality constraint treats the size of the set as a finite domain variable, storing the range of the possible sizes of the set (Müller, 2008a). We have created a constraint in jmle called `setBound` that combines these three elements; it relates a set, its lower bound, its upper bound, and its size. This constraint will be discussed in more detail in Section 5.2.

As an illustration, consider an example of lower bound and cardinality constraints working together. Let $A \subseteq B$, where $\{3, 5, 6\} \subseteq A$, $\{1, 2, 4, 6\} \subseteq B$, and $|B| \in [4..6]$. Using the fact that A is a subset of B, we can update B's lower bound to $\{1, 2, 3, 4, 5, 6\}$. Now we know that B must have at least 6 elements, while the cardinality constraint tells us that B can have at most 6 elements. Thus B has exactly 6 elements, and the set has become known.

3. Related Work

Five papers with some bearing on my work are listed in this section. The first of these describes recent work done on the jmle tool; as my project is a continuation of this work, this paper is highly relevant. The remaining four papers give descriptions of tools similar to jmle, since investigating similar systems may give better insight on how jmle works. The final two of these have a much higher level of detail on the actual constraint solving, so they provide a lower-level view of the translation process.

3.1. **Enhancing the jmle Tool.**

This paper by Michael Keating, Adrian Kostrubiak, and Tim Wahls (2011) explains and motivates jmle and gives the appropriate background. It outlines performance enhancements to the jmle tool–finite domain arc, bounds, and element consistency and an ordering heuristic–and gives data to support their efficacy. It also discusses usability enhancements, such as added support for Java 6 features like generics and enumerations and an Eclipse-based graphical user interface. My project is an extension of this work, as I also aim to improve the jmle tool. The finite domain consistency techniques implemented here are the foundation for my work on finite domain constraints.

3.2. **Symbolic Animation of JML Specifications.**

This paper by Fabrice Bouquet, Frédéric Dadeau, Bruno Legeard, and M. Utting (2005) describes BZ-Testing-Tools, a constraint-based tool for executing formal specifications. Its stated goal is to "use constrained animation to simulate the execution of the formal model and ensure its validity." The paper gives an overview of how the tool's constraint solving works and how it can interact with JML. Of particular interest is a section on

properties checking and how to check for satisfiable and valid predicates. This tool is similar to jmle in key ways, and thus is useful to mention here.

3.3. **Compiling Formal Specifications to Oz Programs.**

In this paper, Tim Wahls (2005) describes a method for executing formal specifications in SPECS-C++ by translating them to Oz declarative programs. Although the source and target languages are different, the system is very similar to parts of jmle. One point of particular interest is a discussion of a distribution strategy for considering finite domain variables. Using a distribution strategy determines the order in which the variables are seen during search; a clever strategy will examine the variables with the smallest domains first in order to traverse the smallest portion of the search tree. This "first-fail" approach is already implemented in jmle, so finite domain variables are already handled efficiently by the constraint solver.

3.4. **A Computation Model for Z Based on Concurrent Constraint Resolution.**

In this paper, Wolfgang Grieskamp (2000) gives a detailed description of the way a constraint solver uses rules to resolve constraints. Grieskamp reduces Z, a specification language, to a small symbolic language called μZ. He then applies concurrent constraint resolution techniques to execute the Z specification. This paper includes many implementation details, including a discussion of the Z $\rightarrow$ μZ mapping, set representation, and reduction and resolution rules. At a high level, this method is similar to jmle in that it makes a specification language executable. This paper also gives insight on the inner workings of a constraint solver and representations of rules.

3.5. **CLPS-B: A Constraint Solver for B.**

This paper by Fabrice Bouquet, Bruno Legeard, and Fabien Peureux (2002) describes a constraint solver, called CLPS-B, used to execute formal specifications for B. In a B model the formal specification takes the form of an abstract machine and is not tied to any one programming language. Domains in CLPS-B are defined differently, so the authors create and define new terminology and redefine the concepts of consistency and satisfiability. Using these definitions they rigorously prove properties of their solver. It is interesting to see such a formal approach applied to a constraint solver since jmle has not yet been analyzed in this way.

4. Strategy

This section contains a discussion at a conceptual level of our strategies for improving jmle. These include treating sizes of sets as finite domain variables, storing lower and upper bounds of sets' elements, and fail rules.

4.1. **Finite domain set sizes.**

Our first change to jmle was treating the sizes of sets as finite domain variables. Regardless of the type of the elements in a set, its size is guaranteed to be an integer. Therefore, we can express the size of any type of set as a finite domain variable.

Even without knowing the exact size of a set, knowing the size's range of values can be valuable. For example, consider two sets A and B, where $A \subseteq B$, $|A| \in [0..10]$, and $|B| \in [3..6]$. We can reduce the domain of $|A|$ to $[0..6]$, since there is no way for A to have more than 6 elements while it is a subset of B.

In some instances, we can use the domain of a set's size to determine it completely. As an example, consider the set C, where $|C| \in [1..3]$ and $\{4, 5, 6\} \subseteq C$. Since C has at least 3 elements and at most 3 elements, it must be true that $C = \{4, 5, 6\}$.

The following are some example rules that use finite domain set sizes to prune variable domains.

- If $A \subseteq B$ and A's maximum size is greater than B's maximum size, lower A's maximum size.
- If $A \subseteq B$ and A's minimum size is greater than B's minimum size, raise B's minimum size.
- If $A \subseteq B$, where B is a known set, and A's maximum size is greater than B's size, lower A's maximum size.

4.2. **Lower and upper bounds.**

As discussed in Section 2.6, we can store lower and upper bounds of a set. A set's lower bound is a subset of elements that are definitely in the set. A set's upper bound is a superset of its elements, which may or may not be known. If it is not known, the upper bound is the universal set, minus a finite (possibly empty) collection of elements that are known not to be in the set. If the upper bound is known, then any element in the set (and by extension, in the set's lower bound) must also be in the upper bound.

A known upper bound allows us to backtrack through the elements of a set. When every rule in the rule set has been tried for a given constraint store but none of them can fire, library code initiates backtracking. One by one, an element from the superset is chosen and added to the lower bound, and all the rules are tried again. If an illegal state is reached, the system backtracks to the last choice point, rejecting the element most recently added.

Storing a lower bound gives us a perfomance advantage while backtracking. Previously, the backtracking code added items from the upper bound to a candidate set which was initially empty. Now, we start backtracking aware of the elements in the lower bound. We know that our set contains these elements, so there is no reason to waste time adding them individually through backtracking. Another benefit is that we can avoid any candidate sets not containing the elements in the lower bound – there is no way for the backtracking code to remove any of these original lower bound elements from the set.

Before and during backtracking, constraint rules using lower and upper bounds can add useful information to the constraint store. Some example rules are listed below. Observe that some of these also take advantage of the finite domain set sizes.

- If A is the same size as its lower bound, A is equal to its lower bound.
- If A is the same size as its upper bound, A is equal to its upper bound.
- If $A \subseteq B$ but A's lower bound is not a subset of B's lower bound, add all elements of A's lower bound to B's lower bound.
- If the size of A's lower bound is greater than A's minimum size, raise A's minimum size.
- If multiple lower and upper bounds are known for the same set, combine this information by taking the union of the lower bounds and the intersection of the upper bounds.

4.3. **Fail rules.**

Fail rules detect and report unsatisfiable states as quickly as possible. In a backtracking situation, it is beneficial to fail quickly to avoid wasting time searching down nonpromising paths. Our fail rules use various properties of sets, including finite domain set sizes and lower and upper bounds. Several illegal conditions that we have made into fail rules are given below.

- $A \subseteq B$, but A's minimum size is greater than B's maximum size.
- There are more elements in the lower bound than the maximum size of the set.
- An element in a set's lower bound is known not to be in the set.
- A set's lower bound is not a subset of its upper bound.

5. Implementation

The finite domain constraints on a set's size were the first step in this project's implementation. With those in place, we worked on implementing and testing rules bounding sets from below and rules bounding sets from above. Ultimately we combined these two ideas by creating a new constraint relating lower and upper bounds. In this section we discuss some implementation details of the different kinds of additions and modifications we made to the constraint solver.

5.1. **Finite domain set sizes.**

Our first step was refactoring the list of rules to allow a set's size to be treated as a finite domain variable. Initially, jmle had a constraint called `setList`, which related a set, an ArrayList containing a subset of its elements, the set's size, and the type of the elements. The existing constraint handling rules assumed that the size of the set given in `setList` was ground. We did not need to modify this constraint, but we did need to change most rules containing `setList` constraints to allow sizes of sets to be finite domain variables. Refactoring and error checking was necessary, as some rules should only be allowed to fire once the set's size has become ground. Other rules did not need to be changed, and in some cases we made multiple copies of rules, with different behavior when the size is ground and when the size is still unknown.

5.2. **The setBound constraint.**

We began by writing rules bounding sets from below with `setList` and from above with `isFSubset`, a constraint that facilitated backtracking through a ground superset. Later, we realized that keeping the two bounds in separate rules was causing us to lose and duplicate information, so we combined them in a new constraint called `setBound`.

This constraint relates a set, its lower bound, its upper bound, its size, a collection of elements known not to be in the set, and the type of the elements. The parameters to the constraint are in this order. The lower bound, upper bound, and not-has set are always ground; the universal set is represented as a special ground instance, so even sets without a known upper bound can have a ground upper bound. Backtracking is allowed if the upper bound is ground and not the universal set, and if no other rules are able to fire.

In order to make backtracking work with `setBound`, we modified the library code that performs backtracking. This code, in a class called JChoice, is responsible for determining when backtracking is possible and adding the correct constraints to the store at each choice point. For example, when backtracking is initiated, it chooses an element from the upper bound and adds it to the lower bound, all in one step.

Examples of constraint rules using `setBound` are shown in Figures 5.4, 5.5 and 5.6.

```
if (I3 = org.jmlspecs.jmlexec.runtime.FDomain.getHi(FD) &&
    I4 = org.jmlspecs.jmlexec.runtime.FDomain.getHi(FD2) &&
    org.jmlspecs.jmlexec.runtime.MyBoolean.TRUE() =
        org.jmlspecs.jmlexec.runtime.MyNumber.lt(I4, I3))
    { isSubset(JC1, JC2, St1) && setBound(JC1, JC3, JC4, JC5, I1, St1) &&
        setBound(JC2, JC6, JC7, JC8, I2, St1) && fdVar(I2, St2, FD2) &\&
        fdVar(I1, St2, FD)} <=>
    { FD3 = org.jmlspecs.jmlexec.runtime.FDomain.upperBoundEQ(I4, FD) &&
        fdVar(I1, St2, FD3) } isSubset4;
```

FIGURE 5.4. Example rule `isSubset4`, which prunes set sizes. If `JC1` is a subset of `JC2` and `JC1`'s maximum size is greater than `JC2`'s maximum size, `JC1`'s maximum size is lowered. The `fdVar` constraints relate a number, its type, and its domain, so `getHi(FD)` returns the maximum value in the domain of `I1`. This is a simpagation rule, so constraints listed after `&/&` in the head of the rule are replaced by the constraints in the body. That is, the constraint `fdVar(I1, St2, FD)` is replaced by `fdVar(I1, St2, FD3)`, where `FD3` is the newly pruned domain.

```
if (org.jmlspecs.jmlexec.runtime.JMLTool.isNotSubset(JC2, JC6, St1))
    { setBound(JC1, JC2, JC3, JC4, I1, St1) && isSubset(JC1, JC5, St1) &\&
        setBound(JC5, JC6, JC7, JC8, I2, St1)} <=>
    { JC9 = org.jmlspecs.jmlexec.runtime.JMLTool.union(JC2, JC6, St1) &&
        setBound(JC5, JC9, JC7, JC8, I2, St1)} isSubset8;
```

FIGURE 5.5. Example rule `isSubset8`, which updates lower bounds. If `JC1` is a subset of `JC5`, but `JC1`'s lower bound `JC2` is not a subset of `JC5`'s lower bound `JC6`, all elements of `JC2` are added to `JC6`. This is a simpagation rule, so constraints listed after `&/&` in the head of the rule are replaced by the constraints in the body. That is, the constraint `setBound(JC5, JC6, JC7, JC8, I2, St1)` is replaced by `setBound(JC5, JC9, JC7, JC8, I2, St1)`, where `JC9` is the union of `JC2` and `JC6`.

5.3. **Eliminating propagation rules.**

Because propagation rules are dangerous, we would like to keep as few of them as possible in our rule set. However, in some situations it is necessary to write a rule where none of the head constraints can be removed without losing information from the constraint store. The example propagation rule `intersection1` in Figure 2.2 has this property – the body constraints do not imply the head constraint, so we lose information if we remove the head constraint. We removed this rule from our rule set because it was firing

```
if (I4 = org.jmlspecs.jmlexec.runtime.FDomain.getLow(FD) &&
    I3 = org.jmlspecs.jmlexec.runtime.FDomain.getHi(FD2) &&
    org.jmlspecs.jmlexec.runtime.MyBoolean.TRUE() =
        org.jmlspecs.jmlexec.runtime.MyNumber.lt(I3, I4))
    { isSubset(JC1, JC5, St1) && setBound(JC1, JC2, JC3, JC4, I1, St1) &&
      setBound(JC5, JC6, JC7, JC8, I2, St1) && fdVar(I1, St2, FD) &&
      fdVar(I2, St2, FD2) } <=> { false } isSubsetfail2;
```

FIGURE 5.6. Example fail rule `isSubsetfail2`. If `JC1` is a subset of `JC5` but `JC1`'s minimum size is greater than `JC5`'s maximum size, `false` is added to the constraint store to show that a contradiction has been reached. The `fdVar` constraints relate a number, its type, and its domain, so `getLow(FD)` returns the minimum value in the domain of `I1`.

repeatedly and filling up the constraint store, but we still need to be able to calculate an intersection. It is useful to have domain pruning rules that help with this calculation.

To try another approach, we considered the intersection of two sets with `setBound` constraints. If their intersection already has a `setBound` constraint, we can prune its domain with a simpagation rule. If the intersection does not have a `setBound` constraint, however, the only sensible way to add a new `setBound` constraint is with a propagation rule. To avoid adding a new propagation rule, we changed the compiler to automatically create `setBound` constraints for many sets – those marked `assignable` in JML, return values of methods, and the results of set operations. This modification removes the need for this particular propagation rule and similar rules.

5.4. **Bag rules.**

A bag is a mathematical construct similar to a set, but unlike a set it can hold multiple copies of the same item. There are many constraint handling rules that deal with properties of bags. Originally, several bag rules used the `setList` constraint to hold a bag's lower bound. We refactored `setList` into `setBound`, but `setBound` does not work with bags because of the way the lower and upper bounds are stored. In order to maintain the basic functionality of bags in jmle, we added a constraint called `bagList` that has the same behavior as `setList`, but is restricted solely to bags. We changed the name from `setList` to avoid confusion with `setBound`.

5.5. **Non-rule additions.**

Although the bulk of the programming for this project was the creation and modification of constraint handling rules, it has also been necessary to modify some Java library code. An example previously mentioned is the backtracking code in JChoice, which is invoked

by the constraint solver. Another example is the JMLTool class, a collection of static methods called directly by the constraint handling rules. We added a few simple helper methods to JMLTool, such as computing a set difference or determining whether one Java Collection is a subset of another. In addition, we added cases to basic set methods such as intersection and union to handle the behavior of the universal set; for example, the intersection of any set A with the universal set is A, and the union of A with the universal set is the universal set.

5.6. **Testing.**

To verify that the constraint rules were firing under the expected circumstances, we created test JML specifications and compiled them with jmle. Turning on a trace flag and running the compiled specifications yielded information about each rule considered and the state of all constraints and variables at each step in the process. Since jmle repeatedly iterates over every rule in the rule set, we want to minimize the number of rules and ensure that they are considered in an appropriate order. Using the information from testing we were able to remove superfluous rules, change the order in which rules are considered, gain insight into how the solver works, and write new rules that take advantage of the contents of the constraint store.

6. Results

We tested the usefulness and efficiency of our new rules by using a set of benchmarks to measure the running time with the new rules and the old rules. We used four specifications as benchmarks: we wrote three of them and found a fourth in the Mozart-Oz documentation. An advantage of writing our own benchmarks is that we can ensure that the specifications take advantage of our new rules. However, an external benchmark written without our rules in mind is more likely to give accurate results.

6.1. **Our specifications.**

We created three benchmarks that would cause the new rules to fire. These benchmarks are optimized to take advantage of the new rules, so they are not necessarily representative of a typical JML specification. However, these tests illustrate the potential these new rules have to improve performance. The results are given in table and graph format in Figures 6.7 and 6.8.

The three benchmarks test different outcomes of a constraint program. Each specification lists several sets, with various subset relationships and element declarations; the goal of each program is to assign elements to sets so that all constraints are satisfied.

The first test is a successful value assignment to a very simple program, the second is a case in which there is contradictory information (eliciting a PostconditionException),

Benchmark	Original rules	New rules
Successful assignment	1623.14	819.20
Failure detection	7689.82	2321.36
Insufficient information	14468.40	904.92

Figure 6.7. Average running times in ms of three benchmarks, using the original set of rules and the new set of rules.

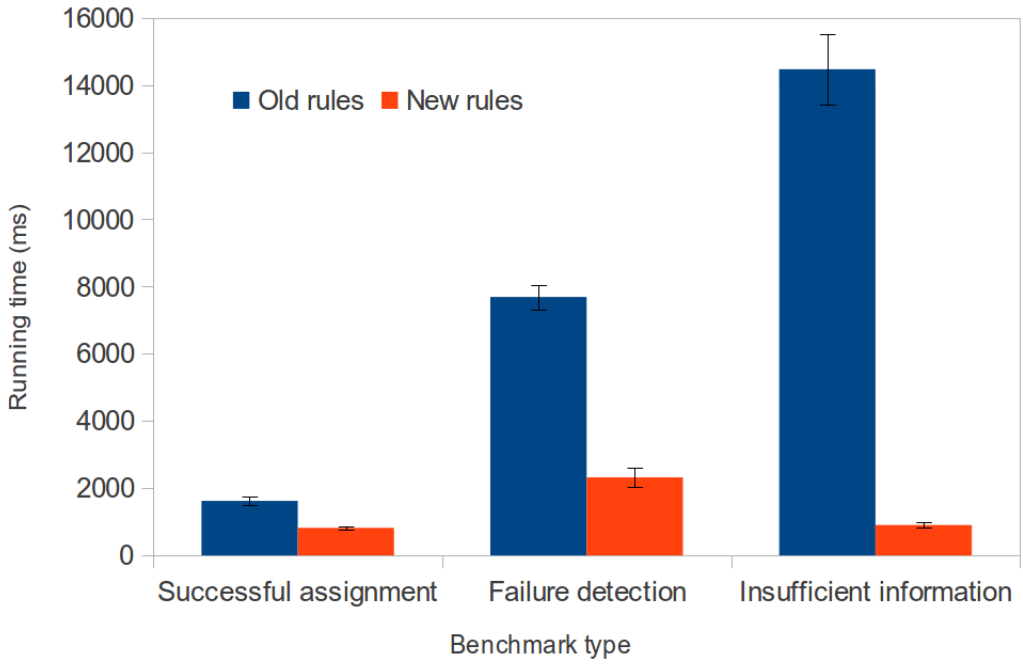

FIGURE 6.8. Average running time (ms) of our three JML specifications, with jmle configured to use the old set of rules or the new, improved set of rules. Error bars show standard deviation.

and the third does not have enough information to create a value assignment (eliciting an InsufficientInformationException). For the second two tests, the running time was measured to immediately after the exception was thrown. The JML specifications for these tests can be found in Appendix A.

Each of the benchmarks show a significant performance improvement with the new set of rules. The improvement of the insufficient information specification is particularly dramatic. Perhaps this is because the old rules do not simplify these kinds of constraints effectively, so the large number of constraints in the store cause the solver to iterate repeatedly over a large set of rules before throwing an exception. Again, these test programs do not necessarily represent a typical JML specification, but these results do show that these sort of rules have the potential to significantly improve jmle's performance.

6.2. **GolfTournament.**

The fourth benchmark is a JML adaptation of the GolfTournament specification from the Mozart-Oz documentation (Müller, 2008b). The goal of the specification is to assign up to 32 golfers to groups of four for a certain number of weeks such that no two golfers play in the same group twice. The full JML specification is in Appendix A.

This specification is much more complicated than the previous three, so running it with the smallest non-trivial parameters (two groups and one week) took on average 299.55 s and 49.65 s with the old and new rules, respectively. This is illustrated graphically in Figure 6.9. It is encouraging that this benchmark also shows an improvement, since we did not create this specification to hit our new rules.

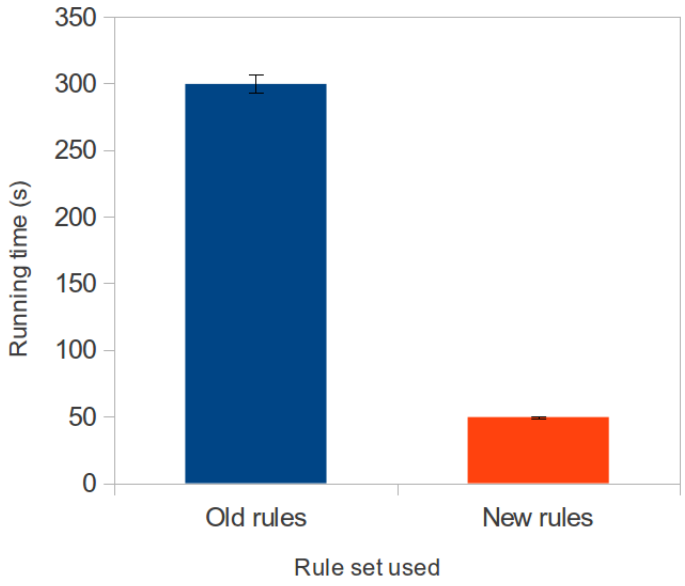

FIGURE 6.9. Average running time, in seconds, of the GolfTournament JML specification, with jmle configured to use the old set of rules or the new, improved set of rules. Error bars show standard deviation.

The running times in Figures 6.7, 6.8, and 6.9 are averages of 50 runs (10 runs for GolfTournament). They were measured using System.currentTimeMillis() on a dual-core 2.0GHz Intel system running Ubuntu 11.04.

7. Future Work

It is encouraging that this approach does improve jmle's performance. Using a similar approach in related domains may cause further improvement. For example, an area for further work is implementing rules for bags similar to the ones implemented for sets. We did some work with lower bounds of bags, but there is much more that could be done. For example, some of the subset rules would translate very easily to subbag rules, while rules that take advantage of specific properties of sets would need to be reworked to be compatible with the properties of bags.

Another area for further work is implementing finite domain and bounds constraints on sequences. For example, one could store sub-sequences or super-sequences as an analogue to lower and upper bounds. Alternatively, the position of an element in the sequence could be treated as a finite domain variable instead of a single integer. In jmle, sets and sequences are represented similarly, so it is natural to try to extend some of our set solving techniques to sequences. However, sets and sequences are very different mathematical constructs, so this would certainly not be a trivial translation.

Appendix A

These JML specifications were used as benchmarks to compare the performance of the old and new rules. Each specification was compiled with jmle and run with a jar file containing the appropriate version of the constraint handling rules.

Successful assignment: This specification gives enough information about three sets, A, B, and C, for jmle to determine their contents. We specify the ranges of sizes of the sets ($|A| \in [0..2], |B| \in [0..4], |C| \in [0..2], |B \cap C| = 1$), the set relationships ($A \subseteq B, A \cap C = \emptyset$), and some of the elements in the sets. The constraint solver concludes that $A = \{1, 5\}, B = \{1, 2, 3, 5\}$, and $C = \{3, 4\}$.

```
/*@ assignable A, B, C;
  ensures (\exists int i; (\exists int j; (\exists int k;
  A.int_size() == i && B.int_size() == j && C.int_size() == k;
  A.isSubset(B) && C.intersection(A).isEmpty() &&
  C.intersection(B).int_size() == 1 &&
  i >= 0 && j >=0 && k >= 0 && i <= 2 && j <= 4  && k <= 2 &&
  A.has(1) && B.has(2) && A.has (5) && C.has(4) && C.has(3) && B.has(3))));
*/
public void findSetsSuccess() {}
```

Failure detection: This specification gives conflicting information about the sets, causing the constraint store to become unsatisfiable. B and D are disjoint subsets of A, which has 5 elements. B has one item in its lower bound, and its subset C has 3 other items in its lower bound, so B must have at least 4 elements. D has at least 2 elements. These set sizes make it impossible for $B \cup D \subseteq A$, so the constraint satisfaction fails.

```
/*@ assignable A, B, C, D;
  ensures (\exists int i; (\exists int j;
  A.int_size() == 5 && B.int_size() == i && C.int_size() == 3 &&
  D.int_size() == j; B.isSubset(A) && D.isSubset(A) && C.isSubset(B) &&
  B.intersection(D).isEmpty() && i >= 0 && j >= 2 && i < 10 && j < 10 &&
  A.has(1) && A.has(2) && A.has(3) && A.has(4) && A.has(5) && B.has(1) &&
  C.has(2) && C.has(3) && C.has(4)));
*/
public void findSetsFail() {}
```

Insufficient information: This specification gives some information about the sets, but not enough to determine their exact contents. The constraint solver determines as much information about the sets as possible, then throws an exception when there is no more information available for it to use.

```
/*@ assignable A, B, C;
  ensures (\exists int i; (\exists int j; (\exists int k;
  A.int_size() == i && B.int_size() == j && C.int_size() == k;
  A.isSubset(B) && C.intersection(A).isEmpty() &&
  C.intersection(B).int_size() == 2 &&
  i >= 0 && j >=0 && k >= 0 && i < 4 && j < 4 && k < 4 &&
  A.has(1) && B.has(2) && C.has(3))));
 */
public void findSetsInsufficient() {}
```

GolfTournament: This specification simulates a golf tournament by assigning golfers to groups of four in multiple weeks. No golfer plays in a foursome with any other golfer more than once. For a more detailed explanation, see the Mozart-Oz website (Müller, 2008b). To collect our data, we ran `foursomes(1,2)`.

```
/*@ assignable \nothing;
    ensures (start == n) ? \result == new JMLEqualsSet<Integer>(n) :
               \result == makeSet(start + 1, n).union(
                new JMLEqualsSet<Integer>(start));
    public static model JMLEqualsSet<Integer> makeSet(int start, int n);
 */
```

```
/*@ assignable \nothing;
    ensures (schedule.int_size() == 1) ? \result == schedule.get(0) :
                \result == schedule.get(0).concat(flatten(schedule.trailer()));
    public static model JMLEqualsSequence<JMLEqualsSet<Integer>>
      flatten(JMLEqualsSequence<JMLEqualsSequence<JMLEqualsSet<Integer>>> schedule);
 */

/*@ assignable \nothing;
    ensures (\exists int numPlayers; numPlayers == 4 * numFoursomes
         && (\exists JMLEqualsSet<Integer> allPlayers;
          allPlayers == makeSet(1, numPlayers)
           && \result.int_size() == numWeeks
           && (\forall int i; 0 <= i && i < numWeeks;
            \result.get(i).int_size() == numFoursomes)
           && (\forall int i; 0 <= i && i < numWeeks;
                 (\forall int j; 0 <= j && j < numFoursomes;
                   \result.get(i).get(j).int_size() == 4
                && \result.get(i).get(j).isSubset(allPlayers)))
            )
         && (\exists JMLEqualsSequence<JMLEqualsSet<Integer>> flat;
          flat == flatten(\result) &&
              (\forall int i; 1 <= i && i < numFoursomes * numWeeks;
                 (\forall int j; 0 <= j && j < i;
                   flat.get(i).intersection(flat.get(j)).int_size() <= 1)))
         );
 */
public static JMLEqualsSequence<JMLEqualsSequence<JMLEqualsSet<Integer>>>
foursomes(int numWeeks, int numFoursomes) {
return null;
}
```

References

Abdennadher, S., Kramer, E., Saft, M., & Schmauss, M. (2002). JACK: A Java constraint kit. In M. Hanus (Ed.), *Electronic notes in theoretical computer science* (Vol. 64).

Bouquet, F., Dadeau, F., Legeard, B., & Utting, M. (2005, July). Symbolic animation of JML specifications. In *Proceedings of the International Conference on Formal Methods 2005 (FM'05)* (Vol. 3582, p. 75 - 90). Springer-Verlag.

Bouquet, F., Legeard, B., & Peureux, F. (2002). CLPS-B: A constraint solver for B. In J.-P. Katoen & P. Stevens (Eds.), *Tools and algorithms for the construction and analysis of systems* (Vol. 2280, p. 235-256). Springer Berlin / Heidelberg. Retrieved from `http://dx.doi.org/10.1007/3-540-46002-0_14`

Chalin, P., Kiniry, J., Leavens, G., & Poll, E. (2006). Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In F. de Boer, M. Bonsangue, S. Graf, & W.-P. de Roever (Eds.), *Formal methods for components and objects* (Vol. 4111, p. 342-363). Springer Berlin / Heidelberg. Retrieved from `http://dx.doi.org/10.1007/11804192_16`

Grieskamp, W. (2000, September). A computation model for Z based on concurrent constraint resolution. In J. P. Bowen, S. Dunne, A. Galloway, & S. King (Eds.), *ZB 2000: Formal specification and development in Z and B, First International Conference of Z and B Users* (Vol. 1878, p. 414 - 432). York, UK: Springer-Verlag.

Haridi, S., & Franzen, N. (2008, July). *Tutorial of Oz.* Retrieved from `http://www.mozart-oz.org/documentation/tutorial/index.html`

Hunt, A., & Thomas, D. (2000). The pragmatic programmer: from journeyman to master. In (p. 109-119). Addison-Wesley.

Keating, M., Kostrubiak, A., & Wahls, T. (2011). Enhancing the jmle tool. In *Proceedings of the 26th annual conference of the Pennsylvania Association of Computer and Information Science Educators (PACISE)* (p. 15 - 21). Shippensburg, PA: Shippensburg University.

Krause, B., & Wahls, T. (2006). jmle: A tool for executing JML specifications via constraint programming. In *In FMICS06, volume 4346 of LNCS* (pp. 293–296). Springer-Verlag.

Marriott, K., & Stuckey, P. J. (1998). *Programming with constraints: an introduction.* Cambridge, MA: The MIT Press.

Müller, T. (2008a, July). *Problem solving with finite set constraints in Oz: a tutorial.* Retrieved from `http://www.mozart-oz.org/documentation/fst/`

Müller, T. (2008b, July). *Scheduling a golf tournament.* Retrieved from `http://www.mozart-oz.org/documentation/fst/node6.html`

Wahls, T. (2005). Compiling formal specifications to Oz programs. In P. van Roy (Ed.), *MOZ 2004, the second international Mozart/Oz conference* (Vol. 3389, p. 66 - 77). Springer-Verlag.