

MintEra: A Testing Environment for Java Programs

by

Basel Y. Al-Naffouri

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

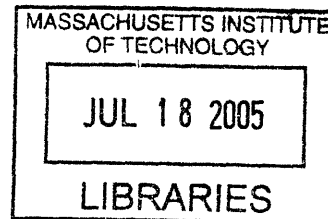
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 17, 2004 [September 2004]

© Basel Y. Al-Naffouri, MMIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

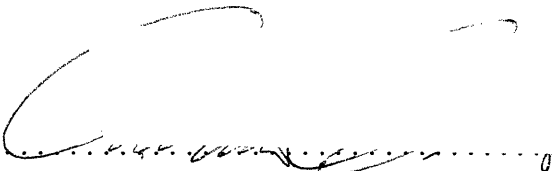


Author  

Department of Electrical Engineering and Computer Science
August 17, 2004

Certified by 

Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by 

Arthur C. Smith
Chairman, Department Committee on Graduate Students

ARCHIVES

MintEra: A Testing Environment for Java Programs

by

Basel Y. Al-Naffouri

Submitted to the Department of Electrical Engineering and Computer Science
on August 17, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We introduce MintEra, an automatic testcase generator and verifier. Using an simple, easy-to-read yet expressive language called AAL, users can specify representation-invariants and assertions within programs. MintEra uses the representation-invariant to generate testcases and translates assertions into Java run-time checks, which verify testcases. The tool then graphically visualize failed testcases to help users debug their code. MintEra encourages documentation of programs by using specification to test and verify code. Effectively, the tool checks code and specification against each other. Thus, MintEra helps users ensure correctness of their programs as well as their specification. In this thesis, we provide a number of extra features that we hope would develop MintEra into an effective tool that could be used by the general software engineering community.

Thesis Supervisor: Daniel Jackson

Title: Associate Professor

Acknowledgments

First and foremost, I would like to thank God All Mighty for all His bounties. The bounties without which I could not do anything. My belief in Him was my guide through hard times.

Second, I would like to thank my family for all their support throughout the years. To my mother, who has always taught me how to work hard and made me the man I am. To my father, who has instilled the love of science in me since I was barely walking. To my sister, whose love and pride made me realize how great she is. To my brother, who has always encouraged me to reach a higher status. If I achieved anything in this life, it was because of you.

I would like to thank my advisor, professor Daniel Jackson, who is the greatest mentor I ever had. I have learnt a lot from his insight, intelligence, and dedication. By working with him, I have learnt more about software engineering than I dreamt of learning. I am proud to be his advisee.

A special thanks to the members of Software Design Group. Sarfraz Khurshid and Darko Marinov, for guiding me step by step to understand automated testing, taking time from their busy schedule to ensure I was on the right track. Manu Sridharan for his help with the Alloy parser and visitors. Greg Dennis for helping me to understand Alloy. Emina Torlak for reading my thesis and helping me with my writing. Samuel Daitch for his help with visualizing testcases in MintEra. Greg Dennis, Derek Rayside, Emina Torlak and Robert Seater for the great discussions and their help throughout my tenure in this group.

Thanks to Waseem Bakr, Nasruddin Nazerali and Saba Gul for offering their help to test MintEra and telling me what changes I needed make. Their comments helped me understand the bigger picture about the tool.

I would like to also thank the MIT Muslim Student Assosiation, which has been a second family for me. The support I had from them helped make the hard MIT life enjoyable. To Ayman, Faisal, Omar, Bilal, Esa, Prasad, Babak, Jalal, Tarik, Osman, Fadilah, Nazbeh, Numan, Minhaj, Ahmed, Saba, Waseem, Nasser, Nasruddin,

Jessica, Tanya Aadel and everyone else, thank you for being my true brothers and sisters.

To Professors Tayo Akinwande, Jane Dunphy, and Janet Sonenburg: your love, care and advice were my guide throughout my MIT studies. I do not know how to return your favors. Thank you.

To my friends that I have known at MIT: Raj, Stephen, Rami, Ralph, Befekadu, Mohammed, Yassine, Fuad, Saeed, and James. Thank you for being my true friends; each one of you has touched my heart, and you will forever be remembered.

Contents

1	Introduction	15
1.1	MintEra	16
1.1.1	Alloy Annotation Language(AAL)	16
1.1.2	TestEra	17
1.1.3	Minshar	18
1.2	MintEra Example	18
1.3	Thesis Outline	19
2	Minshar	21
2.1	Key Ideas of Minshar	22
2.1.1	Logical vs. Code Constraints	22
2.1.2	Incentive for Documentation	22
2.1.3	Translation into Code	23
2.1.4	Set-based Assertions	24
2.1.5	Exploiting Reflection	24
2.1.6	Granular Verification	25
2.2	Example of an AAL Assertion	25
2.3	AAL grammar	27
2.4	Description of Algorithm	28
2.5	Illustration of Algorithm	30
2.6	Related Work	32

3	TestEra	35
3.1	Alloy Model Generator	36
3.1.1	Example of Model Generation	37
3.2	Alloy to Java Translator (a2j)	39
3.3	Java to Alloy translator (j2a)	42
3.4	Example of Alloy Models Generated by TestEra	43
4	MintEra: Combining Minshar and TestEra	49
4.1	Overview of MintEra	50
4.2	Java Parsing	52
4.3	TestEra Modifications	53
4.4	Visualizer	54
4.5	Comparing MintEra and TestEra	55
4.5.1	Testing	55
4.5.2	Documentation	56
4.5.3	Granular Testing	56
4.5.4	State Handling	57
4.5.5	Locality of Error Detection	58
4.5.6	Run-time Checks	58
5	Discussion and Conclusions	61
5.1	User Experience	61
5.1.1	Personal Experience	61
5.1.2	Novice Java Programmers	64
5.2	Future Work	65
5.2.1	Minshar	65
5.2.2	TestEra	68
5.2.3	Graphical User Interface	69
5.3	Conclusions	71

A	Minshar Algorithm and Helper Functions	73
A.1	Minshar Algorithm	73
A.1.1	int expressions	73
A.1.2	Formulas	74
A.1.3	Declarations	77
A.1.4	expressions	77
A.1.5	Helper Functions	79
A.2	Helper Java Functions	79
B	Alloy Language Specification	81
B.1	Alloy Set Operators	82
B.1.1	Dereferencing (.)	82
B.1.2	Union (+)	83
B.1.3	Difference (-)	83
B.1.4	Intersection (&)	84
B.1.5	Reflexive Transitive Closure (*) and Transitive Closure (^)	84
B.1.6	Set Comprehension ({})	85
B.1.7	if-then-else Condition for Expression	85
B.1.8	Empty Set (none[])	85
B.2	Alloy Boolean Expressions	86
B.2.1	Set Equality (=)	86
B.2.2	Subset Checking (:)	86
B.2.3	Element Containment (in)	87
B.2.4	Integer comparison	87
B.2.5	Logical Operations (&&///<=>/=>,)	87
B.2.6	Quantified Formulas	88
B.3	Alloy Integer Expressions	89
B.3.1	Element Count (#)	89
B.3.2	integer arithmetic(+/-)	90
B.3.3	Java int Variables	90

B.3.4 if-then-else Condition for integers	91
C MintEra Example	93

List of Figures

1-1	Architecture of MintEra	16
2-1	A Linked List Implementation with an Assertion in the <i>remove</i> Method	26
2-2	Minshar Grammar	27
2-3	AST for the Expression <i>all n:this.header.*next n !in n.^next</i>	30
3-1	TestEra Stages	36
3-2	Specification for the method <i>remove</i> in Class List	38
3-3	Alloy Instance	39
3-4	Instances Generated by TestEra	47
4-1	Stages of MintEra	50
4-2	An Example of a Visualized Testcase	55
C-1	Failed Testcases	98

List of Tables

3.1 Alloy Signature to Java Class Map	40
3.2 Alloy Relation to Java Field Map	41
3.3 Instance Table	41
B.1 AAL's Set Operators	82
B.2 AAL's Boolean Operators	86
B.3 AAL's Quantifiers	89

Chapter 1

Introduction

Testing and debugging are important stages of the software engineering cycle. They are the means by which a programmer ensures the correctness of a software system. However, programmers limit or even refrain from testing and debugging since these techniques are usually intellectually unstimulating, repetitive, and labor-intensive.

Currently, most testing of software systems is done manually. Besides the problem of being labor-intensive, manual test-generation is usually not comprehensive. Testers can omit crucial test cases, and thus miss the opportunity to uncover (often serious) bugs in software.

Another problem that software testing faces is design and software revisions. Test cases are usually tailored to the software specification. Thus, when these specifications change due to the natural process of software development, test cases become obsolete and unuseable. These test cases have to be reviewed and corrected as the software itself changes, adding more burden on manual test case generation.

Automated test-case generation solves a lot of these problems. It relieves the programmer from the burden of coming up with test cases, thus making testing much less labor-intensive and more appealing. If done correctly, automated testing can also be comprehensive and adaptable to updates in design specification and changes to the software itself. This thesis presents one such automated test case generator, called MintEra.

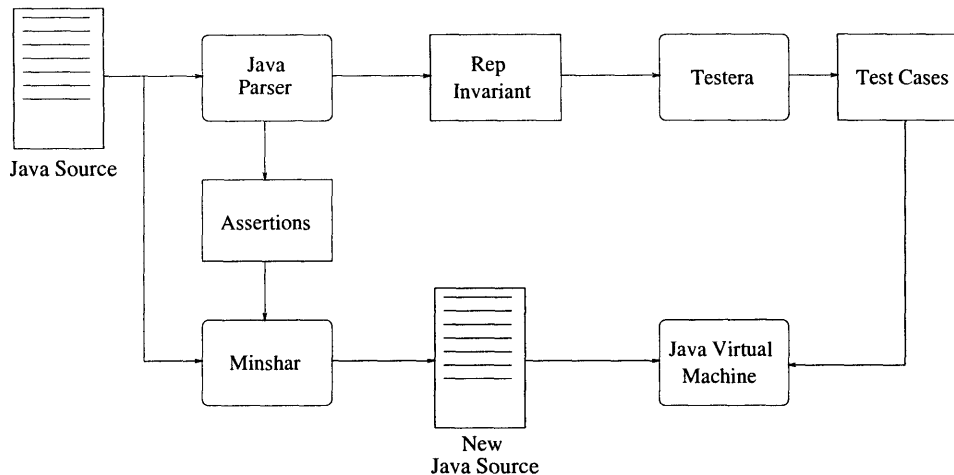


Figure 1-1: Architecture of MintEra

1.1 MintEra

MintEra is a tool for automated test-case generation that aims to be adaptable, easy to use, and comprehensive. The tool uses the Alloy Annotation Language [4] and the Alloy Analyzer [3, 14] as its backbone. MintEra combines two separate tools to achieve better and easier testing: Minshar and TestEra (hence the combined name: MintEra)¹. The next few subsections describe these tools at an elementary level. Please refer to figure 1-1 for the architecture of MintEra.

1.1.1 Alloy Annotation Language(AAL)

MintEra utilizes the Alloy Annotation Language and the Alloy Analyzer to write specifications as well as generate test cases. AAL is an annotation language that is designed for Java programs and is based on the Alloy [2, 14] modeling language. AAL annotations are embedded in Java code and look like comments to the compiler. These assertion, however, can be utilized by MintEra to describe properties of the software. AAL only uses a subset of the Alloy language to describe these properties.

¹The first tool developed under the automated testing project was TestEra, and it was supposed to indicate a new “Test Era”. The same term meant “saw” in Serbian, which was the native language of one of the founders of the project. The automated testing project was then named the MulSaw [16] project and every tool that was developed later on was named saw in the language of the developer. Minshar means saw in Arabic.

Alloy is a modeling language that is used to specify complex structural properties of software. Alloy formulas are expressed using first order logic (including quantifiers such as \forall and \exists) and relations, making Alloy a powerful and accessible language. The user can define signatures (which are sets²) and relations to describe a model. Users can also constrain a model by including “facts”, which are formulas that are always true.

Models written in Alloy can be verified using the Alloy Analyzer(AA). AA translates the Alloy specification into a SAT formula, and uses off-the-shelf SAT solvers to find counter examples to assertions within the model. The Analyzer can also be used to come up with examples of the model being analyzed.

1.1.2 TestEra

TestEra [5, 6] is a test-case generator and verifier that is also based on the Alloy Analyzer. TestEra in its original form uses method pre and post conditions that are written as Alloy specifications to generate and verify testcases.

From the Java `.class` file, TestEra builds an Alloy model of the Java class being analyzed (as well as any supporting classes.) The Alloy-model generator maps classes and field in Java into signatures and relations in Alloy. TestEra also builds the input model for the Java class based on the pre-condition of the method being tested. The tool finally builds an output-module of the Java class that is based on the method’s post condition.

This version of TestEra does testing by utilizing the Alloy Analyzer and the input/output modules. The input module, as well as the Alloy model of the classes, are used to generate “examples” in Alloy that fit the method’s pre-condition. These abstract test-cases are then translated into concrete Java test-cases using the Alloy-to-Java (a2j) translator. The method is then tested on the concrete Java test-cases, and the state of the program after running the method is translated into Alloy using the Java-to-Alloy (j2a) translator. The output module is then invoked on the result

²Alloy does not have any distinction between sets and one-tuple relations. Signatures are more accurately considered to be a one-tuple relation.

of the translation to verify that the testcase fits the post-condition of the method.

While MintEra uses the test-case generation aspect of TestEra, it does not use the verification aspect of the tool. MintEra alternatively utilizes an assertion-based verification that is rooted in the Java method itself. The advantages and disadvantages of using both of these methods are explained in chapter4.

1.1.3 Minshar

The second element of MintEra is an assertion-translator tool named Minshar [17]. Minshar translates Alloy assertions that are embedded in methods into Java code. These assertions are simply Alloy boolean formulas that evaluate to true or false. Minshar thus translates these assertions into Java code that will eventually test a condition and throws a `MinsharAssertionFailedException` if the formula evaluates to false.

Minshar utilizes the Alloy Analyzer to translate assertions. It uses the Alloy Analyzer’s parser to translate the assertion specifications into an Alloy Abstract Syntax Tree (AST). Minshar then traverses this AST and translates it into Java. When faced with a certain grammar production, Minshar translates that production’s subexpressions, groups these translations together, and adds production-specific code to generate the production’s translation.

Note that MintEra doesn’t require the user to write assertions in Alloy. The user can simply write these assertions in Java. The only requirement is that these assertions should throw an exception when they fail so that the tool can recognize a failed test case from a passing one.

1.2 MintEra Example

To illustrate the main concepts of MintEra, we show an example that we visit throughout this thesis. Consider the Linked List implementation below, annotated with AAL specification:

```

public class List {
    /*@
     repok: some m: this.header.*next| no n.next
     */

    public class Node {
        public Node next = null;
        public Object element = null;
    }
    public Node header;

    public boolean remove(Object e){
        Node n,p = header;
        /*@
         assert: some this.header
         */
        for (n = header.next; n!=null; n=n.next) {
            if(n.element.equals(e)) {
                p.next = n.next;
                return true;
            }
            p = n;
        }
        return false;
    }
}

```

The representation-invariant, written as an annotation with *repok* preceding it, is that the linked list is acyclic. Upon running MintEra, it generates testcases that fit the representation-invariant. Also, the tool translates the assertion within the method *remove* into a Java run-time check. The assertion specifies that the *header* field is not *null*.

MintEra runs the method on the testcases generated earlier and detects any testcases that fail (i.e. ones that throw a run-time exception). In this example, we get a number of testcases that do not pass the assertion within the *remove* method. By looking closely at the implementation, we see that *header* was not initialized at any point in time. This common error is detected by MintEra by only providing the representation-invariant and assertions within code.

1.3 Thesis Outline

This thesis aims to describe MintEra: how it works, what tools it depends on, and how these tools are combined. The next chapter describes the Minshar tool, and how it translates Alloy assertions to Java run-time checks. Chapter 3 describes the

TestEra tool and how it generates test cases from representation invariants. Chapter 4 talks about combining Minshar and TestEra, and how MintEra is different from the original TestEra that was developed earlier. Finally, Chapter 5 looks at the experiences of novice Java programmers using MintEra to come up with test cases. This chapter also discusses future work and conclusions.

Chapter 2

Minshar

Assertions in a program module describe a property of the module in terms of program entities, such as local or instance variables. Assertions specify *what* the module should do rather than *how* it should be done. They are often expressed in code as annotations that are viewed by the compiler as comments. Since these annotations provide a more precise description of what the program should do, they are used to improve program documentation and as a design aid. Assertions are also useful as debugging and partial correctness checking mechanisms by translating these annotations into run-time checks. Programmers use annotations as a safety mechanism to ensure correctness of reused code; programmers can defend their code from being used incorrectly by other programmers (e.g., by checking preconditions of a methods). For more information about assertions and their history, please check the introduction of Rosenblum [8].

The Alloy Annotation Language (AAL) is one such language that is designed to be used with Java programs. Our tool, MintEra, uses AAL assertions that are embedded in Java code to express specifications. However, these assertions need to be translated to Java in order to check a certain property of the software.

The Minshar tool (which is part of MintEra) translates AAL assertions into Java run-time checks. In this chapter, we explain the Minshar tool in detail. First, we discuss the key ideas of Minshar, then we explore how Minshar translates AAL assertions into run-time checks.

2.1 Key Ideas of Minshar

Using assertions within code to specify constraints is not a new concept. However, Minshar provides a number of key ideas that are –for the most part– unique. In this section we describe some of these key features.

2.1.1 Logical vs. Code Constraints

Assertion languages currently are either expressed logically or in code. Logical assertions are succinct, simple and easy to write. Examples of logical assertions are formal constraints (written in languages like Z or VDM) and English constraints. Formal languages are high-level, making them easier to read and write. The problem with most logical assertions is that they cannot be checked at runtime (e.g because of quantifiers). They are mainly used for documentation and not for run-time checking.

On the other hand, assertions written in code are executable but are not easy to write. Code assertions are simple conditionals written in imperative languages (such as Java). Since code assertions require a long section of low-level instructions to specify complex constraints, they generally tend to be harder to read. Thus, code assertions cannot be used for documentation.

There is a need for a more declarative yet verifiable assertion language. In such a language, assertions can be written and understood easily, yet they can be translated into code and checked at runtime. Minshar provides such a declarative and verifiable language (AAL) that is designed to be used with Java programs. AAL limits the annotation language essentially to navigations, and quantifiers over navigations – which is similar to the first version of Alloy (Alloy 1). This limitation allows users to write concise annotations that can be translated into code. Thus, AAL has the best of both logical and code worlds: documentation and verification.

2.1.2 Incentive for Documentation

One important feature that Minshar facilitates is documentation. As explained earlier, AAL is a concise, easy and expressive language, making it ideal for documen-

tation. Programmers can write AAL annotations to comment on what their code should be doing (i.e. assertions).

The problem is that programmers usually do not have an incentive to document their code, even with a powerful language such as AAL. Minshar motivates programmers to document their code by translating their assertions to run-time checks. From a user's point of view, documentation using AAL assertions helps them debug their code (and even test it). Thus, Minshar encourages better software engineering practices.

2.1.3 Translation into Code

AAL provides an annotation language that can be translated into Code. Minshar implements this translation algorithm by picking up assertions, which are embedded within Java, and parsing the Alloy formula. The tool then translates a certain expression by first translating each of its subexpressions into Java and then combining the resulting code into one coherent fragment.

After Minshar translates the whole expression, the resulting Java code is inserted immediately after the assertion. User can determine where their checks are executed by simply inserting an AAL assertion at that point.

Notice that assertions could be “turned on” or “turned off” by simply translating the assertions or removing the translation (which is done automatically). The code is not cluttered with run-time checks that are written in code, yet the user has the ability to verify these assertions. Since annotations are simply comments, there is a clear distinction between verification and the program itself. That is, the user does not have Java run-time checks that do not advance the state of the program. We believe that this distinction is very helpful for the user, from a software engineering prospective, in understanding the clear division between implementation and verification.

2.1.4 Set-based Assertions

Since Alloy is a relational language, its formulas manipulate relations (called expressions) to evaluate a condition. Relational logic is one of the main reasons Alloy formulas are expressive. Manipulating relations through navigations and quantifiers over navigations allows users to construct complex relations, and check certain properties on them.

Since AAL caters to Java, it reduces Alloy's relational logic to a set-based one. Even though relational logic could be used to express constraints, set logic is sufficient to express these constraints when dealing with Java. We have thus opted to use set-based assertions in Minshar. Java objects can be seen as singletons (one-element sets) and using navigation (including reflexive and transitive closures) users can construct more sets, which they can then use to build constraints.

Minshar translates Alloy expressions so that they are computed explicitly at runtime. In the translation algorithm, Minshar uses Java *Sets* to represent Alloy expressions. Sets are constructed to wrap starting objects, and through navigation these sets are used to construct more sets. The translation mirrors AAL's set-based assertions, making the translation simple.

Notice that evaluating sets could be done in a lazy fashion. That is, the explicit evaluation of sets could be bundled together. Through this optimization, evaluating costly Alloy expressions could be done much faster. For more information about this optimization, check 5.2.1.

2.1.5 Exploiting Reflection

Minshar uses a library of methods to implement complex features such as reflexive closure. Reflexive closures navigates through a field (or a number of fields) to include every element reachable from the first element, and so they could run into loops and/or null pointers. Implementing reflexive closures require keeping a *visited* list as well as keeping track of null pointers. Since this code repeats often, it should be in its own method.

The problem is that navigation is dependent on the fields of the object itself. Thus, it is impossible to implement a standard library depending on Java's inherent dereferencing. To solve that problem, we have utilized the Java Reflection package. The reflexive closure method takes in a *String* array of fields as well as the object to navigate. The method uses the Reflection package, which accepts the field names as *Strings*, to navigate the object and returns the resulting set. Similarly, other complex operations are implemented as standard libraries by a similar exploiting of Reflection.

2.1.6 Granular Verification

One important feature that Minshar provides is granular verification. Since assertions can be inserted at any point in code, users can track down location of bugs in their programs by adding more AAL constraints. When an error occurs, assertions around the error location will detect that error (if these assertions are written correctly). This helps users reduce the amount of time they spend on debugging by depending on Minshar's verification mechanism.

There are other advantages of adding AAL assertions within code. Adding assertions further documents code and makes understanding it easier. On the other hand, adding extra code assertions clutters the code and forces the user to either comment them out or remove them altogether, which wastes the time and effort spent to come up with these assertions. AAL assertions are simply comments, and thus they can stay within the code. Thus, programmers can use them at any point in time for verification.

2.2 Example of an AAL Assertion

In this section, we provide an example of AAL assertions and how they differ from assertions written in Java. Figure 2-1 contains an implementation of a linked list as a Java class. The method *remove* within that class contains the AAL assertion:

```
all n: this.header.*next| n !in n.^next
```

```

public class List{
    Node header = new Node();

    public static class Node {
        Node next;
        Object element;
    }

    public boolean remove(Object e){
        /*@
        assert: all n: this.header.*next|n !in n.^next
        */
        Node n, p = header;
        for(n = header.next; n!=null; n = n.next){
            if(n.element.equals(e)){
                p.next = n.next;
                return true;
            }
            p = n;
        }
        return false;
    }
}

```

Figure 2-1: A Linked List Implementation with an Assertion in the *remove* Method

We have inserted this assertion to ensure that the list is acyclic¹. We need this check to avoid running into an infinite loop when searching for the element to delete.

The Alloy statement constructs a set of all nodes reachable from the header (*this.header.*next*). For every element *n* in this set, we want to check that *n* is not reachable from itself following the *next* field one or more times *n !in n.^next*. Now consider implementing the same constraint in Java. Here is a segment of Java code that implements it:

```

Node n;
boolean result = true;
HashSet visited = new HashSet();
for(n=header; n!=null; n=n.next) {
    if(visited.contains(n)) {
        result = false;
        break;
    }
    visited.add(n);
}
if(!result)
    throw new Exception();

```

¹We could use the simpler constraint *some n: this.header.*next|no n.next* to express acyclicity. However, we opted to use a more general acyclicity constraint to show the translation algorithm in more detail.

```

formulabody = formulaseq || formula
formulaseq = {formula...}
formulaseq = expr compop expr | intexpr intcompop intexpr
              | (formula) | neg formula | formula logicop formula
              | formula thenop formula [elseop formula] | formulaseq
              | quantifier decl,... formulabody | quantifier expr | let letdecl,...formulabody
intexpr = number | #expr | (intexpr) | let letdecl,... | intexpr
              | if formula then intexpr else intexpr | intexpr intop intexpr
expr = var | expr [expr] | (expr) | none [expr] | {decl [formulabody]}
              | if formula then expr else expr | let letdecl,... | expr
              | expr + expr | expr - expr | expr & expr | expr.varplus | expr.unop varplus
varplus = var | (var+..)
decl = [qualifier] var,... : expr
letdecl = var=expr
intop = +|-
thenop = => | implies
elseop = , | else
neg = not | !
logicop = && | || | iff | <=> | and | or
quantifier = all | no | some | sole | one | two
unop = * | ^
compop = [neg] (: | = | in)
intcompop = [neg] (= | <= | >= | > | <)
qualifier = part | disj | exh
var = id

```

Figure 2-2: Minshar Grammar

The above example enforces our earlier claims about code assertions versus AAL assertions. Expressing properties in AAL is simpler and easier to read than implementing them in Java. Moreover, we still retain the ability to check these assertions by translating them through Minshar.

2.3 AAL grammar

As we explained earlier, Alloy depends heavily on relational logic but Java only requires a set-based one. For that reason (and for reasons related to the Java language itself ²), AAL reduces Alloy's relational logic and operations into set logic and operations. Removing support for relations requires rewriting the Alloy grammar to conform to AAL. The reduced grammar does not support unused operators or operators that are hard to implement. Figure 2-2 provides the complete AAL grammar.

²Supporting any of Alloy's relational operators require access to the Java garbage collector in order to build the relations. For example, in our linked list implementation, the relation *element* has type (Node, Object), which relates a node to its element. We cannot build this relation since we need the Java garbage collector to retrieve all the (Node, Object) pairs.

Expressions in AAL denote sets, which can be viewed as relations with one column. These sets are obtained starting from variables viewed in the current scope of the program. Through navigation and set operations, new sets are created from the original sets. Navigation (\cdot) accesses a field in each element of a set of objects. Reflexive closure ($*$) continuously navigates a field in an object, until that field is *null*, or all objects have been navigated. For example, *this.header.*next* represents the set $\{header, header.next, header.next.next, \dots\}$. On the other hand, transitive closure ($\hat{\cdot}$) is the same as reflexive closure but without including the object itself. For example, *this.header.^next* represents the set $\{header.next, header.next.next, \dots\}$.

2.4 Description of Algorithm

This section describes the algorithm that translates AAL annotation to Java runtime checks. The algorithm assumes that the input, which is an AAL annotation, is parsed into an abstract syntax tree. It translates each grammar production into a pair of strings. The first element of the pair is the name of a variable that holds the result of the expression, while the second element is the actual Java translation. The algorithm calls itself recursively on the production's subcomponents. For example, when translating

```
formulaseq = expr1 compop expr2
```

the algorithm calls itself for translating *expr1*. The result of the call is a pair $\langle v, c \rangle$ where variable *v* holds the result of translating *expr* and *c* is the actual Java code for evaluating the expression. The algorithm uses *v* to evaluate *decl*, and appends its own Java code to *c*.

The reduced Alloy grammar is divided into three main types: formula, intexpr, and expr. Thus each expression evaluates to a variable of one of these types; formulas evaluate to boolean variables, intexprs evaluate to integer variables, and expressions evaluate to set variables.

As an example of the translation algorithm, consider the production *intexpr intop intexpr*. In this production, *intop* is either an addition or a subtraction. The expression adds (or subtracts) one *intexpr* to (from) the other. Here is a portion of the algorithm that translates this production:

```
Ti(intexpr1 intop intexpr2) {
  v = freshvariable("int")
  <v1,c1> = Ti(intexpr1)
  <v2,c2> = Ti(intexpr2)
  return <v, "c1 c2 v=v1 intop v2;">
}
```

Tx represents a translation of a certain type of an expression, which is abbreviated by *x* (i for *intexpr*, e for *expr*, d for *decl*, and f for *formula*). The function *freshvariable()* returns a string representing a unique variable name, and saves the variable type in order to insert the variable declaration at the top of the translation. *<v1,c1>* represent the pair returned by the recursive call to the algorithm, where *v1* is the variable returned, and *c1* is the code associated with this subexpression translation. The function returns a similar pair.

AAL has support for declaring variables (such as *n: this.header.*next*). One use of declared variables is to simplify expression-writing (e.g. using *lets*). For example, the expression *let elts: header.*next.element | <expr>* defines *elts* to be the relation *header.*next.element*. The variable *elts* can be used within the *expr* following this declaration. Declared variables can also be used within quantifiers and set-comprehension operations. For example, the formula *some n: header.*next | <formula>* declares *n* to represent all elements within the set *header.*next*.

Keeping track of declared variables requires introducing a Variable Symbol Table (VST) that maps formula variables to Java variables that represent them. This VST is needed in order to replace any future occurrence of the formula variable (*n* in the above example) with the actual Java variable.

Note also that a hierarchy of VST's is needed in case there are nested declarations. Each VST has a parent VST that is its parent scope. To look up a variable, the search begins at VST that associated with the current scope. If the variable is not found in that VST, then the search moves to the parent VST and so on. If the lookup fails, it

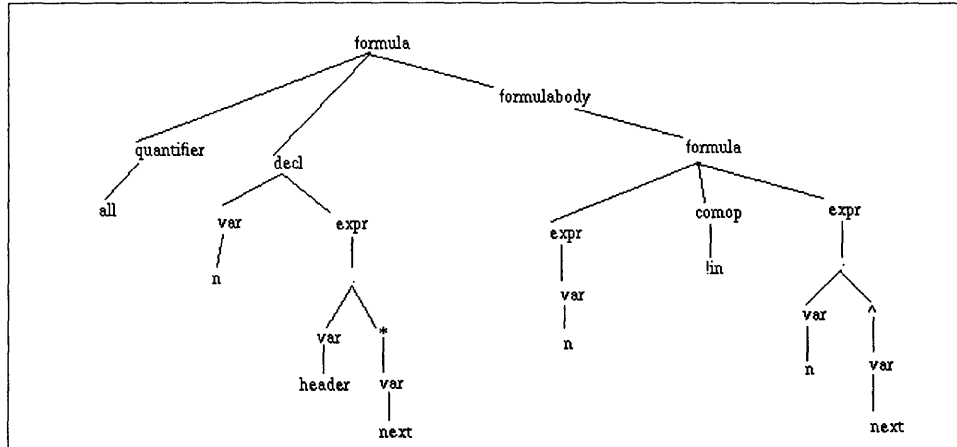


Figure 2-3: AST for the Expression `all n:this.header.*next | n !in n.^next`

is assumed that the variable in consideration is a Java one, and the variable name is not changed.

For the complete algorithm, refer to Appendix A.

2.5 Illustration of Algorithm

To demonstrate how the algorithm works, we show how it translates the linked-list acyclicity assertion explained earlier. Recall that the assertion was written as:

```
all n: this.header.*next | n !in n.^next
```

The Abstract Syntax Tree (AST) representation of the above assertion can be seen in figure 2-3.

The Algorithm translates the formula from leaves to parents. We will first translate the expression `this.header.*next`. The algorithm will produce the following code:

```
v_3=Util.singleton(this);
v_2=new HashSet();
v_2.addAll(Util.dereference(v_3,"header"));
v_1=Util.traverseStar(v_2, new String[] {"next"});
```

The function `traverseStar` makes a breadth first search starting from the header node, obtaining all elements that follow from it by applying the `next` relation. Next,

we translate the whole *decl* expression $n: this.header.*next$. This expression translates to the following code segment, which is appended to the code above:

```
v_6=v_1.iterator();
l_4: while(v_6.hasNext()) {
    v_5=Util.singleton(v_6.next());
```

Notice that the close parenthesis for the while has not been inserted, because we will be include the translated formula ($n \text{ !in } n. \hat{next}$) inside the while loop. This while loop iterates through the elements of the set $this.header.*next$, testing the formula on each element.

Next, we translate the expression $n. \hat{next}$ that occurs at the end of the expression:

```
v_8=Util.traverseCaret(v_5, new String[] {"next"});
```

The method *traverseCaret* includes a breadth first search to obtain all elements accessible from the header node, following the field *next* one or more times. The code for *traverseCaret*, as well as *traverseStar* can be found in Appendix A. Finally, the algorithm translates the formula $n \text{ !in } n. \hat{next}$ to the following Java code:

```
v_7=!v_8.containsAll(v_5);
```

To finish the translation of the whole formula ($all\ n: header.*next \mid n \text{ !in } n. \hat{next}$), we combine all the code fragments together, producing the overall run-time check (notice the declarations at the top):

```
Iterator v_6;
Set v_1,v_2,v_3,v_5,v_8;
boolean v_0,v_7;
v_0=true;
v_3=Util.singleton(this);
v_2=new HashSet();
v_2.addAll(Util.dereference(v_3,"header"));
v_1=Util.traverseStar(v_2, new String[] {"next"});
v_6=v_1.iterator();
l_4: while(v_6.hasNext()) {
    v_5=Util.singleton(v_6.next());
    v_8=Util.traverseCaret(v_5, new String[] {"next"});
    v_7=!v_8.containsAll(v_5);
    if(!v_7){
        v_0=false;
        break l_4;
    }
}
if(!v_0)
    throw new MinsharFailedConstraintException();
```

As the tool currently stands, the performance of the translated code is relatively slow. From the translation above, notice how the translated code builds the expressions one at a time. For example, the code inserts the variable *this* in a singleton, then navigates this set with the field *header* to result in another set. Such simple one-at-a-time evaluation of expressions is slow. As we explain in section 5.2.1, we plan to optimize set-construction so it is evaluated in a lazy fashion. However, we doubt that the running time of translated code will be comparable to code assertions since AAL still requires sets to be built explicitly.

2.6 Related Work

We have already compared AAL to formal specification languages (such as Z [9] and VDM [11]). To recap, AAL limits the formal annotation languages to navigation and quantifiers over navigations, enabling these specifications to be executable. Even though writing assertions in such formal languages might be more succinct, they are in general not executable.

We have also compared AAL to code constraints. Code assertions are large in size, and subsequently would make reading them harder. They clutter the program with code that is not used to advance the state of the program. Assertions written in AAL are simpler, more succinct and more intuitive than code assertions. Minshar can also translate AAL assertions into run-time checks that verifies a condition. Thus, AAL provides users with the right balance between documentation and executability.

AAL is not the only high-level language that support run-time checks. Other widely-used annotation languages are the Java Modeling Language (JML) [1], Eiffel [7] and the Object Constraint Language (OCL) [12]. Here, we compare AAL and Minshar to Eiffel and JML.

Eiffel is an object oriented language that includes design by contract concepts (i.e. pre- and post-condition agreements) in the language itself. Eiffel allows users to express assertions within code. Pre-conditions, post-conditions, and assertions are checked at the beginning, end, and within methods. Eiffel introduces the concept

of inheriting assertions where assertions from the parent class are inherited into the child class. This powerful concept helps users ensure that methods within child-classes obey the constraints of the parent methods. While Minshar currently does not include assertion inheritance, it could adopt some of Eiffel’s inheritance techniques.

Unlike Minshar, Eiffel does not support any notion of quantifiers. Using quantifiers within assertions helps users express complex constraints easily. Moreover, Eiffel lacks support for implicit traversal of objects (transitive and reflexive closures). Minshar also depends on relational expressions, which are absent from Eiffel. The lack of such features makes Eiffel’s assertions closer to code constraints than logical ones. However since Eiffel’s assertions are closer to code, they run much faster than ones written in Minshar.

JML is a modeling language that is designed specifically for Java. It extends the design by contract and inheritance concepts, introduced by Eiffel, by adding a number of features. JML uses more expressive logical constructs such as quantifiers within constraints. It also allows users to compare old and new values of variables. Specifically, it uses the `\old` keyword to refer to variables in the pre-state. JML also provides a powerful cross-referencing mechanism by referring to specification from other JML-annotated classes. It uses the specification information to simulate the method’s behavior.

While JML is more expressive than Eiffel, it is still not as expressive as AAL. JML has a limited support for quantifiers, and does not have any notion of implicit traversal or relational expressions. Moreover, neither JML nor Eiffel supports fully automatic compile-time analysis of the specifications. For example, checking at compile time that the precondition of an overriding method is weaker than the precondition of the overridden method is impossible in JML or Eiffel. Minshar currently does not implement cross-referencing or variable state-comparison, but as we explain in 5.2, we intend to implement these features soon. In fact, we propose a more general concept of state comparison that compares variables in more than two states. Finally, assertions written in JML run faster than ones written in AAL. One reason is that JML specification expressions are a subset of the Java language.

The Object Constraint Language (OCL) is the constraint language of the Unified Modeling Language (UML). OCL has a lot of the features AAL does. Currently, OCL tools can check assertions within code but cannot generate instances based on certain constraints. On the other hand, the Alloy Analyzer can generate instances, and subsequently testcases. Since automatic testcase generation is one of the goals of MintEra, we opted to use AAL as our specification language.

Chapter 3

TestEra

We introduce TestEra, which is one of the tools that MintEra depends on for testcase generation. TestEra was developed by Sarfraz Khurshid as part of his Ph.D. thesis [6] at MIT. We have used Sarfraz’s tool and modified it to fit MintEra’s framework.

TestEra is a testcase generator and verifier that uses the Alloy language and the Alloy Analyzer. Since MintEra uses a different mechanism for verification than TestEra, we need to understand how TestEra was originally developed to evaluate both approaches of testing. In this chapter, we describe the original TestEra tool as it was developed by Sarfraz.

TestEra’s approach to testing is method-based. It generates testcases for a specific Java method, runs these testcases on that method and verifies that these testcases fit a certain criteria. TestEra uses the method’s specification for testcase generation and verification; the tool generates testcases that fit the method’s precondition, and verifies the method by checking the output against the post-condition.

The tool goes through several stages in the process of generating and running testcases. First, TestEra generates full Alloy models given the Java classes and the pre/post conditions. The tool then runs the input Alloy model to generate testcases. Since these testcases are in Alloy format, they need to be translated to Java concrete objects (done by the Alloy-to-Java translator). Now the Java method is run on these testcases. Afterwards, the concrete Java objects are converted back to Alloy (using the Java-to-Alloy translator) where they are verified using the output model generated

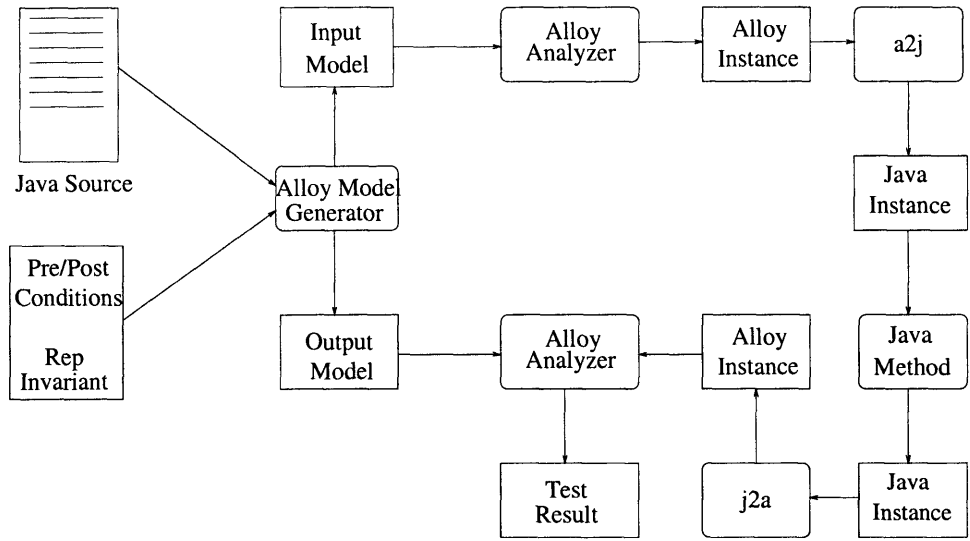


Figure 3-1: TestEra Stages

earlier.

We continue to describe each of these stages in more detail. Figure 3-1 shows the various stages of TestEra.

3.1 Alloy Model Generator

TestEra depends on the Alloy Analyzer (AA) to generate testcases that fit the Java class being tested. Given a certain Alloy model, AA can construct “examples” that fit the model and any additional constraints (or “facts”). After running these testcases on the Java method, TestEra relies on the AA to verify that these testcases satisfy the post-condition. The problem remains in constructing Alloy models –given the Java classes– that the AA can use to construct testcases and verify them.

The tool starts by generating Alloy models that represent the Java classes. TestEra parses the compiled Java files (i.e. `.class` files) to get information about the classes, fields, and methods of the Java program. The tool then constructs an Alloy model for each class. Classes and fields are represented by Alloy signatures and relations respectively.

The tool proceeds to make two extra models: an input and output model, which

use the class models that were just generated. The input model contains the pre-condition of the method, and is used to generate testcases. On the other hand, the output model is used to ensure that the testcases satisfy the post-condition after the method has been run on them.

Note that TestEra has two different states of operation: before running the method for testcase generation (pre-state), and after running the method for verification (post-state). There is usually a need to compare the pre-state to the post-state. For example, after a “pop” operation on a stack, the stack size should have decreased by one. Since Alloy does not handle state inherently, the models have to include a *State* variable that would distinguish the pre-state from the post-state, so that the user could compare the two.

The user could access the variable in the pre-state by adding a (') at the end of the variable name. The tool would then add the appropriate suffix to the relation to indicate that it is evaluated in the pre-state. For example, *header'* accesses the header variable in the pre-state. On the other hand, if the user types *header* without the (') symbol, the tool would evaluate the relation in the post-state.

3.1.1 Example of Model Generation

We will continue discussion of Alloy model generation through an example. We look at a linked list implementation with the following Java code:

```
package util;
public class List {
    public class Node {
        public Node next = null;
        public Object element = null;
    }
    public Node header = new Node();

    public boolean remove(Object e){
        Node n,p = header;
        for (n = header.next; n!=null; n=n.next) {
            if(n.element.equals(e)) {
                p.next = n.next;
                return true;
            }
            p = n;
        }
        return false;
    }
}
```

Representation-invariant:	<code>some n: this.header.*next no n.next</code>
Pre-condition:	<code>#this.header.*next > 1</code>
Post-condition:	<code>e !in this.header.*next.element</code>

Figure 3-2: Specification for the method *remove* in Class *List*

To run TestEra, the user provides the representation invariant, the pre-condition, and the post-condition to the tool. For the linked list example above, assume that the user supplies the specification described in figure 3-2. Here, the representation-invariant is that the list is acyclic, the pre-condition is that the size of the list is at least one, and the post-condition is that the removed element (in this case *e*) is no longer in the list.

TestEra uses the `.class` files generated by the implementation above to construct two Alloy models: one for the class *List*, and one for the class *List.Node*. We describe the exact Alloy models that are generated in section 3.4. Here, we describe the signatures and relations that are found in the generated models.

First, TestEra generates the Alloy model associated with the class *List.Node*. The only signature within that Alloy model is *List_Node*, which represents the *List.Node* class. Basically, the members of the signature *List_Node* represent Java Objects that have the type *List.Node*. The *Node* model also defines two relations: *next* and *element* representing the next and element fields respectively. Notice that *next* is a relation from *List_Node* to *List_Node*, mapping a certain *Node* to its successor. Similarly, the *element* relation is a map from *List_Node* to *Object*.

TestEra also generates a model for the *List* class. Within that model, a signature called *List* is defined, which represents the *List* Java class. There is also a relation called *header* (representing the header field), which is a map from *List* to *List_Node*.

The input model defines a function called *inputConstraint* which includes the pre-condition of the method, as well as the representation invariant. The output model, on the other hand, defines a function called *methodOK* which has both the representation invariant and the method's post-condition. Both models also contain

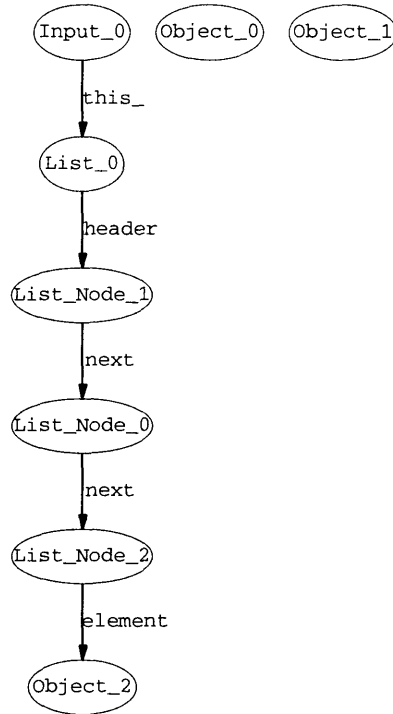


Figure 3-3: Alloy Instance

a static signature (i.e. one-element set) called *Input*. These models also define the arguments and the return value as relations from *Input* to the type of these variables. Finally, the input and output models contain a relation called *this_* that is analogous to the *this* variable in Java.

3.2 Alloy to Java Translator (a2j)

After generating Alloy models from Java classes and pre/post conditions of the method being tested, TestEra runs the Alloy Analyzer on these models to automatically generate examples. The input model asks the analyzer to run the *inputConstraint* function. The Analyzer thus constructs examples that fit the input constraint (including the representation invariant).

The examples that the AA constructs are Alloy instances. These instances are examples that fit the Alloy model and any other constraints imposed by that model. Consider our linked list example in section C. An Alloy instance of the input model

Alloy Signature	Java Class
List	List
List_Node	List.Node
Object	java.lang.Object

Table 3.1: Alloy Signature to Java Class Map

is a linked list that fits the model’s constraints. For example, Alloy could generate an instance similar to that of figure 3-3. In this instance, the *List* signature would contain one list {List_0}, and the *List_Node* signature would contain 3 objects {List_Node_0, List_Node_1, List_Node_2}, and the *Object* signature contains 3 objects: {Object_0, Object_1, Object_2}. Now the relations have to be also filled, so the *next* relation would contain the tuples: (List_Node_1, List_Node_0), and (List_Node_0, List_Node_2). The *header* relation contains one tuple: (List_0, List_Node_1), and finally the *element* relation would only contain the tuple: (List_Node_2, Object_2)¹.

While these Alloy instances are the testcases that we are looking for, they are in abstract form. The tool needs to convert these examples into Java objects, and fill in the appropriate fields of these objects depending on the values of the relations. The Alloy-to-Java (a2j) stage of the tool takes care of this translation.

When TestEra generates the Alloy models, it keeps a table that maps Alloy signatures and relations to Java classes and fields. The tool uses this information for converting the abstract Alloy instances into concrete Java objects. For example, given the linked list implementation above, TestEra has a mapping from *List_Node* (the Alloy Node signature) to *List.Node* (the equivalent Java class). The tool also has a mapping from the Alloy relation *next* to Java field *List.Node.next*. TestEra keeps signature-class and relation-field maps similar to tables 3.1 and 3.2 respectively.

The first step in the a2j translator is to create Java objects. TestEra looks at each signature and finds the Java class that represents it from the signature-to-class table mentioned above. The tool then goes through the list of objects of that signature,

¹All relations in TestEra have a *State* variable to indicate what state these relations are evaluated. We have omitted them from the above relations for simplicity.

Alloy Relation	Java field
header	List.header
next	List.Node.next
element	List.Node.element

Table 3.2: Alloy Relation to Java Field Map

Alloy Object	Java Object
List_Node_0	JN_0
List_Node_1	JN_1
List_Node_2	JN_2
List_0	JL_0
Object_0	JO_0
Object_1	JO_1
Object_2	JO_2

Table 3.3: Instance Table

and creates new Java objects with the corresponding Java type. For example, for every Alloy object that belongs to the signature *List_Node*, TestEra constructs a new object of type *List_Node*.

TestEra also keeps an *instance* table that maps the Alloy instances to the newly created Java objects. This is an instance-to-Object map rather than the signature-class and relation-field map mentioned earlier in this section. For example, if the tool sees an instance of *List_Node* (like List_Node_1), it constructs a Java Object of type *List_Node* (call it JN_1). The tool then adds the pair (List_Node_1,JN_1) into the instance table. The instance table for the example in figure 3-3 looks like table 3.3

One of the main uses of the instance table is to fill field information based on Alloy relations. For example, assume you have the tuple (List_Node_1, List_Node_2) in the *next* relation, where List_Node_1, List_Node_2 are *List_Nodes*. When TestEra wants to convert this tuple, it first looks into the instance table and retrieves the Java objects that map to List_Node_1 and List_Node_2 (call them JN_1, and JN_2 respectively), then the tool sets the *next* field of JN_1 to be JN_2. Notice that TestEra keeps the signature-class, relation-field and the instance maps for use in later stages (namely the Java to Alloy translator).

3.3 Java to Alloy translator (j2a)

After TestEra has successfully converted the Alloy abstract instances into concrete Java objects, we can now run the method on the testcases. The tool finds the Java object that maps to the *this_* relation in Alloy and runs the specified method on it. Since the testcases have been produced based on the input model, then the testcases fit the pre-condition of the method and the method should run successfully (unless there is a problem in the specification).

After running the method on this testcase, the next step would be to verify that the testcase did not produce any errors. The output model that we generated earlier is our mechanism for verification. However, this output model is written in Alloy, so we need to convert the Java objects back to Alloy instances in order for us to run the output model on them.

The Java-to-Alloy translator (j2a) is similar to the Alloy-to-Java one. It depends on the tables that were constructed earlier to convert the Java objects back to Alloy. For every Java class, it finds the corresponding Alloy signature from the signature-class mapping. It then looks at every Java object and finds the corresponding Alloy object that maps to it². Finally, the a2j translator fills in the Alloy relations given the field information from the Java objects.

TestEra has to take care of two issues. First, if the Java method created an object, the tool needs to add an object to the corresponding Alloy instance. For example, given a linked list implementation, an insert method would introduce a new *Node* to the list. When converting the list back to Alloy, a new *List_Node* object is created, and is mapped to the new Java *Node* in the instance table.

The second issue that TestEra faces is mutation. A change in the relations that is due to mutation in Java objects needs to be reflected in Alloy. TestEra thus constructs these relations based solely on the Java fields and the *instance* table, without any consideration to the old relations in the pre-state. For example, in a remove method of a linked list, the *next* relation of this list will be affected by the removed node.

²Notice that TestEra uses object equality (i.e. `==` instead of `.equals()`) to compare objects.

Thus, we can not use the *next* relation that was created before the method ran.

Notice that the instances from the pre-state are kept in the post-state. These instances are used for comparing the two states together since some specifications require such a comparison. For example, if we are testing our remove method from our linked list implementation, then we might want to check that the size of the new list is, at most, one less than the old list's.

After the j2a translation is complete, we can now run the Alloy Analyzer on the Alloy instances that were just constructed. The analyzer verifies that this instance fits the post-condition of the method. TestEra keeps track of the number of tests that pass versus the number of tests that fail, and gives this summary to the user. Thus, the tool completes the test-case generation and verification process. Figure 3.4 shows a number of testcases for our linked list example.

3.4 Example of Alloy Models Generated by TestEra

In this section, we describe the models outputted by TestEra's Alloy model generator. As explained earlier, TestEra generates a model for each Java class, as well as an input and output model. We use the linked list implementation of section C as an example. First, TestEra generates the *List_Node* model representing the *List.Node* class in Java. It is:

```
module util/List_Node

open testera/mutation/State
open java/lang/Object

sig List_Node {
  next: util/List_Node/List_Node ?-> testera/mutation/State/State,
  element: java/lang/Object/Object ?-> testera/mutation/State/State
}
```

The first line in this model is used for naming it. This model is called *List_Node* and it is under the “package” *util*. The next two lines are for importing other Alloy files to use them in this model. The model then defines a signature (i.e. a set of objects) called *List_Node* which maps to the class *List.Node*. There are two relations defined: *next* and *element*. The *next* relation has the type: $(List_Node, List_Node,$

State). It maps a *Node* to its successor as defined in one state (i.e. either the pre-state or the post-state). For example, let N1, N2 be *Nodes* and S1 be a *State*; if the tuple (N1, N2, S1) belongs to the *next* relation, it indicates that N2 is the successor of N1 in the *State* S1. Similarly, the *element* relation has the type (List_Node, Object, State). Notice that the *next* and *element* relations map to the fields of the class *List.Node*.

TestEra also produces an Alloy model for *List*:

```

module util/List

open testera/mutation/State
open util/List_Node

sig List {
  header: util/List_Node/List_Node ?-> testera/mutation/State/State
}

```

This model uses the *List_Node* model defined earlier. It defines a signature called *List* and a relation called *header*, which map to the class *List* and the field *header* respectively.

Once TestEra has built the models for each class, it proceeds to build the input and output models that it uses to generate and verify testcases respectively. The tool expects the pre- and post-condition, the signature of the method being tested, the representation invariant, the number of tests, and the bound on input/output size in the command line. From this information, TestEra builds the input and output models.

We want to test the method *remove* from our linked list example. If the pre-condition of the method is that there is at least one element in the list and we make the representation invariant that the list is acyclic, then the tools generates the following input model:

```

module util/List/remove_input

open testera/mutation/State
open java/lang/Object
open util/List_Node
open util/List

static part sig Pre extends testera/mutation/State/State {}

static sig Input {
  e: option java/lang/Object/Object,
}

```

```

    this_: util/List/List
  }

fun util/List/List::repOk() {
  some n: this_.header::Pre.*next::Pre | no n.next::Pre
}

fun inputConstraint() {
  Input::this_..repOk()
  #Input::this_.header::Pre.*next::Pre>1
}

run inputConstraint for 3 but 1 testera/mutation/State/State, 1 Input

```

This model defines a static signature called *Pre* that extends the signature *State*. This means that *Pre* is just a one-element set that has the type *State*. The model also defines two relations: *e* (which is the input to the method *remove*) and *this_* (which maps to the *this* variable in Java). The *::Pre* suffix in the *repOk* and *inputConstraint* functions tells the analyzer to evaluate any relation in the *Pre* state. This suffix is automatically generated by the tool. The user simply writes the *repOk* as (*some n:this.header.*next | no n.next*). Notice also that the pre-condition from figure 3-2 is in the method *inputConstraint*. The *inputConstraint* function is used to generate examples that fit the model above.

Finally, the tool generates an output model. If the post condition of the method is that *e* is no longer in the list, TestEra would generate the following output model:

```

module util/List/remove_output

open std/ord
open testera/mutation/State
open java/lang/Object
open util/List_Node
open java/primitive/boolean
open util/List

static part sig Pre, Post extends testera/mutation/State/State {}
fact { Post = OrdNext(Pre) }

static sig Input {
  e: option java/lang/Object/Object,
  this_: util/List/List,
  result_: java/primitive/boolean/boolean
}

fun util/List/List::repOk() {
  some n: this_.header::Post.*next::Post | no n.next::Post
}

fun methodOk() {
  Input::this_..repOk()
  Input::e !in Input::this_.header::Post.*next::Post.element::Post
}

```

```
run methodOk for 3 but 2 testera/mutation/State/State, 1 Input, 2 boolean
```

Notice that the function *methodOk* contains the representation-invariant from our linked list example. It also contains a call to the *repOk* function (i.e. the representation-invariant). After the Java method has run, the *methodOk* function verifies the testcases, which are translated back to Alloy instances.

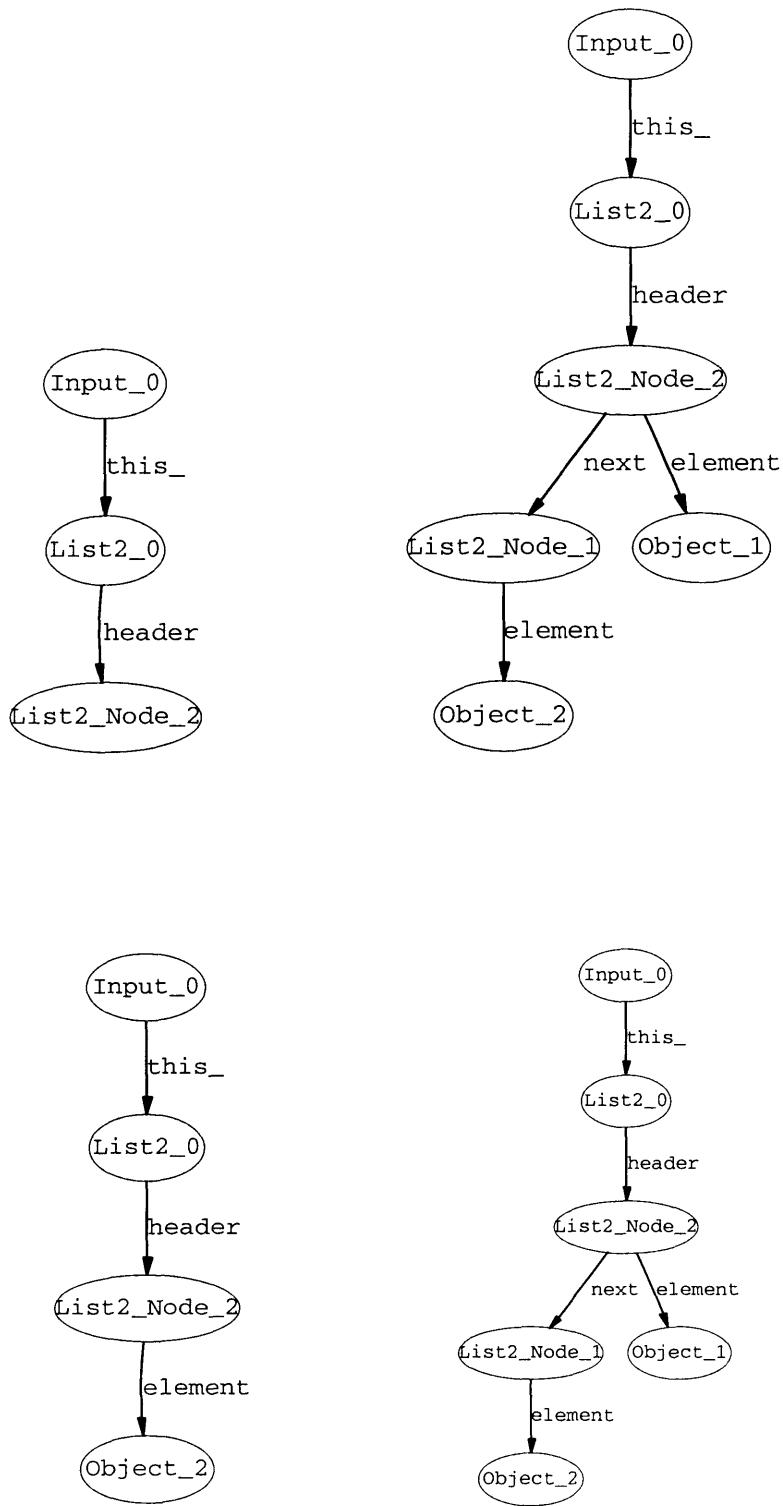


Figure 3-4: Instances Generated by TestEra

Chapter 4

MintEra: Combining Minshar and TestEra

In the last two chapters, we have explained in detail Minshar and TestEra. As we have also mentioned, MintEra depends on these tools. In this chapter, we describe how we combined Minshar and TestEra, what changes we made in TestEra and how MintEra is different from the original TestEra. We also describe additional features that exist in MintEra.

When we designed MintEra, we decided to use a Java parser to read the input Java file and extract meaningful annotations, as well as information about the code itself. Minshar's parsing utility was very primitive, and TestEra used a class parser to extract information about classes and fields. The Java parser presents a better way of getting information about the Java input and its annotations.

We have also made a number of changes to TestEra. MintEra uses TestEra as a testcase generation tool, but not as a verification tool. Thus, we have removed all the Alloy models that were used for verification, and removed state information from the automatically generated input model. We have also eliminated later stages of the tool (specifically the Java-to-Alloy and the output verification phases).

The last feature that was added to MintEra was the ability to visualize testcases. After the tool runs all testcases, the tool shows the user a visual image of the failed testcases to assist the user in tracking down the error.

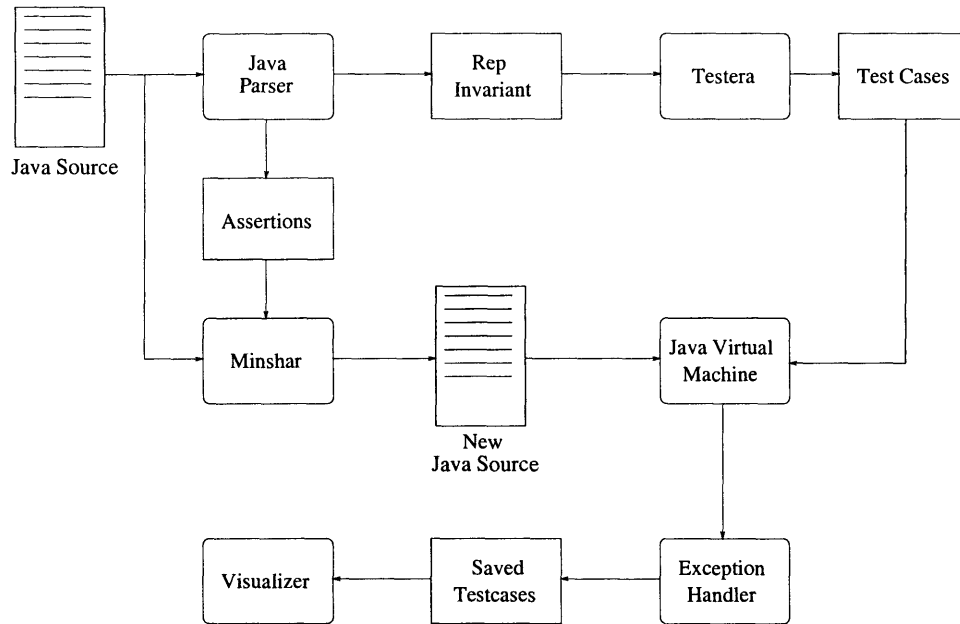


Figure 4-1: Stages of MintEra

4.1 Overview of MintEra

After explaining the various components of MintEra in the last few chapters, here we talk about combining these components together to form the MintEra tool. We will go through the stages of testing a Java file from parsing the Java code to visualizing the testcases. Figure 4-1 shows each of these stages.

The user invokes the MintEra tool by providing an annotated Java program and the signature of the method to test. The first task of the tool is to parse the Java program to pick up these annotations. There are two types of annotations: representation invariants and assertions. MintEra hands assertions to Minshar and the representation invariant to TestEra.

The Alloy assertions that MintEra gets from the Java parser are translated to Java code by Minshar (as prescribed by 2.4). The Java code is then inserted with the Alloy assertion in the Abstract Syntax Tree (AST) of the parsed Java file. After translating all the assertions, MintEra prints the AST into a new Java file that is used for testing. We back up the old Java file so that it replaces the modified Java file after MintEra finishes testing.

MintEra uses the representation-invariant to produce testcases. The tool does not depend on pre-conditions to generate testcases anymore, but rather utilizes the representation-invariant for that purpose. Using the representation-invariant instead of pre-conditions makes it easier for programmers to generate testcases. Users only need to specify one constraint in AAL, the representation-invariant, and would be able to generate testcases that are suited for that Java class¹. However, there is a case for allowing users to use pre-conditions to narrow down the scope of testcases. We are still considering the possibility of adding that feature.

TestEra still uses its bytecode parser to collect information about the classes and fields. After building the input models given the class names, fields and the representation invariant, TestEra then generates Alloy instances. These Alloy instances are represented as boolean arrays that could be saved in a file so that they could be used later.

When both TestEra and Minshar complete their work, the user compiles the newly generated Java class and then MintEra runs the method on the testcases. MintEra goes through the boolean arrays one by one and translates them (using the a2j translator) into Java concrete instances. The tool finally invokes the Java method on these testcases and catches any exception that is thrown. MintEra saves the failed testcases (represented as boolean arrays) and the exceptions thrown.

After MintEra runs all the testcases, the visualizer displays the failed ones. The visualizer shows a graphical form of the testcase as well as the exception that was thrown with that testcase. The visualizer aids the user in identifying the problem in their code. MintEra uses the visualizer from the Alloy Analyzer along with the saved boolean arrays to display these failed testcases.

In the next few sections, we describe the newly added (or changed) stages of MintEra.

¹Note that assertions could be written in Java. So essentially users can limit their Alloy specification to only the representation-invariant

4.2 Java Parsing

Since there is a need to extract information from the Java source code (such as class information and AAL annotations), we have opted to implement a parser for Java programs in MintEra. Minshar uses a scripting language to extract assertions and inserts Java code in its place. On the other hand, TestEra uses a bytecode parser to read class and field information from compiled `.class` files. TestEra also required the user to type in the representation invariant, the pre- and post-conditions, the method signature, and other information as arguments to the program. Entering all this information in the command line would be hectic. Thus, parsing the Java input file (that is annotated with pre/post conditions, rep-invariant, and assertions) provides a more durable and easier solution.

The Java grammar was written for a compiler compiler called SableCC [18]. We had modified a Java grammar –written by the Sablecc authors– to distinguish AAL annotations from normal Java comments. The original grammar simply discarded all comments, so we had to change the grammar to recognize AAL annotations, which begin with a `(/*@)` sign.

Given a grammar, the SableCC compiler generates a parser written in Java. The parser takes in an input Java file and generates a Abstract Syntax Tree (AST). SableCC also provides an AST visitor which crawls the tree and enables the programmer to collect information about the parsed Java file.

We implemented a visitor that extracts the information we needed from the Java file. The visitor first finds the representation invariant and keeps a record of it. The representation invariant has the following format:

```
/*@
   repok: <Alloy assertion>
*/
```

The visitor then crawls through the AST and finds AAL assertions that are embedded within the Java methods. These assertions have the following format:

```
/*@
```

```
    assert: <Alloy assertion>
*/
```

The parser picks up the Alloy assertion, and passes it to Minshar. After the assertion is translated into Java, the result code is appended to the comment. After all assertions have been translated, the AST is printed into a new Java file that contains the translated assertions. This file is the one used by TestEra for testing.

In this release of MintEra, we have decided not to get the class and field information from the Java parser and instead relied on the bytecode parser provided by TestEra. However, we hope to collect all class and variable information (including local variables within methods) from the parser in future MintEra releases.

4.3 TestEra Modifications

The main role of TestEra within MintEra is testcase generation. TestEra generates the input models from the Java specification and the representation invariant. After generating examples of these models in Alloy, these Alloy instances are converted into Java objects. MintEra does not use TestEra for verifying testcases.

MintEra's verification method is based on assertions that are rooted in the Java code. Subsequently, MintEra uses a different verification method than TestEra, which uses post-condition verification in Alloy to ensure correctness. Since MintEra still depends on TestEra for testcase generation, we have changed TestEra to remove any parts that did not deal with testcase generation. Specifically, we have made two major changes: we have eliminated any stages in the TestEra tool that deal with post-condition verification, and removed state information from generated models.

The first change made to TestEra is eliminating the Java-to-Alloy translation and the Alloy verification stages. Since assertion based verification is done in Java (i.e. we test correctness within the Java code), we do not need to translate the Java instance to Alloy anymore. So we can eliminate the j2a stage as well as running the output model in Alloy. We also do not need to generate an output model based on the post-condition.

Another modification we have made to TestEra is removing state information from the automatically generated Alloy models. In the original version of TestEra, there were two states where the Alloy Analyzer operated: the pre- and post-state. These are the states of the Java objects before and after running the method. However, we only run the AA in the pre-state (testcase generation) of MintEra. Since we have one state now, we do not need to annotate the relations with their state anymore. Thus, we have decided to eliminate all state information from our Alloy models, making the analysis simpler.

4.4 Visualizer

To aid the user in understanding the faults that the testcases have revealed, MintEra includes a visualizer that graphically shows the failed testcases. The nodes of the visualized graph are the objects of the testcase, and the edges represent the fields of these objects. Figure 4-2 shows an example of a visualized testcase of a linked list.

After the Alloy input models are analyzed by the Alloy Analyzer (section 3.2), it produces examples that are in the form of boolean arrays². MintEra uses these arrays in the Java-to-Alloy translation to produce concrete Java testcases. If the testcase fails when MintEra runs it on the Java method (i.e. if the method throws an exception that is caught by MintEra), the tool saves the boolean representation of the testcase so it can display it later. The tool thus accumulates all the failed testcases and saves the type of exception that was thrown when running them.

MintEra relies on the AA to visualize the failed testcases. The AA has a feature of displaying an Alloy instance given the Alloy model and the boolean representation. Our tool takes the boolean representation of the failed testcases, along with the analyzed model, and asks the Analyzer to display the testcase. The user can then cycle through the testcases, which are annotated with their exceptions. The AA visualizer allows the user to customize the graph and remove certain relations or provide a better view of the instance. For more information about the visualizer,

²These boolean arrays are the solution to the SAT formula representing the Alloy input model

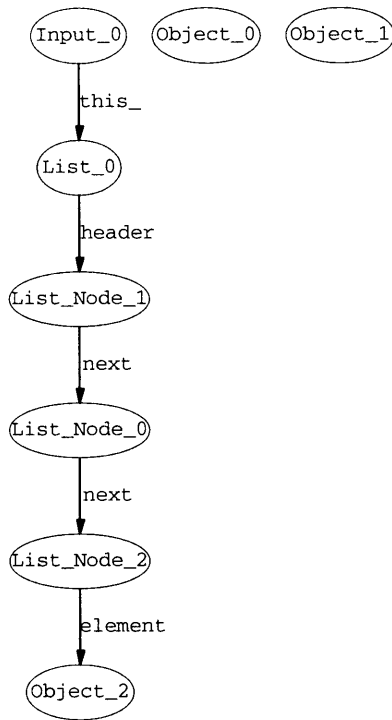


Figure 4-2: An Example of a Visualized Testcase

please check the Alloy website [14].

4.5 Comparing MintEra and TestEra

In our implementation of MintEra, we have changed the way we verify testcases from the original TestEra tool. While TestEra measures the output of the the method (state and returned value) against the post-condition, MintEra uses run-time assertions in code for verification. In this section we describe the key features of both tools, and we compare both methods of testing, discussing the pros and cons of each.

4.5.1 Testing

Of course, the most important feature that both tools provide is automatic testcase generation. Both TestEra and MintEra generate a testsuite from specifications that are written in a simple and effective language, alleviating the burden of manual test-

case generation. Both tools are adaptable to design changes, since they generate Alloy models based on field and class information (which are detected automatically). The testcases are also comprehensive given a certain bound on the input; TestEra and MintEra will generate all possible input cases up to a certain size.

Providing an adaptable, comprehensive, automatic testcase generator will help users test their programs in an easier fashion. These tools allow the programmer to concentrate on the design and implementation phases of the software cycle, while discovering hidden bugs in their code ³.

4.5.2 Documentation

An important feature that both tools provide is better documentation. The user has to write specifications to generate and verify testcases. TestEra requires the user to specify the pre- and post-conditions of methods, forcing the user to better document their method. On the other hand, MintEra requires the user to specify the representation invariant and embed their program with assertions, which document the code itself. From the user's perspective, writing specification now has an immediate reward; if the user writes well-documented code, then they do not have to worry about testing or extensive debugging. Thus, these tools encourage better software engineering practices.

4.5.3 Granular Testing

One essential difference between these two methods of testing is that MintEra allows a more granular method of verification. TestEra checks the correctness of a testcase by checking that testcase against the post-condition of a method. While checking the post-condition of the method is enough to check correctness, it does not allow the user to locate the cause of the error. The user now has to examine the code carefully to find the bug. This is especially hard when dealing with large methods.

³While using MintEra on a linked list implementation, we were able to discover inadequate handling of a null pointers that we were not aware of. MintEra generated testcases that threw a *NullPointerException* which we did not anticipate.

MintEra allows the user to insert assertions at any point in the code, which helps the user pinpoint the bug. By adding more assertions, the testcase fails in the code fragment around which the problem occurs and the the user can identify the location of the error with more ease. Notice that by adding more assertions, the user would be documenting their code better, which is an added advantage.

4.5.4 State Handling

Another essential difference between MintEra and TestEra is handling state. As explained in 3.1, TestEra introduced a state relation that differentiates between the pre- and post-states of the instances. MintEra eliminated this relation since it does not use the Alloy Analyzer for verification. We have made that choice because we wanted to rely on the inherent state machine that exists in Java instead of emulating it in Alloy. Depending on the inherent Java state has two main advantages: smaller models and ability to include more states.

By eliminating state, Alloy models are now much simpler to produce and analyze. Since Alloy depends on an SAT solver to produce examples, having a bigger model exponentially increases the time it takes to evaluate. Thus it would be desirable to have smaller models to avoid long running times. The user can also opt to increase the size of their input, having more testcases for the same amount of running time. For example, instead of generating linked lists that have at most 3 nodes, the user can now generate lists with a maximum of 4 or 5 nodes, adding more confidence in the correctness of the method.

Depending on the inherent Java state also allows the user to evaluate assertions in more than one state. TestEra allows the user to compare at most two states: the pre-state and the post-state. On the other hand, in MintEra the user can utilize more than two states, since each assertion could be viewed as a snapshot of the current state. Thus, referring to a variable in *any* previous assertion (by using special symbols) is referring to an earlier state. Since there is no limit on the number of assertions the user can define, the number of states is also not limited.

4.5.5 Locality of Error Detection

A more serious problem that might occur while using TestEra is testing nested calls to methods. When testing a certain method, a bug might be caused by a call to another method. TestEra will not be able to identify the method where the bug occurred. On the other hand, MintEra would fail either in the method being called or around the method call.

Consider the following scenario as an example. Assume there are two methods within a Java class –call them methods A and B– where method A calls method B. Naturally one would test method B for correctness before method A. Now imagine that method B has a bug that escapes the TestEra and MintEra testing for some reason (bad pre-condition, tests were incomprehensive, input too small...etc). The user is confident of method B’s operation and turns to the A method for testing. However, while running a certain testcase on method A, the bug in B manifests itself.

When using TestEra, the bug will not be detected until the post-condition of method A. Since the user is confident that method B is correct, he/she will try to debug the A method to find the problem, wasting a lot of time. On the other hand, MintEra will fail exactly where it is supposed to fail: in an assertion within method B. The user now is directed to look at method B instead of method A, which reduces the amount of debugging time.

4.5.6 Run-time Checks

An additional advantage of using assertions for verification is that they can be used in software releases. These assertions can be used as safety check when the program is actually used. The programmer can ensure that, during run-time, the state of the program is not faulty. Thus, the software can detect errors before they actually cause some serious harm to the program or –even worse– permanent data.

On the other hand, TestEra does not implement run-time checks. It is hard and computationally expensive to use TestEra’s method of verification (i.e. the Alloy Analyzer) for run-time checks. There is quite a bit of work to be done in order to

use the Alloy Analyzer (generating models, j2a translation...etc). Even if that work is done, it is costly –in terms of performance– to do so. MintEra allows the user to utilize the Alloy language for run-time checks while having a more acceptable running time.

Notice also that AAL assertions are essentially comments, so they are not noticeable by the compiler. However, the programmer can opt to “turn them on” by translating these Alloy assertions into Java. The user can also “turn them off” by removing the translated code (which is done automatically). Thus, the programmer has the option of using these assertions at run-time or not.

Chapter 5

Discussion and Conclusions

In the last few chapters, we have described in detail how MintEra was developed, and how it compares with TestEra. In this chapter, we evaluate the tool itself. We first observe the tool from the user's point of view. Then we look at ways to improve the usability of the tool by suggesting a number of features that we would like to add. Through this evaluation, we hope to understand where the tool stands currently, and where it would stand after implementing the features we suggest.

5.1 User Experience

Since MintEra is intended to be used by software engineers from various level of coding expertise, we need to measure how useful and effective the tool is. We have tested the tool ourselves and asked a number of students to try MintEra to help us evaluate it.

5.1.1 Personal Experience

We have used MintEra to test several Java programs, including a linked list and a red-black tree implementation. Our goal was to assess its usability and effectiveness in detecting bugs. While using the tool, we reached several conclusions about the tool (some of them were even unanticipated). Here, we describe some of these conclusions.

Finding Bugs in Software

The main goal of the tool is to detect bugs in software, and we were able to do just that. The tool has presented us with testcases, some of which had failed. Along the way of finding the source of these bugs, we developed some interesting techniques for debugging. One of these techniques was “commenting out” a part of the assertion to identify what section of the assertion failed with what testcases. Using the visualized testcase with the knowledge of the failed section of the assertion, we were able to find the bug in software that caused the error, correct it and test it again to make sure we have fixed it. We then proceeded to incrementally uncomment the assertion formula by formula until we managed to fix all bugs.

The technique we developed is similar to some debugging strategies that are currently used. Debuggers usually test one constraint at a time while printing the state of the program into standard output. MintEra allows users to utilize the same technique without cluttering the program with unnecessary debugging code.

Visualization was quite helpful in presenting the state of the testcase. The visualized testcase allowed us to trace the code on that instance and understand the cause of the error. However, as we explain later on, extra visualizing features would have helped us debug better.

Documentation Correction

In the previous chapter, we have mentioned that MintEra encouraged documentation of software by requiring the user to write specification. However, while using MintEra for testing, the tool provided another (unexpected) feature: specification checking. While testing a program, some of the testcases failed due to errors in specifications rather than errors in the code. For example, the representation invariant might not have been strong enough, making MintEra generate testcases that can not be produced by the code itself (e.g. red-black trees that cannot be generated by calling the *add* method repeatedly). By identifying that the problem is in specification, the user then can correct them, allowing the user to have better documentation.

The user could also have errors in assertions within the program. The fault could be caused by a check that should not exist. For example, while testing the *remove* method in our linked list implementation, we inserted an assertion checking that the element removed is not part of the list anymore. However, we discovered that this was a wrong check; the element could exist more than once within the linked list, and our remove method is supposed to only remove the first occurrence of the element.

At first, having testcases fail because of specification errors might seem like a problem in our approach of testing. However, this problem is analogous to including bad testcases in manual testcase generation, or anticipating wrong outputs from the method (like feeding negative numbers to a square root method, or checking that the square root of 9 should be 4). MintEra gives the user the opportunity to interactively correct their specification. In effect, the code is checked against specification, and specification is checked against code.

Bugs in Tool

While testing a number of our programs, a set of tests have failed because of bugs in the MintEra itself. Specifically, our translations of Alloy expressions were problematic. MintEra generated tests that managed to highlight these translation bugs. For example, while testing Java's *LinkedList* class, we found that we inserted *null* objects to various sets. Thus causing the condition *no o* when *o* is *null* to fail where it should not.

Of course, translated assertion should not include any bugs. However, the ability of detecting problems within MintEra by using the tool itself is a welcomed side-effect for the developers of the tool.

Additional Features

While using MintEra for testing, we have observed the lack of some extra features that would make the tool easier and more effective. We explain a few features that we personally thought were needed. We have a more detailed discussion of future work in section 5.2.

One observation we made while using MintEra is the frequent use of representation invariants as assertions. A common assertion at the end of every method is checking that the representation invariant still holds. Currently, we simply copied the text of the representation invariant into an assertion. However, a call to the representation invariant as a function call is a better alternative. In fact, defining functions and calling them in general is a feature that we would like to add to MintEra.

Another absent feature was post-state visualization. While we were trying to debug the errors we found when using the tool, we wanted to observe the testcase visually after running the method. A visualization of the testcase after an exception was thrown would help the user see why this testcase failed. A comparison between the pre- and post-state is definitely a desirable feature.

Highlighting the failed portion of an assertion is another important –yet absent– feature. When a testcase causes an error, it would be very helpful to the user to see what formula in the assertion has failed. The user can then observe the testcase visually and understand what the problem is. We have tried to emulate that by our method of “commenting out” parts of the assertion, as explained earlier. However, highlighting the formulas is an easy addition that would be very helpful for the user.

5.1.2 Novice Java Programmers

We have asked three MIT undergraduates, who had just completed their Software Engineering class, to use the MintEra tool. Through this evaluation, we can judge how easy to use MintEra is. The users ran our tool on Java programs that they wrote during the Software Engineering Laboratory.

Learning AAL

One important measure of the success of the MintEra tool is evaluating how easily can new users learn AAL and write constraints in it. We have explained the language and provided a manual similar to Appendix B. Within a short period of time, the users were able to write constraints –including complex ones– in AAL for their own

programs, indicating that AAL in fact is an effective language that is easy to learn and use.

Lack of Features

Users of MintEra complained about lack of features. A common problem among users was the absence of cross-referencing. Since users depend heavily on the Java Collections package, writing constraints involving *HashMaps* or *LinkedLists* is difficult. The problem arises because AAL does not support calling Java methods, and users cannot cross-reference specifications already written for these classes (like JML). Thus, it was difficult for users to write representation-invariants that involve any of the Java Collection classes.

The other problem that users faced was lack of support for arrays. Even though TestEra has support for arrays, MintEra has not ported this feature yet. Users who had arrays in their programs were not able to use them in writing representation-invariants. We intend to fix these problems in future MintEra releases.

Note that since assertions could be written in Java (as long as they throw exceptions when the assertion fails), users can utilize arrays and Java Collection classes with no problem. These issues are problematic only in representation invariants.

5.2 Future Work

Even though MintEra has some useful features, it still needs considerable amount of work to develop into be a standard tool. We have some ideas on what work is needed to help make the tool even more useful. In this section, we describe what extra features we would like to implement.

5.2.1 Minshar

In this section, we describe a number of additions and modification to Minshar that we would like to include. Notice that some of these features are relevant to other parts of MintEra.

Function Calls

There are some of Alloy formulas that are repeated throughout assertions. One example of such repeated assertions is checking the representation invariant after the method has completed. Currently, Minshar does not support functions, so the user has to copy these formulas from one assertion to another. The Alloy language has support for defining and calling functions. However, we need to a way to port Alloy functions to Minshar itself.

Implementing functions within Minshar is relatively straightforward. One way to implement support for Alloy functions is to translate them into Java methods. Since Minshar has the ability to translate any Alloy formula, then translating the body of the function is simple. The input/output variables of these functions are Set, boolean and integer variables.

The issue, however, is attempting to call Java methods within Alloy functions and vice versa. Since all expressions within Minshar are Sets, then using them as arguments (to other Java methods), or calling methods on them (i.e. dynamic dispatching) makes it difficult to call Java functions from within Alloy. The semantics of such method calling is not obvious. For example, what does it mean to call a method on a set of objects? Is it calling that method on *all* elements of that set? Do you include sets as arguments, and let the Java method deal with that set, or do you call every permutation possible? A study of such semantics is another area of future research.

On the other hand, calling translated Alloy functions from within Java methods is simpler. The arguments/output of the Alloy functions are only Set, boolean and integer variables that are well-defined. If we can constraint the calls from within Java methods to use variables that are of that type, then such an interface is possible.

Minshar Optimization

The current running time of Minshar is good enough to use within MintEra. However, if we intend to provide Minshar as a translation tool, then we need to optimize the

running time of the translated code. Here we provide a list of possible optimizations that could help in producing faster code.

The first such optimization is evaluating expressions in a lazy fashion. Instead of evaluating expressions sequentially, we could bundle more than one expressions navigation together to reduce running time. For example, the current method of evaluating the expression *this.header.*next.element* is to evaluate *this*, then *this.header*, then *this.header.*next* and finally *this.header.*next.element*. Instead of such slow evaluation we could evaluate the whole expression at once by running the reflexive closure method on *this.header* and following the field *next.element*.

Another optimizing that we are hoping to implement is to use Java inherent dereferencing mechanism instead of using the Reflection API. We currently use reflection for dereferencing because we do not have access to variable types. Since the Java program is being parsed to extract AAL annotations, we can collect type information about these variables that would allow us to use inherent Java dereferencing (through casting). This problem could also be solved by using Java's new Generics [13] feature, which is supposed to be released soon.

Supporting Java Arrays

A feature that was implemented in TestEra but was not ported into MintEra is supporting Java arrays. We have not implemented arrays in MintEra because integer variables are handled differently between Minshar and Java. As explained later, we hope to develop a unified model of integer variables, which would enable us to implement an array model based on this integer model. Notice that Alloy uses the symbols (`[]`) as a weaker dereferencing symbol. So it is possible to use these symbols exclusively for supporting arrays in MintEra.

Highlighting Failed Alloy Formula within Assertions

A useful feature that we would like to add to the MintEra tool is indicating to the user what portion of the assertion has failed. Since a lot of assertions are multi-formulas, where these formulas are *anded* together, we can detect which of these formulas have

evaluated to false. Separate formulas are separated by carriage returns (which are read as && symbols). Each formula evaluates to a boolean variable, and a sequential check of each formula identifies which one had evaluated to false. Thus, exceptions can be thrown with the text of the formula as the exception's message, indicating the failed section of the assertion.

Implementing State

As we have indicated in section 4.5.4, we have eliminated state from TestEra models citing Java inherent state machine as the reason behind this elimination. However, we have not supported referring to earlier states of Java variables in Minshar.

By appending one (or more) (' symbols to Java variables, the user can refer to these variables in the state they were one (or more) assertions back. The tool thus saves the variable value (from the old state) in another variable, and uses this saved variable in the current assertion. The only issue is detecting what variables need to be saved. Users could indicate these variable by including them in a *save* field within an assertion. Another approach would be for the tool to automatically detect what variables it needs to save by having two passes through the assertions.

5.2.2 TestEra

In this section, we describe future work that we hope to implement within the TestEra tool specifically. Note that some of these features also include changes to Minshar. We have included them here because we felt that most of the work will be done within the TestEra section of MintEra.

Migrating to Alloy 3

While developing MintEra, the developers of Alloy have designed a new version of the Alloy language. The new version of Alloy (named Alloy 3 [14]) has a different grammar which is not backward-compatible but includes a lot of new and improved features –such as atomization and subtypes [15]. Migrating the MintEra tool to Alloy

3 will enable us to utilize these new features.

A few changes will be done to MintEra. First, we need to change the automatically generated Alloy models to be Alloy3-compatible. We also need to modify the classes within MintEra that deals with the Alloy Analyzer directly. And finally, the grammar of MintEra has to be slightly modified to reflect the new Alloy 3 grammar.

Cross-Referencing

MintEra currently has no way of utilizing models generated earlier to specify fields that have the type of library classes. For example, if a certain field has a type of Java's *LinkedList*, there is no way of using the methods of that class in writing specification for the program at hand. JML uses the method's specifications from other classes to emulate behavior of that method. We hope to develop a mechanism for cross-referencing that is similar in principal to JML.

Handling Integers

A problem that both Minshar and TestEra face is handling integer variables. Currently, TestEra has a different mechanism of handling integer variables from Minshar simply because these two tools have been developed independently. Alloy 2 did not have any models for handling integers, and thus each tool has to develop its own mechanism of dealing with integers.

Both methods of handling integers have their drawbacks. While Minshar requires its users to cast all integer variables, TestEra has a separate model for integers and requires its users to call various functions when using these variables. However, we will use the model Alloy 3 proposes for handling integers as a starting point for designing an integer framework for MintEra.

5.2.3 Graphical User Interface

Currently, the only Graphical User Interface (GUI) that exists within MintEra is the testcase visualizer. However, we plan to extend the GUI features of MintEra to

include a number of extra features, which are described here.

Visualization of Testcases in Post-State

One important feature that was needed while using MintEra is visualizing the testcase after it has failed. The visualized testcase in the post-state allows the user to detect the problem in their code (or specification) in an easier manner.

Implementing such a feature is quite straightforward. TestEra had already a Java-to-Alloy translator that could translate Java instances to Alloy. After translating the instance to Alloy, we can now visualize the testcase by utilizing the visualization section of MintEra.

Eliminating Unreachable Objects

In the current version of MintEra, the Alloy instances could include Objects that are not reachable by starting from the *Input* variable in the input model. These objects serve no purpose since they are not part of the testcase (as defined by reachability from the *this_* variable) or the arguments. Subsequently, the visualized testcase tends to be cluttered with unnecessary information. Elimination of such objects makes the visualized testcase easier to read.

We have attempted to remove these objects, by constraining the input model. Our method depended on constraining the signatures to be exactly the objects that are reachable from the *Input* variable. However, that attempt failed.

The solution to this problem exists within Alloy 3. Because of the new subtype system [15] in Alloy 3, defining such a constraint is much easier. Once MintEra becomes Alloy 3-compatible, we can then produce visualized testcases that are simpler to view.

MintEra GUI

One goal of the developers of MintEra is producing a GUI for the tool that acts as a testing environment. The GUI would enable the user to view and edit the Java file being tested. It would also present the user with drop-down menus that would set

options (such as input size, method tested...etc) and run various commands. The user would also receive better error reporting whether it is compiling errors from MintEra or reporting erroneous testcases.

The Alloy Analyzer has a GUI that has all these features, but for Alloy models. Modifying the existing GUI to produce a specialized MintEra GUI should be relatively straightforward.

5.3 Conclusions

In this thesis, we have presented the MintEra testing framework. Through combining two tools, namely Minshar and TestEra, we have managed to create a comprehensive, adaptable, automatic testcase generator and verifier. While MintEra is easy to use as a tool, it needs some additional features that could make it appealing for software engineers.

Users of MintEra annotate their code with representation invariants and assertions. Representation invariants are used to generate testcases up to a certain bound (which is set by the user). The method being tested is then run on these testcases and Alloy assertions (which are translated to Java) check the correctness of the method, and throw exceptions otherwise. Testcases that cause exceptions to be thrown are collected and displayed to the user in a visualized form, which help the user identify errors within their code.

Besides testing, MintEra offers users a number of key features that are helpful for software engineers. AAL is a simple and effective language that could be used to write specification in a concise and easy-to-read manner. These annotations document the program being tested, helping enforce good software engineering practices. Through testing, specifications are measured against code and vice-versa. Thus, the user can test correctness of their code, as well as correctness of their specifications. Assertions within code provide granular testing that help the user pinpoint the location of errors.

MintEra could be extended in a number of ways to improve its usability. We have presented a list of additions that we would like to make to the tool. These changes

range from visualizing the testcases in the post-state to making the tool Alloy 3 compatible.

Our hope, by implementing MintEra, is that using such tools become a standard practice within the software engineering world. We believe that through the use of MintEra and other similar tools, users can rid their programs from bugs, helping them reach better quality of software.

Appendix A

Minshar Algorithm and Helper Functions

A.1 Minshar Algorithm

In This section we present the Minshar Translation algorithm. The algorithm takes every grammar production and translates it into a pair of Strings. The first String is the result variable of the translation (int variable for int expressions, Set for expr, and boolean for formulas.) The second String is the Java code that is the actual translation of the grammar production. Refer to figure 2-2 for AAL Grammar.

A.1.1 int expressions

```
Ti(number) = {
    v = freshvariable("int")
    return <v,"v = number;">
}

Ti(#expr) = {
    v = freshvariable("int")
    <ve,ce> = Te(expr)
    return <v, "ce v = ve.size();">
}

Ti( (intexpr) ) {
    return Ti(intexpr)
}

Ti( if formula then intexpr1 else intexpr2 ) {
    v = freshvariable("int")
```

```

    <vf,cf> = Tf(formula)
    <vi1, ci1> = Ti(intexpr1)
    <vi2, ci2> = Ti(intexpr2)
    return <v, "cf if (vf) { ci1 v=vi1;} else {ci2 v=vi2;}">
}

Ti(intexpr1 intop intexpr2) {
    v = freshvariable("int")
    <v1,c1> = Ti(intexpr1)
    <v2,c2> = Ti(intexpr2)
    return <v,"c1 c2 v=v1 intop v2;">
}

```

A.1.2 Formulas

```

Tf(expr1 [neg] : expr2) {
    v = freshvariable("boolean")
    <ve1,ce1> = Te(expr1)
    <ve2,ce2> = Te(expr2)
    if(neg)
        return <v,"ce1 ce2 v = (ve.size!=1)? false:!ce2.containsAll(ce1);">
    else
        return <v,"ce1 ce2 v = (ve.size!=1)? false:ce2.containsAll(ce1);">
}

```

```

Tf(expr1 [neg] = expr1) {
    v = freshvariable("boolean")
    <ve1,ce1> = Te(expr1)
    <ve2,ce2> = Te(expr2)
    if(neg)
        return <v,"ce1 ce2 v=!ve1.equals(ve2)">
    else
        return <v,"ce1 ce2 v=ve1.equals(ve2)">
}

```

```

Tf(expr1 [neg] in expr2) {
    v = freshvariable("boolean")
    <ve1,ce1> = Te(expr1)
    <ve2,ce2> = Te(expr2)
    if(neg)
        return <v,"ce1 ce2 v=!ve2.containsAll(ve1)">
    else
        return <v,"ce1 ce2 v=ve2.containsAll(ve1)">
}

```

```

Tf(intexpr1 intcompop intexpr1) {
    v = freshvariable("boolean")
    <vi1,ci1> = Te(intexpr1)
    <vi2,ci2> = Te(intexpr2)
    if(compop=="=")
        compop=="<="
    if(neg)
        return <v,"ci1 ci2 v!=(vi1 compop vi2);">
    else
        return <v,"ci1 ci2 v=vi1 compop vi2;">
}

```

```

Tf((formula)) {
    return Tf(formula)
}

```

```

Tf(neg formula) {
    v = freshvariable("boolean");
    <vf,cf> = Tf(formula)
}

```

```

    return <v,"cf v=!vf;">
}

Tf(formula1 && formula2)|(formula1 and formula 2) {
    v = freshvariable("boolean")
    <vf1, cf1> = Tf(formula1)
    <vf2, cf2> = Tf(formula2)
    return <v,"cf1 cf2 v=vf1&&vf2;">
}

Tf(formula1 || formula2)|(formula1 or formula 2) {
    v = freshvariable("boolean")
    <vf1, cf1> = Tf(formula1)
    <vf2, cf2> = Tf(formula2)
    return <v,"cf1 cf2 v=vf1||vf2;">
}

Tf(formula1 <=> formula2)|(formula1 iff formula 2) {
    v = freshvariable("boolean")
    <vf1, cf1> = Tf(formula1)
    <vf2, cf2> = Tf(formula2)
    return <v,"cf1 cf2 v=(vf1==vf2);">
}

Tf(formula1 thenop formula2) {
    v = freshvariable("boolean")
    <vf1, cf1> = Tf(formula1)
    <vf2, cf2> = Tf(formula2)
    return <v,"cf1 cf2 v=!vf1||vf2;">
}

Tf(formula1 thenop formula2 elseop formula3) {
    v = freshvariable("boolean")
    <vf1, cf1> = Tf(formula1)
    <vf2, cf2> = Tf(formula2)
    <vf3, cf3> = Tf(formula3)
    return <v,"cf1 cf2 cf3 v=(vf1 ? vf2 : vf3);">
}

Tf(formulaseq) {
    return Tfs(formulaseq)
}

Tf(all declplus formulabody) {
    <i,vf,l,c> = header(declplus,formulabody)
    va = freshvariable("boolean")
    s1 = "va=true; c if(!vf) {va=false; break 1;}"
    s2 = footer(i)
    return <va,"s1 s2">
}

Tf(no declplus formulabody) {
    <i,vf,l,c> = header(declplus,formulabody)
    va = freshvariable("boolean")
    s1 = "va=true; c if(vf) {va=false; break 1;}"
    s2 = footer(i)
    return <va,"s1 s2">
}

Tf(some declplus formulabody) {
    <i,vf,l,c> = header(declplus,formulabody)
    va = freshvariable("boolean")
    s1 = "va=false; c if(vf) {va=true; break 1;}"
    s2 = footer(i)
    return <va,"s1 s2">
}

Tf(sole declplus formulabody) {

```

```

    <i,vf,l,c> = header(declplus,formulabody)
    va = freshvariable("boolean")
    vb = freshvariable("boolean")
    s1 = "va=true; vb=false; c"
    s1 = "s1 if(vf) {
        if(vb) {va=false; break 1;}
        else vb=true;
    }"
    s2 = footer(i)
    return <va,"s1 s2">
}

Tf(one declplus formulabody) {
    <i,vf,l,c> = header(declplus,formulabody)
    va = freshvariable("boolean")
    s1 = "va=false; c"
    s1 = "s1 if(vf) {
        if(va) {va=false; break;}
        else va=true;
    }"
    s2 = footer(i)
    return <va,"s1 s2">
}

Tf(no expr) {
    v = freshvariable("boolean")
    <ve,ce> = Tf(expr)
    return <v, "ce v = ve.isEmpty()">
}

Tf(some expr) {
    v = freshvariable("boolean")
    <ve,ce> = Tf(expr)
    return <v, "ce v = !ve.isEmpty()">
}

Tf(sole expr) {
    v = freshvariable("boolean")
    <ve,ce> = Tf(expr)
    return <v, "ce v = (ve.size())<=1)">
}

Tf(one expr) {
    v = freshvariable("boolean")
    <ve,ce> = Tf(expr)
    return <v, "ce v = (ve.size())==1)">
}

Tf(two expr) {
    v = freshvariable("boolean")
    <ve,ce> = Tf(expr)
    return <v, "ce v = (ve.size())==2)">
}

Tfb(|formula) {
    return Tf(formula);
}

Tfb(formulaseq) {
    return Tfs(formulaseq)
}

Tfs({formulastar}) {
    v = freshvariable("boolean")
    if(formulastar.size()==0)
    return <v,"v=true;">
    s1=""
    s2 = " v ="

```

```

for(i=0;i<formulastar.size()-1;i++) {
  <vf,cf> = Tf(formulastar[i]);
  s1 = "s1 cf"
  s2 = "s2 vf &&"
}
<vf,cf> = Tf(formulastar[i]);
s1 = "s1 cf"
s2 = "s2 vf;"
return <v,"s1 s2">
}

```

A.1.3 Declarations

```

Tdp(declplus) {
  s = ""
  i = 0;
  while(declplus.hasNext()) {
    <v,c> = Td(declplus.next())
    s = "s c"
    i = i+v
  }
  return <i,s>
}
Td([qualifier] varplus: expr) {
  <ve,ce> = Te(expr)
  i = varplus.size()
  s = "ce"
  while(varplus.hasNext()) {
    v = freshvariable()
    e = freshvariable()
    s = "s Iterator e = ve.iterator();
      while(e.hasNext()) {"
    v = singleton(e.next());"
    a = varplus.next()
    VST.add(a,v)
  }
  return <i,s>
}

```

A.1.4 expressions

```

Te(var) {
  v = vst.get(var)
  if(v==null) {
    v = freshvariable("Set")
    return <v,"v = singleton(var);">
  }
  else {
    return <v, "">
  }
}

Te((expr)) {
  return Te(expr)
}

```

```

Te(expr1 [expr2]) {
    return Te (expr2.expr1)
}

Te( if formula then expr1 else expr2) {
    v = freshvariable("Set")
    <vf,cf> = Tf(formula)
    <ve1, ce1> = Ti(expr1)
    <ve2, ce2> = Ti(expr2)
    return <v, "cf if (vf) { ce1 v=ve1;} else {ce2 v=ve2;}">
}

Te(none [expr]) {
    v = freshvariable("Set")
    return <v,"v=new HashSet();">
}

Te(expr.varplus) {
    v = freshvariable("Set")
    <ve,ce> = Te(expr)
    s = "ce v = new HashSet();"
    for(i=0;i<varplus.length;i++) {
        s = "s v.addAll(dereference(ve,varplus[i]));"
    }
    return <v, s>
}

Te(expr.*(varplus)) {
    v = freshvariable("Set")
    <ve,ce> = Te(expr)

    s = "ce v = traverseStar(ve, new String[] {"

        //varplus is an array of strings
        for(i=0;i<varplus.length-1,i++)
            s = "s \"varplus[i]\", "
        s = "s \"varplus[i]\"]);"
    return <v, s>
}

Te(expr.^(varplus)) {
    v = freshvariable("Set")
    <ve,ce> = Te(expr)

    s = "ce v = traverseCaret(ve, new String[] {"

        //varplus is an array of strings
        for(i=0;i<varplus.length-1,i++)
            s = "s \"varplus[i]\", "
        s = "s \"varplus[i]\"]);"
    return <v, s>
}

Te(expr1+expr2) {
    <ve1,ce1> = Te(expr1)
    <ve2,ce2> = Te(expr2)
    return <ve1,"ce1 ce1 ve1.addAll(ve2);">
}

Te(expr1-expr2) {
    <ve1,ce1> = Te(expr1)
    <ve2,ce2> = Te(expr2)
    return <ve1,"ce1 ce1 ve1.removeAll(ve2);">
}

Te(expr1&expr2) {
    <ve1,ce1> = Te(expr1)
    <ve2,ce2> = Te(expr2)
}

```

```

    return <ve1,"ce1 ce1 v=ve1.retainAll(ve2);">
}

```

A.1.5 Helper Functions

```

header(decplus,formulabody) {
    l = freshlabel()
    temp = new VST();
    temp.parent = vst;
    vst = temp;
    <i,cd> = Tdp(decplus) //returns number of whites
    <vf,cf> = Tfb(formulabody)
    vst = temp.parent;
    return <i,l,vf,"cf l:cd">
}

footer(i) {
    for(j=0;j<i;j++) { //closing the parens
        s = "s }"
    }
}

```

A.2 Helper Java Functions

```

static Set singleton(Object o) {
    Set result = new HashSet();
    result.add(o);
    return result;
}

static Set traverseStar(Set s, String[] f) {
    Set result = new HashSet();
    LinkedList workList = new LinkedList(s);
    Set visited = new HashSet();
    while (!workList.isEmpty()) {
        Object o = workList.removeFirst();
        if (!visited.contains(o)) {
            result.add(o);
            visited.add(o);
            for (int i = 0; i < f.length; i++)
                optionallyAdd(workList, o, f[i]);
        }
    }
    return result;
}

static boolean optionallyAdd(LinkedList l, Object o, String n) {
    Object d = null;
    try {
        d = dereference(o, n);
        l.add(d);
        return true;
    } catch (Exception e) {
        return false;
    }
}

static Set traverseCaret(Set s, String[] f) {

```

```

Set result = new HashSet();
LinkedList workList = new LinkedList();
for (Iterator i = s.iterator(); i.hasNext(); ) {
    Object o = i.next();
    for (int j = 0; j < f.length; j++)
        optionallyAdd(workList, o, f[j]);
}
Set visited = new HashSet();
while (!workList.isEmpty()) {
    Object o = workList.removeFirst();
    if (!visited.contains(o)) {
        result.add(o);
        visited.add(o);
        for (int i = 0; i < f.length; i++)
            optionallyAdd(workList, o, f[i]);
    }
}
return result;
}

```

```

static Object dereference(Object o, String n)
throws Exception {
    Class c = o.getClass();
    Field f = c.getField(n);
    return f.get(o);
}

```

```

static Set dereference(Set s, String n) {
    Iterator i = s.iterator();
    Set res = new HashSet();
    while(i.hasNext()) {
        res.add(dereference(i.next(),n));
    }
    return res;
}

```


Appendix B

Alloy Language Specification

In this chapter, we explain in detail the Alloy language specification. MintEra uses a subset of Alloy as the language for writing assertions and representation invariants. Alloy is a powerful and accessible language. Thus, writing specifications in Alloy is easy and concise.

MintEra assertions and representation invariants are Alloy formulas, which are boolean expression in nature. While Alloy models have multi-arity relations, MintEra formulas consist only of one and two-arity relations. This restriction stems from the nature of the Java language. Representing Java objects and primitives only requires one-arity relation (i.e. a set of objects), whereas two-arity relations is sufficient to represent fields. In other words, a field is simply a relation between two objects.

Alloy has operators and quantifiers that allow the construction of more complex expressions, formulas, and integer expressions. By understanding how these operators work, the user can write Alloy formulas that express a condition on the code (either an assertion or representation invariant).

Expressions in Alloy can be divided into three types: boolean, set and integer expressions. The rest of this chapter is divided into 3 sections based on this type division. First we look at expressions that result in sets, then we look at boolean formulas, and last we look at integer expressions.

Op	Description	Example	Result
.	dereferencing	r.left	{r.left}
*	reflexive transive closure	r.*left	{r, r.left, r.left.left,...}
^	transitive closure	r.^left	{r.left, r.left.left,...}
+	set union (\cup)	r+r.left	{r, r.left}
+	field union	r.(left+right)	{r.left, r.right}
-	set difference (\setminus)	r.*left-r	{r.left,r.left.left,...}
&	set intersection (\cap)	r.*left&r	{r}
{}	set comprehension	{all x:r no r.right}	{r} if r has no right child
none[]	empty set(ϕ)	none[r]	{}

Table B.1: AAL’s Set Operators

B.1 Alloy Set Operators

In this section, we describe AAL set operators that are used in MintEra. They are summarized in table B.1.

B.1.1 Dereferencing (.)

Dereferencing is the essential operation in Alloy. The dereferencing operator in AAL carries the same semantics as in Java. To set up an example that will be used in the rest of this chapter, assume we have a Tree implementation that is given by the following Java code:

```
class Tree {
  Node r;

  class Node {
    Node right;
    Node left;
    Object element;
  }
}
```

Now, the expression *r.element* simply dereferences the root node with the “element” field.

Notice that dereferencing in AAL, unlike in Java, is set based. In a dereferencing operation, Alloy dereferences every element of the original set to construct a new set.

For example, the expression $(r+r.right).left$ (where $+$ is the union operator) results in the set $\{r.left, r.right.left\}$ ¹.

B.1.2 Union (+)

Alloy provides an operation to take the union of two sets. This carries the exact semantic meaning as the mathematical concept \cup . There are two ways of using the union operator: set union and field union.

Set union is the simple operation of combining two sets. The result of this operation is a new set that includes all the elements of both original sets. For example, the alloy expression $r.left+r$ results in the set: $\{r.left, r\}$.

On the other hand, the field union operator provides a more powerful way of expressing unions. The operator is used when dereferencing at the same time as taking a union. For example, given the above tree implementation, one can write: $r.left+r.right$ to express the set $\{r.left, r.right\}$. However, using the field union operator, one can write: $r.(left+right)$ to mean exactly the same expression². The effectiveness of field union will become more clear later on in this chapter when talking about reflexive and transitive closures.

B.1.3 Difference (-)

The difference operator in Java is also a set operator. It has the same meaning as the mathematical set difference. The result of this operation is another set that represents the difference between them. For example, the result of the alloy expression: $r.left-r.right$ is either the singleton set $\{r.left\}$ (if $r.left \neq r.right$) or the empty set (if $r.left=r.right$).

¹The Alloy language, in fact, provides a more powerful notion of dereferencing based on relations. This relational dereferencing is more general and includes the simple dereferencing mechanism that exists in Java. For more information about Alloy's dereferencing, please check [2]

²Note that in Alloy, the operators mentioned above are exactly the same, since they express relation union. The expression $left+right$ takes the union of the two relations. The expression $r.(left+right)$ is just dereferencing r with the relation $left+right$. The distinction has been made to simplify explanation of these operators in familiar Java terms

B.1.4 Intersection (&)

Another set operator that is implemented in Alloy is the intersection operator (\cap). The result of intersecting two sets is another set whose members belong to both original sets. For example, the alloy expression $r \& (r+r.left)$ produces the set $\{r\}$.

B.1.5 Reflexive Transitive Closure (*) and Transitive Closure (^)

One of the most useful operators in Alloy is transitive closure. This operator allows for implicit traversal of data structures in Alloy. It provides an effective way of traversing that helps alleviate some of the problems that users face when doing explicit traversal (like cyclicity and null pointer detection.)

The first such operator is the reflexive transitive closure (*), which is the process of continuously navigating a field in an object, until that field is null, or all objects have been navigated. For example, given the Tree model described above, take the Alloy expression $r.*left$. This expression is simply traversing the *left* field *zero* or more times, resulting in the set $\{r, r.left, r.left.left, \dots\}$.

The second operator is the transitive closure (^). It is similar to the reflexive transitive closure except that a field is traversed *one* or more times rather than zero or more times. For example, the alloy expression $r.^next$ results in the set $\{r.left, r.left.left, \dots\}$.

Combining any of these operators with the field union operator(+) provides a more effective way of traversal. For example, given the operators that we have mentioned, there is no way to construct the set of all nodes in our Tree example, since the expression $r.*left+r.*right$ only constructs the right-most and the left-most branches of the Tree. However, using the field union, we can construct the set of all the nodes in the tree by using the expression $r.*(left+right)$. The resulting set is $\{r, r.left, r.right, r.left.right, r.left.left, r.right.left, r.right.right, \dots\}$. Of course we can substitute transitive closure in place of reflexive transitive closure in the above example to produce the set $\{r.left, r.right, r.left.right, r.left.left, r.right.left, r.right.right, \dots\}$.

B.1.6 Set Comprehension (`{}`)

Set comprehension is another useful operations in Alloy. It allows the user to construct sets that are constrained by a certain formula. In our above `Tree` implementation, if we wanted to construct the set of elements that do not have a right child, we would simply write: `{all x:r.*(right+left) | no x.right}`. This statement can be read as: of all the elements x in the set $r.*(right+left)$, include only x 's that have no *right* element. (Please check section B.2.6 for more information about quantifiers such as *all*).

The power of the set comprehension operation stems from the fact that one can construct any arbitrary set based on a condition. This is done implicitly, without need for an explicit traversal. Of course, any arbitrary formula, or set of formulas, can be used to constraint the set.

B.1.7 if-then-else Condition for Expression

The if-then-else operation chooses between two sets based on a condition. The formula following the *if* is tested, and if it is true, then the expression following the *then* keyword is chosen, otherwise the expression following the *else* is chosen as the result of this expression. For example, consider the statement `if (r in r.*left) then r else r.*left`. Since the formula `r in r.*left` is true, the first expression (i.e. `r`) is chosen. If the formula instead was `r in r.^left` (which is false if the tree is acyclic), then the second expression `r.*left` would be chosen.

B.1.8 Empty Set (`none[]`)

Alloy provides an operation to create an empty set. By using the expression: `none[expr]`, where `expr` is any well-formed expression, you can get an empty set that has the same type as `expr`. For example, the expression `none[r]` gives an empty set of type `Node`.

Op	Description	Example	Result
:	checking subset(\subseteq)	$r.\hat{left}:r.*left$	true
in	checking containment (\in)	r in $r.*left$	true
=	checking set equality(=)	$r.*left=r+r.\hat{left}$	true
=	equals(=)	$1=2$	false
>	greater than(>)	$1>2$	false
>=	greater than or equal(\geq)	$1>=2$	false
<	less than(<)	$1<2$	true
=<	less than or equal(\leq)	$1=<2$	true
&&	and	$1=1&&2=1$	false
	or	$1=1 2=1$	true
<=>	equivalent(\iff)	$1=3<=>2=1$	true
=>,	if-then-else (\Rightarrow)	$1=3 \Rightarrow 1=1, 1=2$	false

Table B.2: AAL's Boolean Operators

B.2 Alloy Boolean Expressions

In this section, we discuss Alloy boolean operators. All Alloy assertions and representation invariants have to be boolean formulas (they cannot be set or integer expressions.) Please check table B.2 for a comprehensive list of these operations.

B.2.1 Set Equality (=)

The (=) operator tests for equality between two sets. Two sets are equal if they have exactly the same elements. For example, the following formula is true: $r.*left=r+r.\hat{left}$. However, the formula $r.*left=r.\hat{left}$ is false if the tree is acyclic.

Notice that you can test set *inequality* by putting the ! operator before the = operator. For example, the following formula: $r.*left!=r.\hat{left}$ is true if the sets $r.*left$ and $r.\hat{left}$ are unequal.

B.2.2 Subset Checking (:)

Alloy provides an operator that checks whether a set is a subset of another set or not, which is the mathematical operator (\subseteq). The resulting expression is a boolean formula. For example, the alloy expression $r.\hat{left}:r.*left$ would return true since

$r.*left$ includes all the elements in $r.^left$. However, the mirror expression $r.*left:r.^left$ would return false since $r.*left$ includes r but $r.^left$ doesn't (again, only if the tree is acyclic.)

Similar to set equality, you can add the $!$ operator before $:$ to indicate the negative of this operation. The following formula: $r.*left !: r.^left$, for example, asserts that the set $r.*left$ is not a subset of $r.^left$.

B.2.3 Element Containment (in)

This operator is Alloy's implementation of the mathematical element containment operator (\in). The Alloy operator takes two sets and checks that the left operand is contained in the right operand, with the condition that the left operand has to be a singleton (i.e. the set has one element only.) Note that this expression is also a boolean formula. For example, the Alloy expression $r \text{ in } r.*left$ would be true, but $r \text{ in } r.^left$ would be false.

By prepending the $!$ operator to the in operator, you can assert that an element is not contained within a set. For example, the formula: $r !\text{in } r.^left$ checks to make sure that $r.^left$ does not contain the element r .

B.2.4 Integer comparison

Alloy provides simple integer comparison operations that result in boolean formulas. The usual operator ($=, >, <, >=, =<$) are provided. For example, the formula $1=1$ is true. On the other hand, the formula $1>1$ is false.

Like in set comparison, one can use the $!$ operator before any of these integer operators to negate the formula. Thus, The operator $!<=$ reads "not less than or equal", and the operator $!=$ is "not equal" and so on.

B.2.5 Logical Operations (&&/||/<=>/=>,)

Alloy has support for logical operators that compare formulas. These operators carry the same meaning in Alloy as they do in Mathematical Logic. The $\&\&$ operator is the

logical “and” and the `||` operator is the simple logical “or”. The `<=>` operator is the equivalence operator (i.e. both formulas are true, or both are false). For example, the formula `(1=1 && 2>1)` is true. However, the formula `(r in r.*left <=> r in r.^left)` is false.

The last logical operator in Alloy is the then-else operator (`formula1=> formula2, formula3`). This operation works by looking at `formula1` and based on it, one of the next two formulas is chosen. If `formula1` is true, then the result of the whole operation is based on the result of the `formula2` (following the “=>” operator.) Otherwise, the result of the operation depends on the result of the `formula3` (following the comma “,”). For example, consider the formula: `(1=1 => 2<1, 2>1)`. Since the first formula `1=1`, is true, the result of the whole expression is the result of the formula `2<1`. Since this last formula is false, the whole expression becomes false.

Note that the else section of if-then-else is optional. A user can write: `formula1 => formula2`. If `formula1` is true, then the result of the whole formula depends on the result of `formula2`. However, if `formula1` is false, then the whole formula is false.

B.2.6 Quantified Formulas

Quantifiers are considered one of Alloy’s most useful expressions for two reasons. First, they allow implicit formula checking on all elements of a set without need for an explicit traversal. Second, these operations are intuitive to use since they rely on familiar notations such as \forall and \exists .

The first form of quantified formulas is (`quantifier var:set | formula`). In this form, `quantifier` is one of the following keywords: *all*, *some*, *no*, *sole*, *one*, and *two*. Please refer to table B.3 for an explanation of each of these quantifiers and the equivalent mathematical formula. The `var` in this form is the variable that spans the elements of the *set*. The `formula` is the condition being tested on each member of the set, usually involving the variable `var`.

We continue with an example of quantified expressions to make them clearer. Consider the formula `all x:r.*(left+right) | x.right = x.left`. This formula is equivalent to the question: in every node of the tree, are the left and right children of this node

Quantifier	Description	Example	Mathematical Expression
all	\forall	all $n:r.\hat{\text{next}} n!=r$	$\forall n \in \{r.\text{next}, \dots\}: n \neq r$
no	\nexists	no $n:r.\hat{\text{next}} n!=r$	$\nexists n \in \{r.\text{next}, \dots\}: n \neq r$
some	\exists	some $n:r.\hat{\text{next}} n!=r$	$\exists n \in \{r.\text{next}, \dots\}: n \neq r$
one	exists exactly once	one $n:r.\hat{\text{next}} n!=r$	$\exists_1 n \in \{r.\text{next}, \dots\}: n \neq r$
two	exists exactly twice	two $n:r.\hat{\text{next}} n!=r$	$\exists_2 n \in \{r.\text{next}, \dots\}: n \neq r$
sole	exists 0 or 1 times	sole $n:r.\hat{\text{next}} n!=r$	

Table B.3: AAL's Quantifiers

equivalent? If there is one node in the tree that has a left child that is different from the right one, then this whole formula is false. On the other hand, the formula *some $x:r.*(\text{left}+\text{right}) \mid x.\text{right} = x.\text{left}$* asks the question: are there any nodes that have a right child that is equivalent to the left child? If there is such a node, then this formula is true. However, if all nodes of the tree have distinctive right and left children, then this formula is false.

Another type of quantified formulas is directly related to the cardinality of sets. It has the form: *quantifier set*, where the *quantifiers* are the same as the other form. For example, consider the formula: *no $r.*(\text{left}+\text{right})$* . This formula asks the question: is the tree empty? That is: is it true that there are no elements in the set *$r.*(\text{left}+\text{right})$* ? Similarly, the formula *one $r.*(\text{left}+\text{right})$* ensures that there is exactly one node in the tree.

B.3 Alloy Integer Expressions

There is a limited support for integers in the current MintEra release, but further work is expected to be done to have better integer operations. Here, we describe the integer operators that currently exist in Alloy.

B.3.1 Element Count (#)

The element count operator simply finds the number of elements in a given set. Take for example the expression *$\#r$* in the above tree implementation. Since there is only

one root element, the result of that expression is 1. On the other hand, the expression $\#(r.*(left+right))$ returns the total number of nodes in the tree.

B.3.2 integer arithmetic(+/-)

Alloy provides integer arithmetic operations to manipulate integers. Addition and subtraction are the only operations provided. The output of this operation is an integer which is the result of the addition or subtraction: $1+1$ is 2, $\#r-1$ is 0, and so on.

B.3.3 Java int Variables

Since Alloy treats all variables as expressions (i.e. sets), there is a need to distinguish integer variables that exist in Java. If the user is writing *assertions* in Alloy, then they should include a casting operator (`int`) before using that variable. MintEra will treat that variable as an integer. For example, the statement $(int\ x) > 1$ tests whether the integer variable x is greater than 1. If the statement was written as $x > 1$ it would result in an error. The user can use normal numbers (such as 1,2..etc) with such variables.

If the user is writing representation invariants using integer variables, they will be considered as signatures in Alloy. Thus, there is no need to cast them with an (`int`) operator. However, when trying to compare an integer variable and a normal number (such as 1,2,..etc), then the user should use the function *equals* to convert normal numbers to a type similar to the Java variable. The user should also always use the functions LE, GE, LT, GT for $=, >=, <, >$. For example, to check if the Java variable x is greater than one, we write $GT(x, equals(1))$ ³.

³The reason behind this discrepancy between assertions and representation invariants is that the Minshar and TestEra tools have been implemented separately. Each tool has used a different mechanism for dealing with integer variables that better suits its purposes. We hope to alleviate this discrepancy by having a better model for integers in MintEra

B.3.4 if-then-else Condition for integers

Similar to the if-then-else expression for expressions, the if-then-else condition for integers allows choosing an integer expression based on a condition. For example, in the expression *if (r in r.*left) then #r else 2*, the formula *r in r.*left* is true, so the result of the whole expression is the integer *#r*, or just 1.

Appendix C

MintEra Example

We provide a complete example of how MintEra generates and verifies testcases from an annotated Java file –a complete linked list implementation. Notice the representation invariant and assertions embedded within the following program:

```
import java.util.*;
public class List {
    public Node header = new Node();
    public Node end = header;
    int size = 1;
    /*@
     repok: some m: this.header.*next| no m.next
     no this.end.next
     some this.end
     some this.header
     this.end in this.header.*next
    */
    public static class Node {
        public Node next;
        public Object element;
    }
    public boolean add (Object o) {
        /*@
         assert: no this.end.next
        */
        Node n = new Node();
        end.next = n;
        n.next = null;
        n.element = o;
        end = n;
        /*@
         assert: some m: this.header.*next| no m.next
         some o => o in this.header.*next.element
         no this.end.next
         this.end in this.header.*next
        */
        size++;
        return true;
    }
}
```

```

public boolean remove (Object o) {
    Node p, n = header;
    p=n;
    /*@
    assert: some m: this.header.*next |no m.next
    */
    for(; n!=null; n = n.next){
        if(n.element==null&& o==null)
            break;
        if(n.element.equals(o))
            break;
        p = n;
    }
    if(n==null) {
        /*@
        assert: some o=> o !in this.header.*next.element
        some m: this.header.*next| no m.next
        no this.end.next
        some this.header
        this.end in this.header.*next
        */
        return false;
    }
    p.next = n.next;
    if(end==n)
        end = p;
    /*@
    assert: some m: this.header.*next| no m.next
    no this.end.next
    some this.header
    this.end in this.header.*next
    */
    return true;
}

public boolean contains(Object o) {
    /*@
    assert: some m: this.header.*next | no m.next
    */

    Node n;
    for (n = header; n!=null; n = n.next) {
        if(n.element == o) {
            /*@
            assert: o in this.header.*next.element
            some m: this.header.*next| no m.next
            no this.end.next
            some this.header
            this.end in this.header.*next
            */
            return true;
        }
    }

    /*@
    assert: o !in this.header.*next.element
    some m: this.header.*next| no m.next
    no this.end.next
    some this.header
    this.end in this.header.*next
    */
    return false;
}

public Object get(int index) {
    if(index>size-1)
        return null;
    /*@

```

```

    assert: some m: this.header.*next | no m.next
    (int index) < #(header.*next)+1
  */
  Node n;
  int i;
  for (n = header,i=1; n!=null&& i<index; n = n.next,i++);
  /*@
    assert: some n
    some m: this.header.*next| no m.next
    no this.end.next
    some this.header
    this.end in this.header.*next
  */
  return n.element;
}
}

```

To run MintEra, the user provides the method to be tested, as well as the location of the above Java file. Assume the user wants to test the *remove* method. Note that MintEra translates all the assertions in the file, since *remove* might call another method where an error might occur.

From the Java class and the representation-invariant, the tool generates three Alloy files. The first model is the *List_Node* one representing the *List.Node* class:

```

module List_Node

open java/lang/Object

sig List_Node {
  next: option List_Node/List_Node,
  element: option java/lang/Object/Object
}

```

MintEra also generates the *List* model representing the *List* class:

```

module List

open java/primitive/integer
open List_Node

sig List {
  header: option List_Node/List_Node,
  end: option List_Node/List_Node,
  size: option java/primitive/integer/integer
}

```

Finally, the tool generates the *remove_input* model, used to generate testcases:

```

module List/remove_input

open java/lang/Object
open List

```

```

open java/primitive/integer
open List_Node
open java/primitive/boolean

static sig Input {
  o: option java/lang/Object/Object,
  this_: List/List
}

fun List/List::repOk() {
  some m: this.header.*next| no m.next
  no this.end.next
  some this.end
  some this.header
  this.end in this.header.*next
}

fun inputConstraint() {
  Input::this_..repOk()
}

run inputConstraint for 3 but 1 Input, 2 boolean

```

Notice that the state information no longer exists in the input model. These Alloy files are hidden from the user, who does not have to worry about understanding them. However, MintEra uses these files to generate examples in the form of binary arrays. It then stores them in a file called *List-remove.sol*. The tool then loops through the testcases one by one, converting them into Java concrete instances. MintEra runs the *remove* method on each one of these testcases and catches any thrown exceptions. It stores these exception, along with the binary array representing the testcase, for visualization purposes. AAL assertions throw an exception called *MinsharFailedConstraintException*. Seeing such an exception means that an assertion has been violated.

While running the above example, the tool has passed 71 testcases, and failed 29 (out of 100). The tool visualizes the testcases that have failed and informs the user of the exception message. Figure C-1 shows some of the visualized testcases that the *remove* method had not passed. These testcases failed because of a *NullPointerException* that is thrown within the method *remove*, in the line:

```
if(n.element.equals(o))
```

After careful inspection of the *remove* method, along with the visualized testcases, we notice that in the code section:


```
if(n.element==null&& o==null)
    break;
if(n.element.equals(o))
```

the user checks *n.element* being *null* in the previous line. However, it is checked with another condition: that *o* is not *null*. Looking at the testcases, we notice that all of them include a non-null *o*, but at least one element in the list is *null*. We can fix the problem by changing the above segment to:

```
if(n.element==null && o==null)
    break;
if(n.element!=null && n.element.equals(o))
```

After running MintEra again, the method passes all testcases, suggesting that we might have fixed the problem. Having a bigger scope, or different testcases could surface the same problem, or perhaps discover other problems.

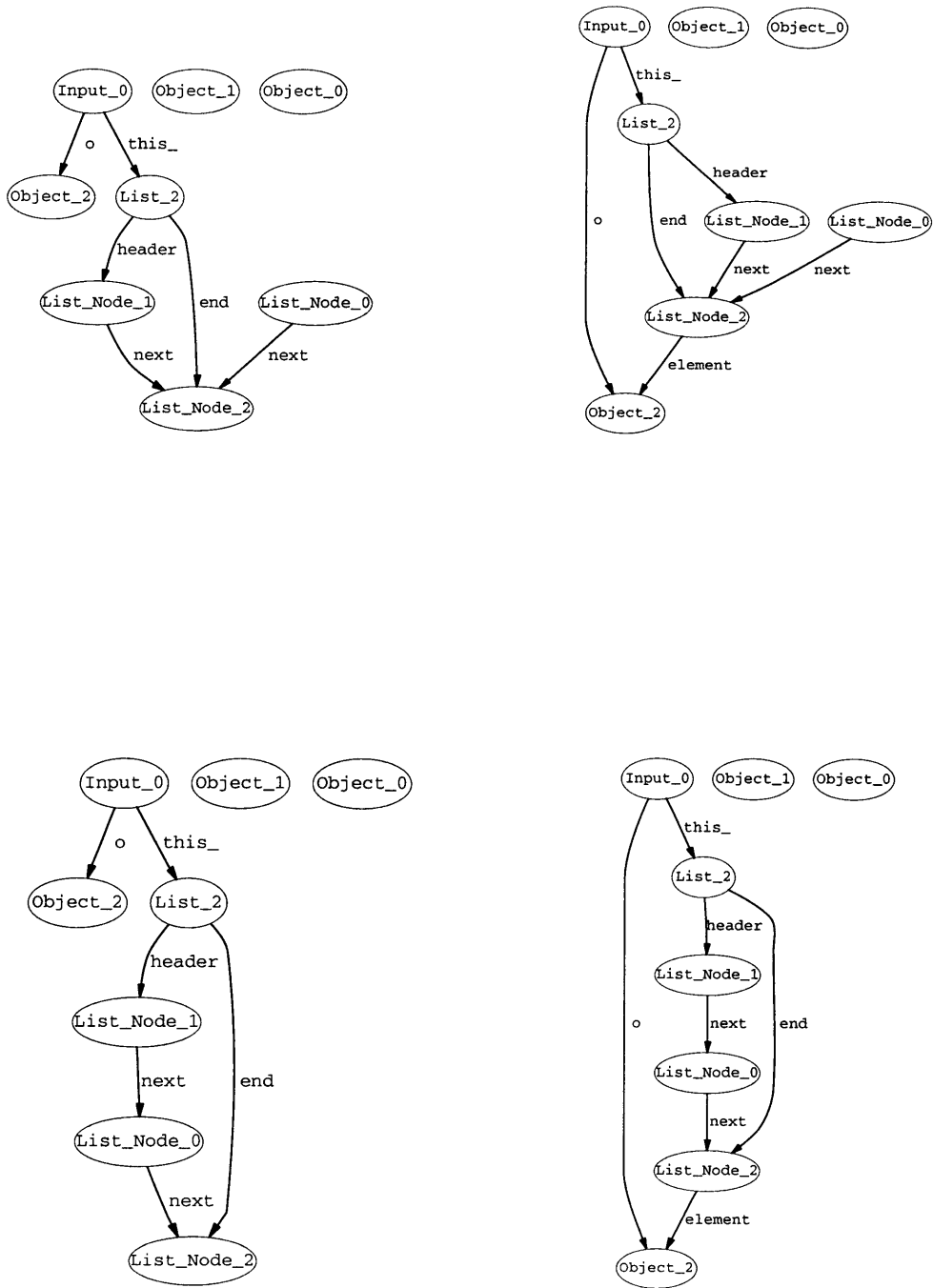


Figure C-1: Failed Testcases

Bibliography

- [1] Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, 226 Atanasoff Hall, Ames, Iowa 50011, 2000.
- [2] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Austria, September 2001.
- [3] Daniel Jackson. Automating first-order relational logic. In *Foundations of Software Engineering*, pages 130–139, San Diego, CA, November 2000.
- [4] S. Khurshid, D. Marinov, and D. Jackson. An analyzable annotation language. In *17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, Seattle, WA, November 2002.
- [5] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.
- [6] S. Khurshid. Generating structurally complex tests from declarative constraints. *Ph.D. Thesis*, Department of Electrical Engineering and Computer Science, MIT. February 2004.
- [7] Bertrand Meyer. Eiffel: The Language. *Prentice Hall*, 1992.
- [8] D.S. Rosenblum. A practical approach to programming with assertions. In *IEEE Transactions on software Engineering*, volume 21(1), pages 19–31, January 1995.

- [9] J. Spivey. An introduction to Z and formal specifications. In *Software Engineering Journal*, January 1989.
- [10] J. Warmer and A. Kleppe. The Object Constraint Language: Precise Modeling with UML. *Addison-Wesley*, 1999.
- [11] Cliff B. Jones. Systematic Software Development Using VDM. *International Series in Computer Science*. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
- [12] Warmer, J., Ed. Response to the UML 2.0 OCL RfP (ad/2000-09-03). *Object Management Group (OMG)*, Jan. 2003. Revised submission, version 1.6. OMG Document ad/2003-01-07.
- [13] Qusay H. Mahmoud. Programming with the New Language Features in J2SE 1.5. <http://java.sun.com/developer/technicalArticles/releases/j2se15langfeat/>, June 2004.
- [14] MIT Software Design Group. The Alloy Analyzer. <http://alloy.mit.edu>.
- [15] J. Edwards, D. Jackson, E. Torlak, and V. Yeung. Subtypes for Constraint Decomposition. In International Symposium on Software Testing and Analysis, Boston, MA, July 2004.
- [16] D. Marinov and S. Khurshid. MulSaw Project on Software Reliability. mul-saw.lcs.mit.edu
- [17] B. Al-Naffouri. An Algorithm for Automatic Test Generation of Run-time Assertions from Alloy Specifications. *Advanced Undergraduate Project*, Department of Electrical Engineering and Computer Science, MIT. June 2002.
- [18] E. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS USA 98 (Technology of Object-Oriented Languages and Systems)*, IEEE, 1998.