The Thesis Committee for Vidya Priyadarshini Narayanan
Certifies that this is the approved version of the following thesis :

Milao: A Novel Framework for Mixed Imperative and Declarative Formulation and Solving of Structural Constraints

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Sarfraz Khurshid

Dewayne Perry

Milao: A Novel Framework for Mixed Imperative and Declarative Formulation and Solving of Structural Constraints

by

Vidya Priyadarshini Narayanan B.Tech

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2009

Dedication

Dedicated to my parents and brother

Acknowledgements

Words cannot express my acknowledgement for the constant support and guidance from my advisor Dr. Sarfraz Khurshid. A special mention to Dr.Perry Dewayne for sparing his precious time in evaluating my Thesis. I wish to thank all people who have helped me in any capacity with my Thesis and graduate studies at UT in general. I would also like to thank current and alumni members of SVVAT (http://svvat.ece.utexas.edu/) group for their feedback and numerous group discussions which I enjoyed. A special thanks to my colleague, Shadi Abdul Khalek, for being patient with my innumerous set of questions. Finally, I wish to thank my dearest brother for his constant love and support and my father for the innumerous sacrifices he has made for me to pursue this Master's degree without which it was only a distant dream.

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CCF-0702680, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

December 4, 2009

Abstract

Milao: A Novel Framework for Mixed Imperative and Declarative Formulation and Solving of Structural Constraints

Vidya Priyadarshini Narayanan, MSE

The University of Texas at Austin, 2009

Supervisor: Sarfraz Khurshid

Advances in constraint solving and increases in processing power have enabled new approaches for automating specification-based testing. However, writing specifications and scaling techniques that utilize them remain challenging. We introduce Milao -- a novel framework for mixed imperative and declarative formulation and solving of structural constraints -- which addresses both these challenges. One, Milao introduces a mixed style for writing specifications using a combination of declarative and imperative styles, which provides flexibility in specification formulation and reduces its burden on the user. Two, it introduces a mixed technique for solving constraints using a combination of solvers in synergy. As enabling technologies, the Alloy tool-set and the Java PathFinder model checker are used. Initial experiments witness the benefits of our framework.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

The benefits of using specifications in software testing have long been known [1]. Traditional specification-based techniques required much manual effort, and hence posed a burden on the user and remained primarily confined to academic settings. Recent years have seen much work in automation of specification-based techniques. A significant part of automation is due to the advancements in constraint solving techniques, which have been well-supported by the wide availability of faster processors. Witness, for example, the recent resurgence of symbolic execution, which was developed over three decades ago [2], but has only recently started to find its way into industrial settings.

While the recent technological advances have increased the effectiveness and efficiency of specification-based techniques, scalability remains an issue and most existing techniques are able to handle no more than moderately sized units of code. There are two challenges to scalability: (1) the need for manually writing specifications; and (2) the ability to efficiently utilize specifications to automate efficient analyses.

We introduce Milao -- a novel framework for *mixed* imperative and declarative formulation and solving of structural constraints – which addresses both these challenges. One, it presents a mixed style for writing specifications that represent *input constraints*, which describe desired test inputs, using a combination of declarative and imperative programming styles. Specifically, the user formulates the specification using a combination of the Alloy specification

language -- a first-order logic based on sets and relations -- and the Java programming language. Alloy, with its support for path (navigation) expressions using transitive closure, allows succinct formulations of the properties of heap traversals. In contrast, Java, with its wide use, provides a familiar notation that is likely to pose a minimal learning burden. We have designed a structures style for writing constraints using a combination of Alloy and Java, thereby providing the users much flexibility in how they formulate their specifications and making them easier to write. Indeed, the users may write part of the specification in Alloy and part of it in Java. Our structured style has another major benefit: it allows users to state their preferences for solvers, which opens the possibility of a synergistic application of multiple solvers, e.g., to use dedicated solvers in tandem.

Two, Milao presents a mixed technique for solving input constraints using a combination of solvers. Specifically, we show how the Alloy Analyzer -- a first-order logic analyzer that uses off-the-shelf propositional satisfiability (SAT) solvers -- and the Java PathFinder (JPF) -- a Java model checker that implements its own special Java Virtual Machine -- can be used in synergy as a basis of solving structural constraints to support automated test generation. A key challenge in putting together the Alloy Analyzer and JPF is the difference in data models of their respective languages. Alloy is based on sets and relations on atoms, whereas Java programs have primitives, references, and arrays. To bridge this gap in the data models, we use data translations based on a relational model of the program heap: abstraction translations translate Java data structures into

Alloy instances and concretization translations translate Alloy instances to Java data structures.

We have conducted initial experiments to demonstrate the potential benefits of Milao. As subject programs, we used Java programs that implement complex data structures. These programs have previously been used to evaluate several other projects on automated testing.

This thesis makes the following contributions:

1) **Mixed declarative and imperative specifications**. It introduces the idea of using a declarative language and an imperative language in conjunction, to write specifications, specifically complex data structure invariants.

2) **Mixed declarative and imperative constraint solving**. It introduces the idea of using abstraction and concretization data translations to enable the solving of mixed constraints.

3) **Milao framework**. It presents the Milao framework, which enables mixed imperative and declarative formulations and the solving of structural constraints.

4) **Demonstration of benefits**. It presents initial experimental results to demonstrate the potential benefits of the Milao framework.

The rest of this thesis is organized as follows. Chapter 2 describes the background work upon which this tool is built. In particular this section is divided into two subsections. Section 2.1 describes the basics of Alloy Specification Language. Section 2.2, gives a description of the tool support (Alloy Analyzer). Section 2.3 describes the basics of Model Checking and gives a description of the

Java Path Finder. Chapter 3 is divided into two subsections. Section 3.1 describes an example of testing a linked data structure and illustrates how programmers can use the test framework to automate generation of test inputs. Section 3.2 describes the key algorithms of the test framework. Section 4 describes some of the subject programs that were used have performed with the test framework. Section 4 describes our test framework prototype and evaluates its performance against executing programs that are written purely in Alloy. We present the limitation of the test framework in Section 5, discuss related work in Section 6 and in Section 7 provide the conclusion and future possible extensions.

Chapter 2 Background

2.1 An Overview of Alloy

In this section, we describe the basics of the Alloy specification language. Alloy is a strongly typed language that assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. An Alloy specification is a sequence of paragraphs that can be of two kinds: signatures, used for construction of new types, and formula paragraphs, used to record constraints. Each specification starts with a module declaration that names the specification; existing specifications may be included in the current one using open declarations. The relevant parts of Alloy are shown in this chapter using the list example. This section focuses on the syntax and semantics of Alloy.

2.1.1 Signature Paragraphs

A signature paragraph introduces a basic (or uninterrupted) type. For, example

```
sig List {}
```

introduces the signature `List` as a set of atoms. A signature paragraph may also declare a subset. For example,

```
one sig List0 extends List {}
```

declares `List0` to be a subset of the `List` signature; the keyword "`one`" declares the subset to be a singleton set. A signature declaration can also introduce relations (that are called `fields`) and constraints on their values. For example,

```
sig List  {
  header: lone Node
```

```
}

sig Node  {
  elem: Int,
  next: lone Node
}
```

introduces `List` and `Node` as uninterpreted types. The field declaration for `header` introduces a relation of type `List -> Node`. The qualifier `lone` specifies that the `List` may have one or no `Nodes` in the `header` field, i.e., `header` is a partial function.

2.1.2 FACTS AND PREDICATES

A `fact` is a formula that takes no arguments and will not be invoked explicitly. In other words, `fact` statements in Alloy puts explicit constraints on the model. When Alloy searches for satisfying instances, it discards any instance that violates any `fact`. Thus, if the fact is trivially false, then we will simply get no instances for the model.

A predicate is a parameterized formula that can be invoked elsewhere. For example, the predicate p declared as:

```
pred p (param1:T1, …, paramn:Tn) {
  [list of constraints -- each must evaluate to true or false]
  [multiple constraints are implicitly conjoined]
}
```

has n parameters: the parameter param1 of type T1, and param2 ….,paramn of types T2, ..., Tn respectively.

2.2 ALLOY ANALYZER

The Alloy Analyzer [3] is an automatic tool for analyzing models created in Alloy. Given a formula and a *scope*, a bound on the number of atoms in the universe, the analyzer determines whether there exists an instance of the formula (that is, an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it. Since first order logic is undecidable, the analyzer limits its analysis to a finite scope. The analysis [3] is based on a translation to a Boolean satisfaction problem, and gains its power by exploiting state-of-the-art SAT solvers.

The models of formulae are termed instances. The following valuation of sets and relations introduced earlier in this section represents an example instance:

```
Seq/Int=[[0], [1], [2], [3]]
This/List=[[List$0]]
This/Node=[[Node$0], [Node$1]]
This/List.header=[[Node$1]]
This/Node.next=[[Node$1, Node$0]]
```

The analyzer can enumerate all possible instances of an Alloy model. The analyzer adapts the symmetry-breaking predicates of Crawford et al. [4] to provide the functionality of reducing the total number of instances generated—the original boolean formula is conjugated with additional clauses in order to produce only a few instances from each isomorphism class [5]. The input parameters to the analyzer can be set such that the analyzer enumerates exactly non-isomorphic

instances. However, the resulting formulae tend to grow very large, which slows down enumeration causing it to take more time to enumerate fewer instances.

2.3 An overview of Java Path Finder

Our current embodiment of Milao uses the Java PathFinder model checker (JPF), an explicit state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). Since it is built on a JVM, it can handle all of the language features of Java, but in addition it also treats nondeterministic choice expressed in annotations of the program being analyzed. These features for adding non-determinism are used to implement lazy initialization of fields. JPF supports specialized program annotations to cause the search to backtrack when a certain condition evaluates to true—this is used to stop the analysis of infeasible paths (when path conditions are found to be unsatisfiable). Lastly, JPF supports various heuristics [6], including ones based on increasing testing-related coverage (e.g., statement, branch and condition coverage), that can be used to guide the model checker's search.

Chapter 3 The Milao Framework

This section presents the Milao framework. We first give an illustrative example. Next, we describe the overall architecture and design of the framework.

3.1 EXAMPLE

This section illustrates how Milao enables generation of test inputs from input constraints that are written using more than one language and solved using more than one constraint solvers. A singly linked list can be declared as follows in Java:

```
public class List {
    Node header;
    int size;
    static class Node {
        int elem;
        Node next;
    }
}
```

Figure 1: Java Declaration of Singly linked list

The header field represents the first node of a non-empty list. If the list was empty the header field would contain the value null. The size field represents the number of nodes that are present in the list. Objects of the class List represent a singly linked list and objects of the inner class Node represent the nodes in the list. The integer field elem represents the integer value of each node and the Node field next points to the next node.

Let us say the user wants to generate instances of a singly linked list that satisfy the following constraints

- Acyclicity Property: Constraint specifying that the list has no cycles.
- Sortedness Property: Constraint specifying that the list elements are sorted (in ascending order)
- SizeOk property: Constraint specifying that the size field is set exactly to the number of nodes that are actually in the list.

The key idea behind constraint solving in Milao is that part of the solving is done using SAT/Boolean backend of the Alloy Analyzer and part of the constraints are solved using lazy initialization implemented with Java Path Finder (JPF).

In this particular example, the user chooses to solve the Acyclicity property using the Alloy Analyzer, followed by solving the Sortedness property using JPF, and then solving the SizeOk property using the Alloy Analyzer. Thus this example illustrates multiple executions of the Alloy Analyzer with an interleaved execution of JPF.

The following mixed constraint describes the Acyclicity, Sortedness, and SizeOk properties:

```
boolean repOk() {
    if (!acyclic()) return false;
    if (!sorted()) return false;
    if (!sizeOk()) return false;
    return true;
}
boolean acyclic() { all n: header.*next | n !in n.^next }
boolean sorted() {
    Node current = header;
    while (current != null) {
        if (current.next != null)
            if (current.elem > current.next.elem) return false;
        current = current.next;
    }
    return true;
}

boolean sizeOk() { size = #header.*next }
```

Figure 2: Mixed Constraint for Acyclic Singly Linked List

Note how the Acyclicity and SizeOk properties are written using Alloy formulas, whereas the Sortedness property is written as a Java predicate.

Since the Alloy Analyzer requires complete Alloy programs (and not just Alloy formulas), the following Alloy model is used to solve the Acyclicity property:

```
sig List { header: lone Node }
sig Node { next: lone Node }
fact { Node in List.header.*next }
pred acyclic(l: List) {
  all n: l.header.*next | n !in n.^next
}
```

Figure 3: Alloy Model for Acyclic Singly Linked List

To solve the model in Figure 3 and enumerate instances satisfying the model up to 3 nodes, we use the "run" command as follows:

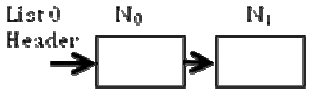
```
run acyclic for exactly 1 List, 3 Node
```

The dot '.' operator in Alloy represents relational join (which in the case of "l.header" amounts to relational image since "l" is a singleton set), the "in" keyword denotes 'a subset of', the star '*' operator denotes reflexive transitive closure, the caret '^' operator denotes transitive closure. Hence the predicate "acyclic" states that for each node in the set of all nodes reachable from the header following zero or more traversals of next, the node is not present in the set of all nodes that can be reached from itself in one or more traversals of next. Hence this predicate ensures that only linked list instances satisfying the acyclic property are generated. The run command specifies that we are interested in generated satisfying instances with one List atom and up to three Node atoms.

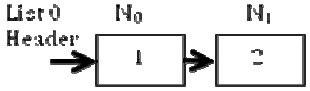
Running the above Alloy model in Alloy Analyzer gives us several Alloy instances that satisfy the Acyclicity property.

Figure 4 illustrates how such an Alloy instance feeds into the next phase where lazy initialization of the integer fields (elem) is performed using JPF to satisfy the Sortedness property, and how the combined solution feeds into the final phase where the Alloy Analyzer is used for solving the SizeOk property.

Phase 1: Acyclicity Property (Alloy)



Phase 2: Sortedness Property (JPF)



Phase 3: SizeOk Property (Alloy)

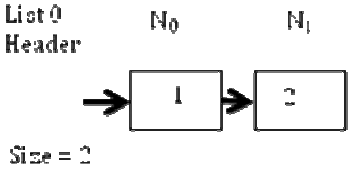


Figure 4: Phases of Constraint Solving

To illustrate lazy initialization with JPF, consider the Sortedness Property. The following Java code shows how instrumentation enables field initialization when the field is accessed for the first time. The basic idea behind lazy initialization is to use non-deterministic assignment to initialize fields at the time of first access. To observe when the first access happens, shadow fields (e.g., elem_init) are used. All field access in the original code are replaced with method invocations (e.g., elem()) to enable lazy initialization. Details are available in [7].

For the linked list, the following method elem() is introduced in the class Node and the method sorted() is in the class List:

```
int elem(int bound){
    if(elem_init)
        return elem;
    elem = Verify.getInt(0,bound);
    elem_init=true;
    return elem;
    }
boolean sorted(){
    Node current = header;
    int sizeBound = 3;
    while(current!=null){
            if(current.next!=null)
                if(current.elem(sizeBound)>current.next.elem(sizeB
                ound))
                    return false;
                current = current.next;
            }
            return true;
    }
```

Figure 5: Instrumented code for solving the Sortedness property using JPF

The integer parameter bound is the value used to set a bound for the range of integer values the linked list nodes can take. In this example, we run the experiment with sizeBound equals 3. Note the non-deterministic assignment of field elem using the JPF library method getInt(x, y), which non-deterministically returns an integer in the range x, x + 1, …, y.

The third property to satisfy is the SizeOk property. The following Alloy model encodes the partial solutions that have been generated in the first two phases and solves the constraint on the size field.

```
sig List{
  header: lone Node,
  size: Int
  }
sig Node{
  next: lone Node,
  elem: Int
  }
sig List0 extends List{}
pred sizeok(l:List){
  # l.header.*next = l.size
  }
sig Node0 extends Node{}
fact { List0.header=Node0 }
fact { Node0.elem=1 }
sig Node1 extends Node{}
fact { Node0.next=Node1 }
fact { Node1.elem=2 }
fact { no Node1.next }

run sizeok for exactly 1 List, 3 Node
```

Figure 6: Alloy Model checking the Size Constraint

The cardinality '#' operator in Alloy is used to represent the number of elements present in the set. Given the Alloy model in Figure 5, the Alloy Analyzer enumerates instances that satisfy the SizeOk property and by construction also satisfy the Acyclicity and Sortedness properties.

3.2 ARCHITECTURE

In this section, we present the overall architecture and design of the Milao framework and how it enables automated test input generation for Java programs. Milao leverages Alloy and JPF and applies them in tandem for constraint solving.

Figure 6 shows the overall architecture. Given a mixed constraint that is a conjunction of sub-constraints, the Constraint Selector/Refiner module selects a sub-constraint to solve next. This sub-constraint is delegated to either the Alloy

Analyzer of JPF. The solution generated by either of them undergoes a data translation (A2J or J2A) as needed to transform it into a form suitable for solving of the remaining sub-constraints. The translated solution feeds back into the Constraint Selector/Refiner module which uses its input solution as a basis of generating the next sub-constraint subject to the partial solution generated thus far. When all constraints are solved or infeasible constraints are detected the Constraint Selector/Refiner generates its output solution report that contains the result of applying the Milao approach of mixed constraint solving.

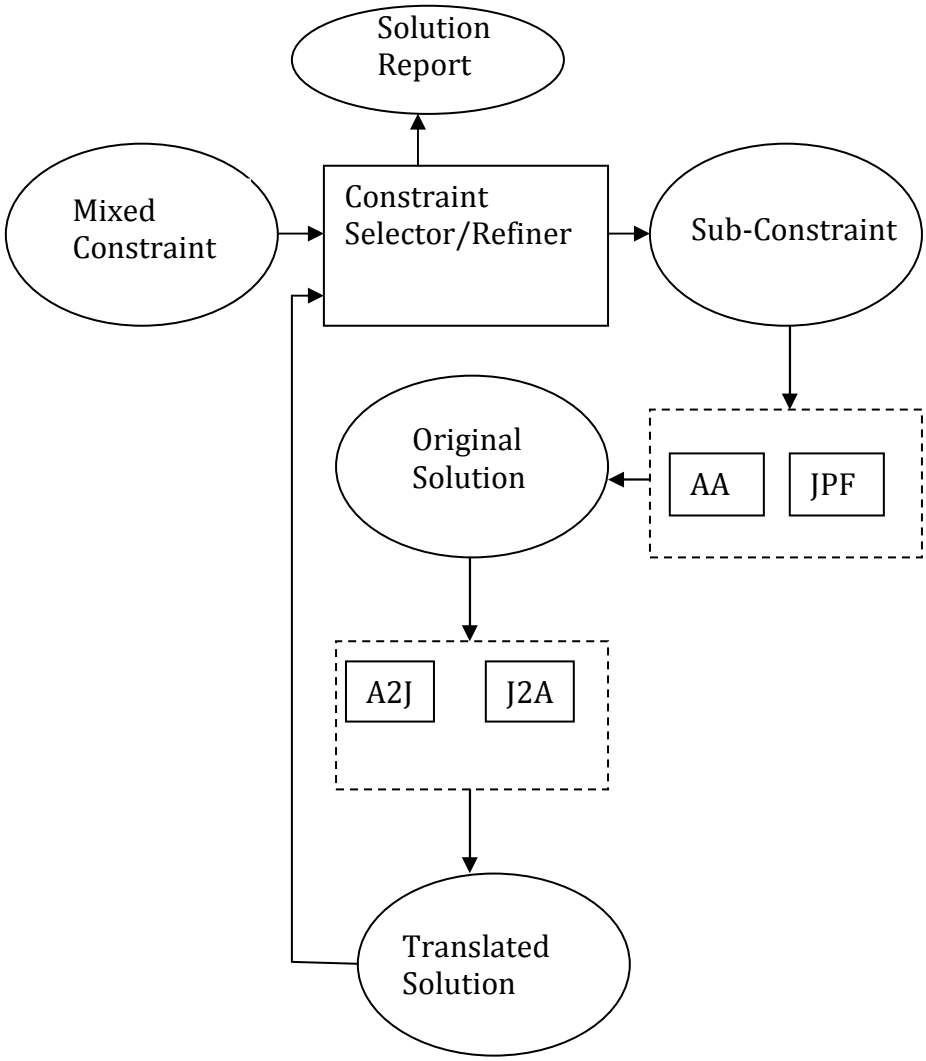


Figure 7: Milao Framework

Milao provides a template for writing the mixed constraint in a structured manner. Figure 7 presents this template. Let C be the class of the root object with n fields f1, …, fn. The mixed constraint (repOk) is written in the following style: check each property and return false if it is violated; if all properties are satisfied, return true. Each property is written in the style of a Java predicate (method that

returns a Boolean). However, if the property is written in Alloy, the body of the predicate is an Alloy formula, whereas if the property is written in Java, the body of the predicate is Java code that checks the property.

To enable an application of the standard Java development tools on mixed constraints, Milao supports Java annotations and allows writing the mixed constraint so that it compiles using a standard Java compiler. To achieve this, the user uses the annotation to write Alloy predicates while the corresponding method bodies simply return true (to implement stubs that enable compilation). For each predicate, the annotation specifies the language of the predicate, the solver that should be used, and the fields that are being solved for. This info about the fields allows Milao to slice any data declarations that are unnecessary for solving the constraints on the given fields. For example, the Alloy model for the Acyclicity property (Figure 2) did not need to have data representation for the size field of List or the elem field of Node. Moreover, the use of mixed constraint templates allows users to define a constraint prioritization to guide the overall solving process. The predicate annotations provide users further control by specifying what solvers to use.

To re-cap, the predicate annotation attributes specify:

- The programming language in which the method is written.
- The type of Solver to be used for solving.
- The predicate body representing the constraint
- The list of fields that need to be solved by this method

```
Class C {
   Type1 f1;
   Type2 f2;
        .
        .
        .
   Typen fn;

   boolean repOk() {
      if(!p1()) return false;
           .
           .
      if(!pk()) return false;
   }

   @Language = L1, Solver = S1, Pred = P1, Fields = {f11,f12,..f1n1}
   boolean p1() {
           .
           .
   }
           .
           .
   @Language = Lk, Solver = Sk, Pred = Pk, Fields = {fk1, fk2, ..fknk}
   boolean pk() {
           .
           .
   }
}
```

Figure 8: Template for writing mixed constraints

3.2.1 Example Template Usage

Figure 9 illustrates a use of the template by showing the mixed constraint from Figure 2 written using annotations.

```
boolean repOk(){
    if (!acyclic())return false;
    if (!sorted()) return false;
    if (!sizeOk()) return false;
    return true;
}

@genspecification (Language = "Alloy", Solver = "SAT",
                   pred = "all n: header.*next | n !in " +
                   "n.^next\n}",fields =
                   {"List.header","Node.next"})
boolean acyclic(){
    return true;
}

@genspecification (Language = "Java", Solver = "JPF", pred = "",
                   fields = {"Node.elem"})
boolean sorted(){
    Node current = header;
    while(current!=null){
        if(current.next!=null)
            if(current.elem()>current.next.elem())
                return false;
            current = current.next;
    }
    return true;
}

@genspecification (Language = "Alloy", Solver = "SAT",
                   pred = "size = # header.*next",
                   fields = {"List.size"})
boolean sizeOk(){
    return true;
}
```

Figure 9: Definition of Hybrid RepOk for Sorted Acyclic List

3.2.2 Partial Evaluation/ Incremental Solving

Given a mixed constraint (written using the template) Milao uses invocations of the Alloy Analyzer and JPF together with concretization and abstraction translations to solve the given constraint. We next describe details of how Milao invokes the Alloy Analyzer and JPF in tandem.

To illustrate, recall our running example of singly linked list declares the following four fields.

- L_h – List.header
- L_s – List.size
- N_n – Node.next
- N_e – Node.elem

Given the mixed repOk method for this example, solving proceeds in three phase: (1) solving for L_h and N_n is done using Alloy Analyzer; (2) solving for N_e is done using Java Path Finder; and (3) again Alloy Analyzer is used to solve for L_s. Figure 9 below depicts the order in which field values are computed. Note the solving for Sortedness and SizeOk can proceed in parallel (and the solutions can then be merged).

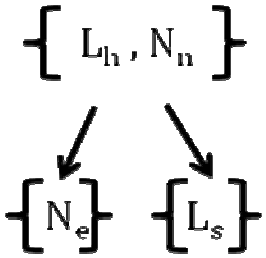


Figure 10: Order of Solving

In more detail, the user-guided solving of constraints proceeds as follows:

- In the first phase, the parser parses the input mixed constraint and generates an *.als (Alloy Analyzer file) containing the Alloy model specifying the constraints that need to be solved first (Acyclicity). The tool then uses Alloy Analyzer to enumerate instances for the Alloy input specification. That is, in this

phase only those fields chosen by the user to be solved declaratively using Alloy are determined (List.header and Node.next). The user can also specify the choice of the solver to be used for solving the Alloy constraints.

- In the second phase, *incremental* solving is done for each of these instances. Each instance is in turn concretized into a Java object graph (a2j) and the fields chosen by the user to be determined using the Java PathFinder are solved for (Node.elem). Executing JPF using lazy initialization on the Java predicate that specifies the constraints achieves this.

- In the third and final phase, the Java state (object graph) is converted back into an Alloy instance (j2a) and an *.als file is generated to include the partial instance generated thus far and to solve for the remaining field(s) (List.size). The tool provides the instances that Alloy Analyzer generates upon executing the generated .als files.

These phases are simply iterated over to for solving constraints that consist of several sub-constraints. Note the concretization and abstraction translations may be performed independently of any fixed ordering of sub-constraints, and therefore they allow much flexibility in specifying a likely optimal ordering.

3.2.3 Alloy Predicates

We next discuss the Alloy specifications that Milao uses. Alloy specifications are built from the given mixed constraint. Figure 11 describes the

basic algorithm for building an Alloy specification using a given mixed constraint and a selected Alloy predicate.

A generated Alloy model is written to disk and the Alloy Analyzer is executed to find a satisfying instance. The Alloy model generated comprises of Alloy predicates, which represent the constraints that need to be solved by Alloy. Milao builds Alloy predicates by combining the signature of the Java annotated method with the annotation attribute values

```
AlloyModel generateModel(Classfiles jar, Predicate p){
    result is alloy model consisting of:
    foreach class C in jar (which does not have a library spec)
        sig declaration for C
    foreach field element f in class C
        relation declaration for f
    for annotated method p with language attribute L = "Alloy"
        predicate definition P with the value of the predicate
        attribute (pred) as its body
}
```

Figure 11: Input Specification Generation Algorithm

Recall the singly linked list example introduced in Section 3.1. Milao uses the following Alloy specification for generating all valid instances of a sorted acyclic linked list for a given bound of 2. This alloy model checks for acyclic and cardinality (Size) constraint.

```
sig List{
  header: lone Node,
  size: Int
}
sig Node{
  next: lone Node,
  elem: Int
}

// checks whether the list is acyclic
pred acyclic(l: List){
  all n: l.header.*next | n !in n.^next
}

pred repOk(l: List) {
  acyclic[l]
}

run repOk for exactly 2 Node, 1 List
```

Figure 12: Alloy Model for Acyclic Linked List

3.2.4 Abstraction and Concretization Translations

We next discuss the test driver that the testing tool generates to automate test input generation. A test driver consists of Java code that performs abstraction and concretization translations and appropriately invokes the Alloy Analyzer and JPF. The translations are based on previous work that developed the TestEra framework [8].

A concretization translation, abbreviated a2j, translates Alloy instances to Java objects or data structures [8]. Likewise, an abstraction, abbreviated j2a, translates Java data structures to Alloy instances.

Concretization a2j is performed in two stages. In the first stage, a2j creates for each atom in the Alloy instance, code instrumentation which in turn creates

corresponding objects of the Java classes. This corresponding mapping between the atom and the object is stored in a 'name' field of every object. In the second stage, a2j establishes the relationships among the Java objects created in the first stage and builds the actual data structure by allocating memory etc for the objects in the data structure.

Figure 13 describes the concretization algorithm a2j. The algorithm takes as input an Alloy instance and instruments Java code to create objects that represent a test input. For every object that is created, the algorithm adds a string field 'name', which is assigned the name of the object. This helps in storing correspondence between Alloy atoms and Java objects. In the first step, a2j instruments code to create Java objects of appropriate classes for each atom in the instance. In the second step, a2j sets values of objects according to the tuple values in the input relations. Finally, a2j writes the instrumented java code onto a file and invokes the Java program to create the actual objects. This creates the actual representation of the data structure (in this example) with values set for those fields that the Alloy Analyzer had been guided to solve.

```
void a2j(Instance a){
    String filea2j, filej2a;
    StringBuffer inscodea2j, inscodej2a;

    // for each atom create a corresponding Java object
    foreach(sig in a.sigs())
    foreach(atom in sig) {
        inscodea2j.append("SigClass obj = new SigClass()");
        inscodea2j.append("obj.name = obj");
    }

    // establish relationships between created Java objects
    foreach(rel in a.relations())
    foreach(<x,y> in rel)
    inscodea2j.append("x.rel=y");
    // write the instrumented code onto a java file
    writeToFile(filea2j.java,inscodea2j);
}
```

Figure 13: Alloy to Java Instrumentation Algorithm

```
void j2a(Object result, Instance ret){
    foreach class C in Java program {
        Sig declaration for C
    foreach field f in class C {
        Relation declaration for f
    }
   Foreach assignment instruction i in class C
     Generation of facts that set tuples in relations
    }
}
```

Figure 14: Java to Alloy Instrumentation Algorithm

Figure 14 gives the j2a algorithm which traverses the given program heap to generate Alloy facts that force the fields that have already been solved for to take appropriate values and to enable using these values for solving for the other fields.

Chapter 4 Subject Programs

This section describes the data structures that we use to evaluate Milao.

4.1 BINARY TREE

A binary tree is a rooted acyclic graph in which each node has at most two children. Typically the child nodes are called *left* and *right*.

The following two constraints are checked for in a binary tree:

1. Acyclic Property: No cycles exists in the tree, i.e., there is no sharing of nodes along left or right fields.

2. SizeOk property: The size field of the tree is correctly set to the actual number of nodes in the tree;

The above two properties of the binary tree are readily expressible in Alloy as well as in Java. Figure 15 shows a mixed constraint that specifies them:

```
boolean repOk(){
    if(!acyclic()) return false;
    if(!sizeOk()) return false;
    return true;
}
@genspecification (Language = "Alloy", Solver = "SAT",
       pred = "all n: root.*(left + right) {\n" +
              "  n !in  n.^(left + right)\n" +
              "  no n.left & n.right\n" +
              "  sole n.~(left + right)\n" +
              "}", fields =
                   {"Tree.root","Tree.left","Tree.right"})
boolean acyclic(){
    return true;
}
@genspecification (Language = "Java", Solver = "JPF",
                   pred = "", fields = {"Tree.size"})
boolean sizeOk(){
    return true;
}
```

Figure 15: Hybrid repOk for Binary Tree

The first phase solves for acyclicity using the following Alloy predicate.

```
// checks for acyclicity
pred acyclic(t: Tree){
  all n: t.root.*(left + right) {
    n !in n.^(left+right)
    no n.left & n.right
    sole n.~(left + right)
  }
}
```

The second phase takes each Alloy instance in turn, concretizes it to a Java object graph and uses lazy initialization to solve for the size property.

4.2 BINARY SEARCH TREE

A binary search tree is a binary tree with an additional constraint that for any node, the element in the node is larger than all elements in the node's left sub-tree and smaller than all elements in the node's right sub-tree.

The following three constraints are checked for in a binary search tree:

1. -Acyclic Property: No cycles exists in the tree.
2. -Search Property: The element values of each node satisfy the binary search tree property

- SizeOk property: The size field of the tree is correctly set to the total number of nodes in the tree;

Figure 16 gives the partial mixed constraint that specifies binary search trees.

```
boolean repOk() {
    if(!acyclic()) return false;
    if(!search()) return false;
    if(!sizeOk()) return false;
    return true;
}

@genspecification (Language = "Alloy", Solver = "SAT",
                   pred = "...",fields {"..."})
boolean acyclic(){
    return true;
}

@genspecification (Language = "Java", Solver = "JPF",
                   pred = "", fields = {"Node.elem"})
boolean search() {
    ...
}

@genspecification (Language = "Alloy", Solver = "SAT",pred = "
                    #root.*(left+right) = size",fields =
                   {"Tree.Size"})
boolean sizeOk(){
    return true;
}
```

Figure 16: Hybrid repOk for Binary Search Tree

Milao first uses the Alloy Analyzer to solve for acyclicity. It then applies concretization translation and uses JPF to solve search constraints. Finally it applies abstraction translation and uses the Alloy Analyzer to solve for sizeOk using the Alloy predicate:

```
//checks for size consistency
pred sizeOk(t:Tree) {
  #t.root.*(left+right)=t.size
}
```

In addition to binary tree and binary search tree we also use the sorted singly-linked list (Section 3.1) for our initial evaluation of Milao.

Table 1 lists the constraints for each subject data structure and the solvers used.

DATA STRUCTURE	CONSTRAINTS AND SOLVER
Binary Tree	Check for acyclic property -> Alloy Check for sizeOk property -> JPF
Binary Search Tree	Check for acyclic property -> Alloy Check for search property -> JPF Check for sizeOk property -> Alloy
Acyclic Singly-Linked List	Check for acyclic property -> Alloy Check for sizeOk property -> JPF
Sorted Acyclic Singly-Linked List	Check for acyclic property -> Alloy Check for sorted property -> JPF Check for sizeOk property -> Alloy

Table 1: Summary of how mixed constraints are solved

Chapter 5 Initial Evaluation

This section describes an initial experimental evaluation of the Milao framework. We performed the experiments using the Alloy Analyzer and the JPF model checker. For all the line graphs given below, the x-axis represents the number of nodes used in the experiment and the y-axis represents the time taken in milliseconds to generate one instance of a valid data structure.

We compare the time per instance using the traditional *pure* Alloy approach (where all the constraints are written in Alloy and solved using one invocation of the Alloy Analyzer) with the mixed approach of Milao.

Table 2 and Figure 19 give the performance comparison for Binary Tree.

Table 3 and Figure 20 give the performance comparison for Binary Search Tree.

Table 4 and Figure 21 give the performance comparison for Acyclic Singly-Linked List.

Table 5 and Figure 22 give the performance comparison for Sorted Acyclic Singly-Linked List.

We can observe from these results that the pure Alloy approach works well for small sizes, Milao's approach scales better for these examples.

Number of Nodes	Execution time using Alloy Analyzer (ms)	Execution time using MILAO (ms)
2	41.3	8.3
4	52.3	7.16
6	62.5	27.5
10	295.6	45.3
15	694.3	1374.5
20	16007.83	11084.83

Table 2: Solving time per instance comparison for Binary Tree

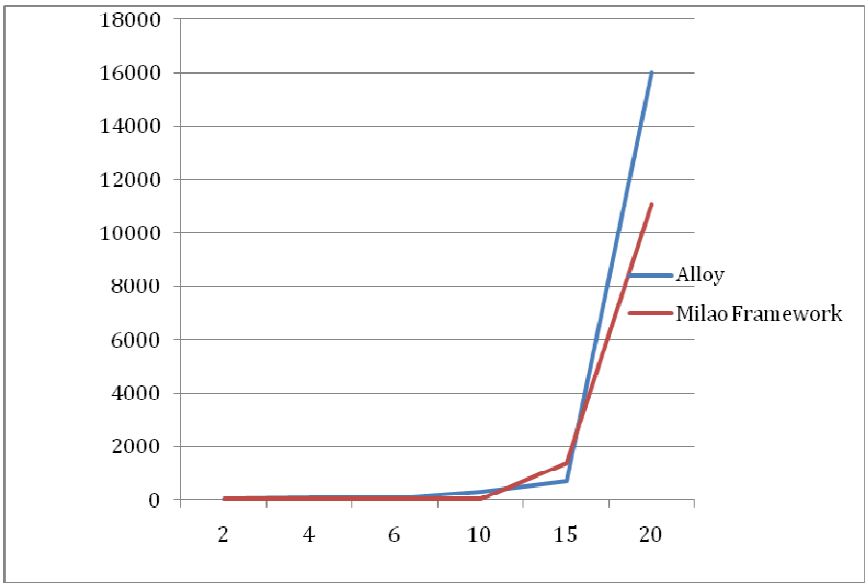


Figure 17: Solving time per instance comparison for Binary Tree

No. of Nodes	Execution time for 1 instance using Alloy Analyzer (ms)	Execution time for 1 instance using MILAO (ms)
2	21.3	908.2
4	31.6	1203.82
6	453.83	1705.96
7	836	1797.6
8	21998.16	2510.1

Table 3: Solving time per instance comparison for Binary Search Tree

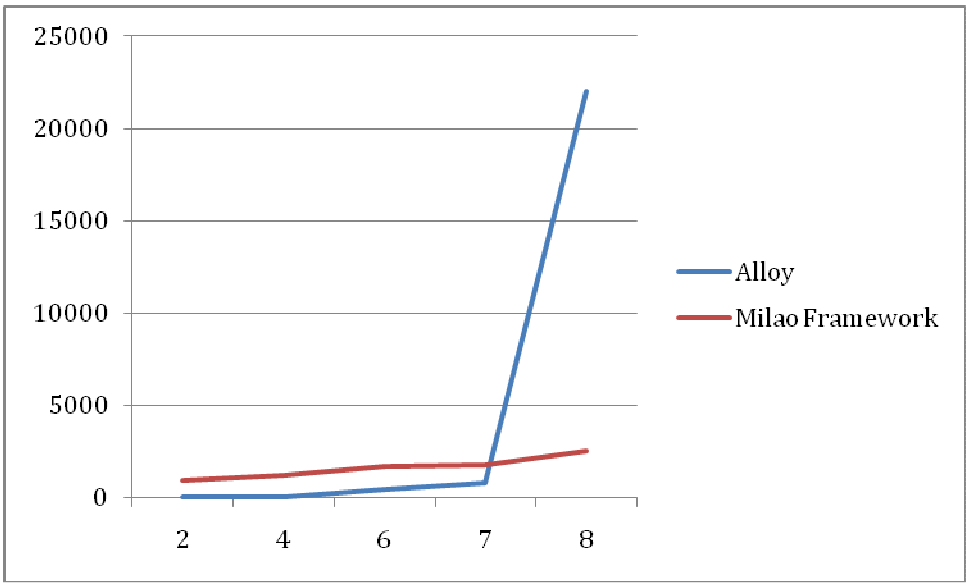


Figure 18: Solving time per instance comparison for Binary Search Tree

Number of Nodes	Execution time using Alloy Analyzer (ms)	Execution time using MILAO (ms)
2	9.5	119.6
4	10.5	144.16
6	24.6	152.83
10	185	625.3
15	1975.16	2123.6
20	20308.83	45832.75

Table 4: Solving time per instance comparison for Acyclic Singly-Linked List Data Structure

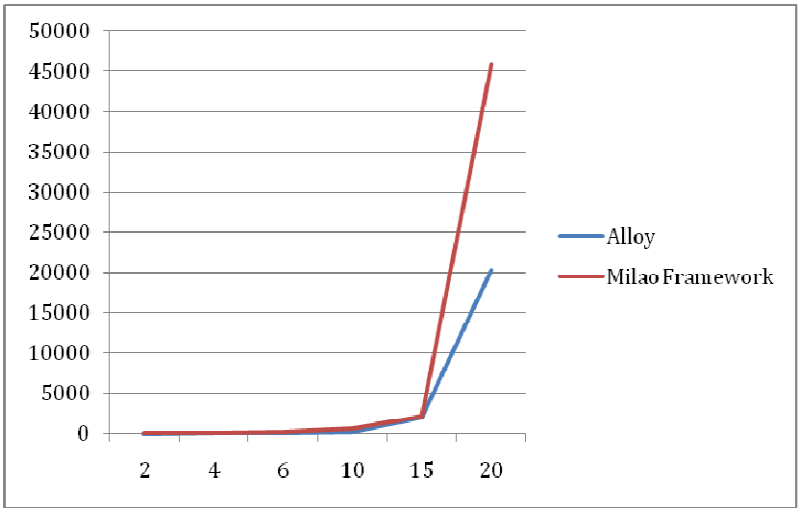


Figure 19: Solving time per instance comparison for Acyclic Singly-Linked List Data Structure

Number of Nodes	Execution time using Alloy Analyzer (ms)	Execution time using MILAO (ms)
2	14.5	128
4	28.16	161
6	256.5	216
10	22600.83	297

Table 5: Solving time per instance comparison for Sorted Acyclic Singly-Linked List Data Structure

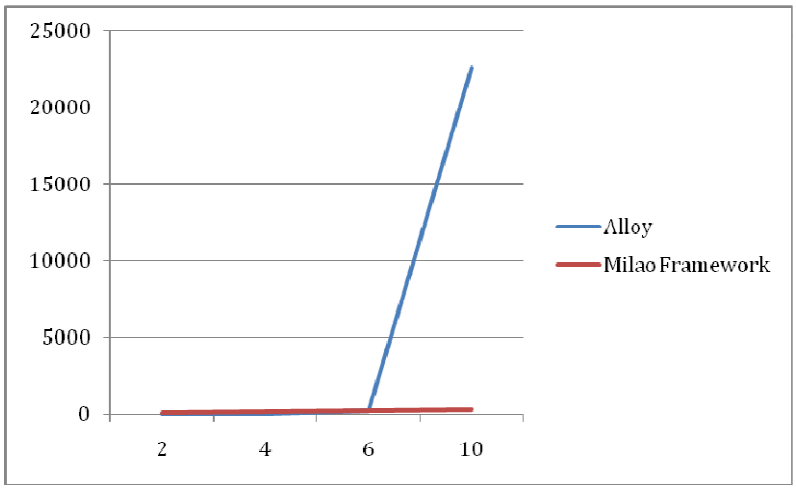


Figure 20: Solving time per instance comparison for Sorted Acyclic Singly-Linked List Data Structure

Chapter 6 Related Work

There is a plethora of research on specification-based testing. A recent overview can be found in [9]. Two frameworks that are most closely related to the Milao framework are TestEra[8] and Korat[10]. Indeed, Milao is inspired by and based on TestEra and Korat. The key difference is that TestEra and Korat support one language (Alloy for TestEra and Java for Korat) and one solver (SAT for TestEra and systematic backtracking search for Korat), whereas Milao introduces the idea of using more than one languages and more than one solvers in tandem to provide specification-based testing.

Our work also draws inspiration from recent work by Uzuncaova [11] which shows how to use a combination of solvers for solving Alloy formulas and how to prioritize constraints. The key difference is our support for more than one language for writing constraints and the synergistic use of the Alloy Analyzer and the Java PathFinder – two tools designed for fairly different purposes.

The rest of this section describes the basic concepts of and Korat.

6.1 TestEra

TestEra is a powerful framework for automated testing of Java programs. The key idea behind TestEra is to use structural invariants for input data to automatically generate test inputs. As an enabling technology, TestEra uses the first-order relational notation Alloy and the Alloy Analyzer. The automatic

constraint solving ability of the Alloy Analyzer is used to generate concrete inputs to a program. The program is executed and each input-output pair is automatically checked against a correctness criteria expressed in Alloy. TestEra requires no user input besides a method specification and an integer bound on input size.

A TestEra specification for a method states the method declaration (i.e., the method return type, name, and parameter types), the name of the Java classfile (or sourcefile) that contains the method body, the class invariant, the method precondition, the method post-condition, and a bound on the input size. In our current implementation, we give this specification using command line arguments to the main TestEra method.

Given a TestEra specification, TestEra automatically creates three files. Two of these files are Alloy specifications: one specification is for generating inputs and the other specification is for checking correctness. The third file consists of a Java test driver, i.e., code that translates Alloy instances to Java input objects, executes the Java method to test, and translates Java output objects back to Alloy instances.

TestEra's automatic analysis proceeds in two phases:

- In the first phase, TestEra uses the Alloy Analyzer to generate all non-isomorphic instances of the Alloy input specification.
- In the second phase, each of the instances is tested in turn. It is first *concretized* into a Java test input for the method. Next, the method is executed on this input. Finally, the method's output is *abstracted* back to an Alloy instance. The output Alloy instance and the original Alloy input instance valuate the signatures and relations of the Alloy input/output

specification. TestEra uses the Alloy Analyzer to determine if this valuation satisfies the correctness specification. If the check fails, TestEra reports a counterexample. If the check succeeds, TestEra uses the next Alloy input instance for further testing.

6.2 KORAT

Korat is another framework for constraint-based testing of Java programs. It supports Java specifications, specifically Java predicates that represent desired input constraints, which are based on the method preconditions. Thus Korat does not require the use of a new specification language.

Korat uses a backtracking search and monitors the given predicate's executions to prune the search. Moreover, Korat provides isomorphs breaking and enumerates only inputs that are non-isomorphic.

The user provides Korat:

1) A type declaration for the data structure under consideration.

2) A finitization method to check bounds on the input.

3) repOk, a boolean predicate method in Java which represents desired structural invariants.

Karta's functioning can be briefly explained with the following notions:

1. Predicate method: Korat uses a Java predicate method which returns a boolean based on the method's precondition.
2. Candidate Inputs: Candidate inputs are generated by Korat given a predicate method and a bound on the size of its inputs. The candidate inputs return true for the predicate method.

3.Pruning: Korat does an efficient search for candidate inputs. It monitors the fields accessed while generating candidate inputs. It keeps a check on the values which return false for the predicate and uses this to prune significant portions of the state space. This effectively reduces execution time without missing out on any valid candidates.

4. Specifications: The specifications used by Korat are written as Java predicates.

5. Non – isomorphism: Korat does not generate structures that are isomorphic to each other.

Chapter 7 Conclusions and Future Work

We presented Milao, a novel framework for mixed imperative and declarative formulation and solving of structural constraints. Milao provides flexibility in writing specifications and supports specification structuring that guides constraint solving and enables using more than one solvers. A key technical challenge of supporting multiple languages and multiple solvers is the differences in the underlying data models. Milao addresses this challenge by building on data abstraction and concretization translations used in previous work on the TestEra framework for specification-based testing of Java programs. Our initial embodiment of Milao uses Alloy and Java for writing mixed constraints. Alloy's intuitive path expressions allow succinct formulation of heap traversals. Java offers familiarity and poses minimal burden on programmers familiar with commonly used imperative languages. Thus, Milao provides a flexible style for writing specifications, which is likely to be accessible to a wide class of users. For constraint solving Milao uses the SAT-based Alloy Analyzer and lazy initialization with Java PathFinder. Thus, Milao shows how to use in tandem, tools that were originally designed for fairly different purposes. Initial experiments with Milao show the promise approaches based on mixed constraints hold. Milao assumes the correctness of the mixed constraints given by the user. It also assumes the feasibility of using the chosen solvers and ordering of constraints.

There are several future directions for exploration. First, a more rigorous evaluation and a comparison with other "pure" approaches (e.g., Korat or lazy initialization) which use one language is needed -- such a comparison is likely to lead to new approaches for optimized solving using mixed constraints. Second, a robust implementation for the Milao framework is needed -- the implementation will enable an application of Milao to larger case-studies. Third, mixed constraint solving naturally supports an application of parallel algorithms, which can enable further performance optimizations. Fourth, Milao's approach of mixed constraints also lends naturally to a synergistic application of techniques that may originally have been designed for different purposes, e.g., symbolic execution can be used in conjunction with the Alloy Analyzer and the Java PathFinder to further optimize performance and effectiveness.

References

[1] J. Goodenough and S. Gerhart. Toward a theory of test data selection. IEEE Transactions on Software Engineering, June 1975

[2] J. C. King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976

[3] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In Proc. 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, June 2000.

[4] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In Proc. Fifth International Conference on Principles of Knowledge Representation and Reasoning, 1996.

[5] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In Proc. Workshop on Theory and Applications of Satisfiability Testing, June 2001.

[6] A. Groce and W. Visser. Model checking java programs using structural heuristics. In Proc. International Symposium on Software Testing and Analysis (ISSTA). ACM Press, July 2002.

[7] S. Khurshid, C. Pasareanu and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. *9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003),* Warsaw, Poland. Apr 2003.

[8] S. Khurshid and D. Marinov. TestEra: Specification-based Testing of Java Programs Using SAT. *Automated Software Engineering Journal*, Volume 11, Number 4. October 2004. (Journal version of ASE'2001 paper.)

[9] D.Marinov. Automatic Testing of Software with Structurally Complex Inputs. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, December 2004.

[10] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 123–133, July 2002.

[11] E. Uzuncaova. Efficient Specification-based Testing Using Incremental Techniques. PhD thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, Dec. 2008

Vita

Vidya Priyadarshini Narayanan was born in 1985 in Ernakulam, India, daughter of Sri. Gopalachary Narayanan and Vaidehi Narayanan. She did her schooling in Jawahar Higher Secondary School, Chennai, India. She attended Sri Venkateswara College of Engineering affiliated to Anna University, Chennai where she received her Bachelor of Technology in Information Technology in May 2007. She joined University of Texas, Austin, in August 2007 to pursue her Masters degree. She had internship experiences with Cisco during the summer of 2008 and Polycom during the summer of 2009.

Her primary interests are software test automation, algorithms and communication protocols.

Permanent address: Plot No. 61, New No. 3, First Link Street, Raghava Reddy Colony, Ashok Nagar, Chennai, Tamil Nadu 600083

This thesis was typed by the author.