Systematic Testing Using Test Summaries: Effective and Efficient Testing of Relational Applications

Systematic Testing Using Test Summaries: Effective and Efficient Testing of Relational Applications

by

Shadi Abdul Khalek, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2011

Dedicated to my fiancee and family.

Acknowledgments

I cannot thank my family enough for the support, love, and encouragement they gave me throughout my studies. The fact that they will be absolutely as happy as I am with any of my accomplishments makes me desire it more. My father calling me "Dr. Shadi", even through my master's years, kept me going till the end. My mother's encouragement asking me not to procrastinate since my first year till the last minute of writing this acknowledgment has always been priceless, and my siblings looking up for me made me work harder to prove them right. I am so grateful to have them around me in my life.

My heartfelt gratitude goes to my fiancee Rana. She has been my inspiration to graduate and fulfill further dreams in my life. She kept me focused all the time. The last two years of my PhD. studies wouldn't have been the same without her in my life. Thank you for being patient all these years!

It has been an incredible journey here at UT Austin. I am so lucky to have been picked by Prof. Sarfraz Khurshid to join his team. I thank him so much for his continuous support, great guidance, and everlasting knowledge and advice he shared. Having him as my advisor was an honor. I wish him all the success and happiness in his life and career.

I would like to thank my committee members, Prof. Adnan Aziz, Prof. Allen E. Emerson, Prof. Don Batory, and Prof. Dwayne Perry, for their great advice, feedback, and support for my work. It is a privilege having them on my committee. I thank my professors at the American Univerisity of Beirut, Professors Elie Kayrouz, Paul Attieh, Fatima Abu Salem, Haidar Safa, and Wassim Masri.

I thank Dr. Fadi Zaraket for his warm welcome at Austin, he made me feel home since my first day.

I thank Dr. Bassem EL-Karablieh for the jump start he helped me with in my research track. I am very thankful for his good advice and the research knowledge and experience he shared with me. I am grateful to have him as a colleague and good friend.

Special thanks to all my colleagues in the Software Verification Validation and Testing group. It was a pleasure to working with smart and fun researchers: Dr. Engin Uzuncaova, Dr. Danhua Shao, Zubair Malik, Junaid Haroun, Guowei Yang, Divya Gopinath, Razieh Nokhbeh, and Lingming Zhang. I wish them the best of luck and success in achieving their goals and getting their degrees. In addition, I thank the Students in Software Engineering for uniting software engineering students together in a friendly environment.

On top of all, my experience at UT Austin could not have been as great without the good friends I had. First, I would like to thank the Fakhreddine's:

Mohamed (Moh), Ali, Suraya and Faris, Sarah, and Mohammad. I thank them for making me part of their family. I thank Yehia Zayour, Hala Nasser, Michele Saad, Salam Akoum, Omar El-Ayache, Marcel Nassar, Amin Abdel Khalek, and all of the Lebanese Social Club members. I thank my friends at Google, Andrew Chen, Ziad Hatahet, Evan Golschmidt, Camilo Arango, and Yongrim Rhee. You all made my journey cheerful.

I also thank my friends in Lebanon, all of whom kept in touch with me despite the distance. I thank Ali Abdel Khalek, Ragheed Abdel Khalek, Rabih Abdel Khalek, Fares Samara, Abdalla El-Horr, Amine Chehab, and Alfaisal Jauhary.

A special thank you to Melanie Gulik for helping me with any problem I had as a Student. She has always been there for all the graduate students and treated our problems as hers.

The work presented in this dissertation was funded in part by the National Science Foundation (awards CCF-0845628, IIS-0438967, and CCF-0702680) and the Air Force Office of Scientific Research (award FA9550-09-1-0351).

Systematic Testing Using Test Summaries: Effective and Efficient Testing of Relational Applications

Publication No. ____________________

Shadi Abdul Khalek, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Sarfraz Khurshid

This dissertation presents a novel methodology based on *test summaries*, which characterize desired tests as constraints written in a *mixed* imperative and declarative notation, for automated systematic testing of relational applications, such as relational database engines. The methodology has at its basis two novel techniques for effective and efficient testing: (1) *mixed-constraint solving*, which provides systematic generation of inputs characterized by mixed-constraints using translations among different data domains; and (2) *clustered test execution*, which optimizes execution of test suites by leveraging similarities in execution traces of different tests using *abstract-level undo operations*, which allow common segments of partial traces to be executed only once and the execution results to be shared across those tests.

A prototype embodiment of the methodology enables a novel approach for systematic testing of commonly used database engines, where test summaries describe (1) input SQL queries, (2) input database tables, and (3) expected output of query execution. An experimental evaluation using the prototype demonstrates its efficacy in systematic testing of relational applications, including Oracle 11g, and finding bugs in them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview and Problem Description

Reliability of complex software systems is a crucial goal to increase software dependability. The cost of failures in systems and data lost can be very expensive [40, 101]. Current methodologies depend heavily on testing to validate systems. Conceptually, testing is simple: run a program against a set of test inputs and check the correctness of the outputs. In practice however, testing remains expensive and largely manual. Many existing tools, e.g. JUnit [68], support the automation of test execution and error reporting. However, the crucial part of the test generation is least automated and involves manual effort.

Automating test generation for complex relational applications, such as database management systems (DBMS) is a challenging task. Automatic DBMS testing includes generating (1) test queries for a given database schema, (2) a set of test databases, and (3) oracles to verify the result of executing the queries on the input databases using the DBMS. Some approaches exist to automate the first two artifacts of the tests [90, 95]. SQL test queries can be generated using grammar based generators [44] and test data can be generated using query aware test input generators [16, 17]. Database generation approaches, however, do not provide mechanisms to verify the result of executing test queries on the generated databases. This process is usually performed either manually or by differential testing, i.e. comparing the execution results of a previous version of the DBMS with the current version to verify that any changes did not break a working component.

In this dissertation, we present a framework for automated black-box testing of database engines using *test-summaries*, abstract properties of desired tests, leveraging state-of-the art constraint solvers. We automate the generation of all three main artifacts, and show the effectiveness of using test-summaries in finding bugs in widely used commercial and open source database systems [1, 2, 3].

Our approach is based on writing summaries as logical constraints, each test summary consists of: input constraints that define desired test inputs and output constraints that characterize the test oracles (correctness checks) and defines the relations between inputs and outputs. We use tools that analyze those constraints and perform exhaustive bounded testing against all inputs from a bounded input space [1, 71]. This systematic, intent-driven approach allows developers to focus on the creative aspect of designing tests and test strategies while the tools perform the tedious tasks such as creation, execution, and verification of output of concrete tests.

Writing constraints and modeling systems using different modeling languages is receiving increased attention from developers and testers. They already write some constraints as assertions in their code and use them routinely for runtime checking. However, requiring a different language for writing constraints makes the learning burden high for developers. At the same time, some languages and constraint solvers are more effective than others.

In this dissertation, we present a new approach, and techniques that embody it, to increase the usability, efficiency, and effectiveness of systematic constraint-based testing and apply it to the domain of relational applications.

1.2 Overview of Our Approach

Our approach has three core research thrusts:

***Usability*: Writing Mixed-Notation Constraints.** We present a novel technique for mixed declarative and imperative formulation of structural constraints. In specific, we introduce a new notation to support a mixed style for writing specifications that represent *input constraints*, which describe desired test inputs, using a combination of declarative and imperative programming styles. Specifically, the user formulates constraints using a combination of the expressive Alloy language [65] — a declarative, first-order logic based on sets and relations — and the widely used imperative Java programming language. This approach enables the user to write and mix constraints freely using the two different paradigms. Alloy, with its support for path (navigation) expressions using transitive closure, allows succinct formulations of complex structural properties, such as those of class invariants in object-oriented code. Java, with its wide use, provides a familiar notation that is likely to pose a minimal learning burden.

***Effectiveness*: Combining Constraint Solvers.** We present new techniques for solving input constraints using a combination of solvers that support different classes of input constraints written using different programming paradigms. Specifically, we leverage the Alloy Analyzer [63] — a fully automatic SAT-based tool-set for checking Alloy formulas — and the Java PathFinder (JPF) [108] — an explicit state, open-source software model checker for Java — and use them in synergy for solving structural constraints and enable automated test generation.

***Efficiency*: Clustered Test Execution.** We present a novel technique for efficiently executing suites of unit tests, where several tests in a suite may contain a common initial segment of execution – a property often exhibited by systematically generated suites, e.g., those for bounded exhaustive testing. Our insight is that we can *cluster* execution of such tests by defining *abstract-level undo* operations [41, 91], which allow a common execution segment to be performed once, and its result to be shared across the tests, which then perform the rest of their operations.

1.3 Contributions

This dissertation makes the following contributions [1, 2, 3, 4, 5, 6]:

1.3.1 Mixed-notation for writing Constraints [5]

We present JABAL, a framework for mixed declarative and imperative formulation of structural constraints. Our approach enables the user to leverage the widely used Java language and the expressive Alloy language to model constraints. Thus, it does not require the user to follow one specific programming paradigm, rather it enables them to write a constraint using a paradigm they find more suitable for the specific constraint and to mix it freely with other constraints that may be written using a different paradigm. We introduce a new notation for writing mixed constraints; specifically we introduce the semi-colon operator which enforces an explicit dependency among constraints, akin to sequencing of statements in an imperative language.

1.3.2 Solving Mixed Constraints for Test Input Generation [5]

We enable JABAL to leverage different constraint solvers, namely the Alloy Analyzer and JPF, using data translations based on a relational model of the program heap. Abstraction translations translate Java data structures into Alloy instances and concretization translations translate Alloy instances to concrete Java data structures. We introduce *def-use* annotations, which state the sets of fields defined and used by a constraint, to combine the different solvers and guide solving of constraints.

1.3.3 Constraint-based SQL Query Generation [2]

We present a new technique that leverages the Alloy tool-set to model the language constraints of a useful subset of SQL and provides automated generation of syntactically and semantically valid SQL queries (with respect to the constraints). Queries generated using our technique are used as inputs for systematic relational database engine testing.

1.3.4 Systematic Testing of Relational Database Engines [1, 3]

We present a technique for systematic testing of relational databases. We present a novel use of test summaries to automate systematic testing of relational database management systems. Our framework uses test summaries to automatically generate: (1) SQL test queries with respect to a database schema, (2) a set of test databases (data populating tables), and (3) oracles to verify the result of executing the queries on the input databases using the test database management system.

1.3.5 Clustered Test Execution [4]

We present a technique for clustered execution of unit tests for Java programs. Given a set of tests, our framework analyzes the tests to build a *cluster tree* – a trie [99] structure that represents the sharing of initial code segments across the tests. Each node represents a code segment in a test. Thus, each path in the cluster tree represents a test case. A depth-first traversal of the cluster-tree forms the basis of test execution: visiting a node represents executing the corresponding code segment. The search backtracks by performing user-defined abstract-level undo operations, which enable sharing of results for initial segments of execution common to different tests.

1.3.6 Implementation [6]

We developed a prototype implementation for each of our techniques. We developed Alloy models which create the skeleton for SQL queries and database schemas. These models can be extended to cover additional parts of the SQL grammar and to add user constraints or specifications for more specific test suites. We also developed an Eclipse [39] plug-in for writing Alloy constraints within Java classes. This plug-in is extensible and can be further extended to support mixed-constraints notation.

1.3.7 Experimental Evaluation

We present an experimental evaluation to show the applicability and effectiveness of each of our approaches using a suite of relational applications and commonly used data structures.

1.4 Organization

This dissertation is organized as follows: Chapter 2 presents our approach for mixed-notation formulation syntax and semantics, followed by Chapter 3 describing our approach for mixed constraint solving and experimental evaluation.

Chapter 4 presents our framework which uses test summaries for systematic black-box testing of relational database management systems, and related experimental evaluation on different databases.

Chapter 5 presents our approach for efficient test execution through clustering using abstract undo operation, and experimental evaluation.

Finally, Chapter 6 compares our work with related work in the field of constraint solving, database testing, and optimizing test execution.

Chapter 2

Test Summaries: A Mixed-Constraint Notation

2.1 Overview

The benefits of using specifications in software testing have long been known [53]. Traditional specification-based techniques required much manual effort, and hence posed a burden on the user and remained primarily confined to academic settings. Recent years have seen much work in automation of specification-based techniques. A significant part of automation is due to the advancements in constraint solving techniques, which have been well-supported by the wide availability of faster processors. Witness, for example, the recent resurgence of symbolic execution, which was developed over three decades ago [27], but has only recently started to find its way into industrial settings [111].

While the recent technological advances have increased the effectiveness and efficiency of specification-based techniques, the use of specifications in testing real applications is largely confined to simple assertions, and developers often deem richer forms of specifications more trouble than they are worth. Indeed, manually writing a complex specification, say an *input constraint* that defines complex inputs to a real program, requires careful thought and much effort.

To facilitate writing specifications, researchers have developed frameworks to support different programming paradigms, including support for declarative languages and for imperative languages. However, existing frameworks that provide automated analysis still require using one specific programming paradigm for writing specifications. We believe this requirement is too rigid and seriously impedes a wider scale deployment of specification-based approaches. Moreover, having invested much effort in writing a complex specification, it is natural for developers to expect an efficient and effective automated analysis that provides results that justify the effort.

We believe there are two primary research questions that must be addressed to make specifications more attractive to practitioners: (1) "how to facilitate writing specifications?" and (2) "how to effectively utilize specifications to automate efficient analysis?".

We present JABAL, Java and Alloy Based Language, — a novel framework for mixed imperative and declarative formulation and solving of structural constraints — which addresses both these questions. Our approach enables the user to leverage the expressive declarative language Alloy and the widely used imperative language Java to write and mix constraints freely using the two different paradigms.

JABAL introduces a new notation to support a mixed style for writing specifications that represent *input constraints*, which describe desired test inputs, using a combination of declarative and imperative programming styles. Specifically, the user formulates constraints using a combination of the expressive Alloy language [65] — a declarative, first-order logic based on sets and relations — and the widely used Java programming language. Alloy, with its support for path (navigation) expressions using transitive closure, allows succinct formulations of complex structural properties, such as those of class invariants in object-oriented code. Java, with its wide use, provides a familiar notation that is likely to pose a minimal learning burden. Thus, JABAL does not require the user to follow one specific programming paradigm, rather it enables them to write a constraint using a paradigm they find more suitable for the specific constraint and to mix it freely with other constraints that are written using a different paradigm.

Our framework increases the usability of systematic constraint-based testing by introducing the following:

- **A Mixed constraint notation and its semantics**. We introduce a new notation for writing mixed constraints using the declarative language Alloy and the imperative language Java. We formally define syntax and semantics of our notation.

- **An Imperative-style sequencing in a declarative language**. Our notation introduces the semi-colon operator to Alloy, which enforces an explicit dependency among constraints, akin to sequencing of statements in an imperative language. Specifically, constraints that follow a semicolon assume the constraints that precede it to hold, which makes it easier to formulate constraints using different programming paradigms, e.g., a Java constraint can assume an Alloy pre-condition and not have to check it again.

2.2 Example

We present an example on how to write mixed constraints using our notation and how JABAL proceeds with solving them. We use B-Trees [99] as our example data structure. We first describe the general properties of B-Trees, and then we describe modeling the constraints of the data structure using our mixed constraint notation that is based on Alloy and Java.

A B-Tree is a generalized form of a binary search tree where each node in the tree can have more than two children. B-Trees are often used in database management systems to index the data and to provide fast access, insertions, and deletions in logarithmic time.

Given an order *d*, i.e., the maximum number of children for each node in a B-Tree, the following properties should hold:

1. The tree is full (all leaves have same height).
2. Every node except the root has at least *d/2* children.
3. Every node has at most *d* children.
4. The root is either a leaf node or has at least 2 children.

5. An internal node with m children contains *m + 1* keys.
6. If a node is not a leaf, then for any $0 \leq i < \#keys$,

   $minKey(sub[i]) < keys[i] < maxKey(subs[i+1])$.

Consider modeling the first four structural properties using Alloy [65], and properties 5 and 6, which specify constraints on keys, using Java.

Alloy is a strongly typed specification language. It assumes a universe of atoms (or elements) partitioned into subsets, where every type is associated with a subset. An Alloy specification consists of a sequence of paragraphs where a paragraph enables defining new types, introducing relations between types, and adding constraints on these types and relations. Being an analyzable relational language, Alloy semantics are closely related to those of object oriented programs.

We use Alloy signatures to model the objects of the data structure, and relations on each signature represent the instance variables of each object:

```
one sig BTree {
    root : lone BTreeNode,
    nodeCount : one Int,
    keyCount : one Int
}
sig BTreeNode {
    subs : seq BTreeNode,
    keys : seq Int,
    order : one Int
}
```

The keyword *one* declares the relation $nodeCount : BTree \times Integer$ to be a total function, i.e., each *BTree* must have a node count. The keyword

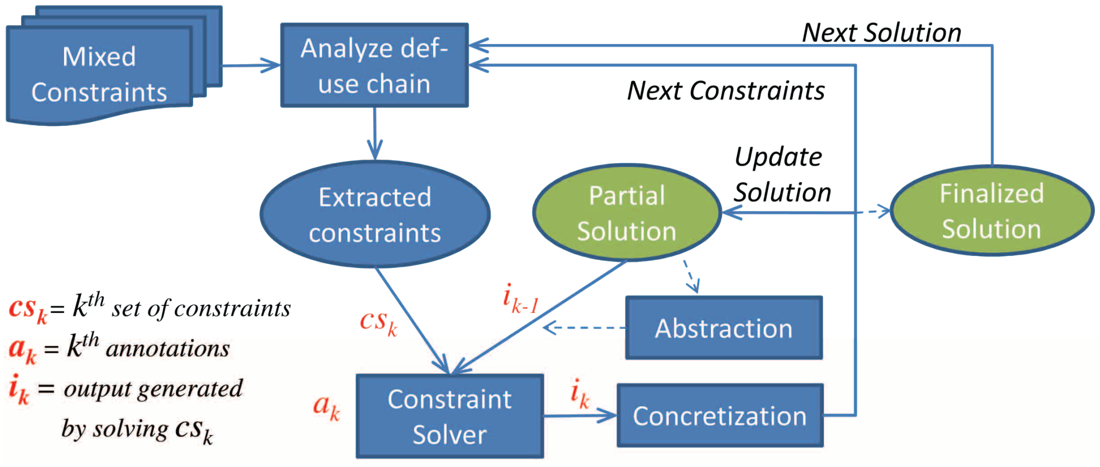


Figure 2.1: JABAL main framework steps for solving mixed-constraints.

lone defines a partial function, i.e., each *BTree* has at most one root. The *seq* construct represents a sequence of elements, which models arrays in Alloy, where each element has a corresponding index. A *BTreeNode* in a B-Tree has a sequence of children which are accessed through the *subs* relation, a sequence of keys accessed by the *keys* relation, and an *order* relation which identifies the maximum and minimum number of keys and children for each *BTreeNode* element.

We model the first four properties in Alloy as follows:

```
pred tree() {
    all node: BTreeNode | lone subs.node //0 or 1 parent
    lone node: BTreeNode | no subs.node //at most one root
}
pred full() {
    #BTreeNode.subs = 0 or (all b1,b2 : BTreeNode |
    #b1.subs=0 and #b2.subs=0 => #b1.~*(subs)= #b2.~*(subs))
}
pred acyclic() {
```

```
         all n: BTreeNode | n !in n.^subs
}
pred orderPreserved() {
     all node: BTreeNode | #node.subs = 0
     or (#node.subs > BTreeNode.order fun/div 2
         and #node.subs <=BTreeNode.order)
}
pred correctNodeCount() {
     BTree.nodeCount = #BTreeNode
}
```

The *tree* predicate states that a *BTreeNode* element can be a child of at most one *BTreeNode* element, plus there is at most one root for the B-Tree. The *full* predicate states that all the leaf elements have the same height by calculating the size of the transpose of *subs* relation of leaf nodes. The tilde operator '∼' represents relational transpose, the '*' operator represents reflexive transitive closure, and the '#' operator represents set cardinality. The *acyclic* predicate states that the tree has no cycles. The *orderPreserved* predicate states that the number of children of a *BTreeNode* can be between *order*/2 and *order*. Finally, the *correctNodeCount* states that the *BTree.nodeCount* should be equal to the total number of nodes.

Next, we model the key constraints in a B-Tree using Java:

```
class BTree {
     BTreeNode root;
     int keyCount;
     int nodeCount;
     ...
     boolean keysAreSorted() { ... }
     boolean searchProperty() { ... }
     boolean correctKeyCount() {
         return keyCount == countKeysRecursively(root);
```

```
    }
}
class BTreeNode {
    int[] keys;
    BTreeNode[] subs;
    int order;  ...
}
```

Finally, to solve the complete set of constraints, we combine the constraints written in Alloy with those in Java and guide the solver by using the following *MixedConstraints* construct of JABAL:

```
MixedConstraints {
    @MixedConstraints(defs = {"BTree.root","BTreeNode.subs",
       "BTreeNode.order"})
    tree[]
    full[]
    acyclic[]
    orderPreserved[];
    java: BTree.correctKeysSubsCount();
    java: BTree.searchProperty();
    java: BTree.correctKeyCount();
    @MixedConstraints (defs = {"BTree.nodeCount"})
    correctNodeCount[];
}
```

We next describe how JABAL solves these mixed constraints. By default, the SAT-based back-end of Alloy is used for solving Alloy constraints, and the Java PathFinder is used for solving Java constraints. The semi-colon separator guides the solving process. Each set of constraints preceding a semi-colon represents one step of solving. The *defs* annotation element provides a guide for solving the corresponding constraints (Section 2.4).

For this *MixedConstraints* example, the first step is to solve the first four predicates. The *defs* element lists the fields that are generated by solving the corresponding predicates. Thus, the Alloy Analyzer generates values for *BTree.root, BTree.subs*, and *BTreeNode.order* variables. These values are used as inputs to the next step to solve the Java-based constraints. In the second step, the Java constraints are solved in the sequential order they appear in. A Java constraint is identified by the *java:* keyword preceding the predicate invocation. JPF is used to solve these constraints on keys. In the final step, the output generated using Java constraints is used as an input to solve the last *correctNodeCount* constraint in Alloy. This constraint is solved using the Alloy Analyzer to generate a valid *BTree.nodeCount* value. The output of the last step is a set of B-Tree data structures with all the instance variables initialized subject to the given constraints.

2.3 Why Mixed Constraints?

Mixed constraints introduce a novel approach for writing and solving constraints. The benefits of the approach are several-fold:

- JABAL eases the learning burden on the user — the ability to freely mix constraints using two different programming paradigms allows the user to choose their favorite paradigm to start using the framework and then to gradually learn the other paradigm, which provides flexibility and convenience in constraint formulation.
- JABAL provides a novel way of combining solvers that were originally de-

signed for different programming paradigms, which allows our approach to support new powerful analysis, e.g., the combination of Alloy and JPF could even enable a novel approach for checking multi-threaded data intensive applications.

- JABAL's support for non-conventional solver combinations opens an avenue for novel techniques for highly optimized solving.
- The unification of two different paradigms allows traditional analysis that were originally designed for programs in a particular paradigm to be applied to programs in another paradigm, thereby enabling a host of new analysis for existing programs, e.g., a traditional pointer analysis could provide the basis of slicing an Alloy specification.
- JABAL provides a fresh perspective on incremental and online solving, which is likely to become the backbone of future development environments that provide continuous feedback to the developer.

2.4 Syntax and Semantics

Our framework extends Alloy's grammar to support writing mixed constraints. Fig. 2.2 presents syntax for a core of the Alloy grammar[1] as well as the additional syntax introduced by JABAL. The grammar uses the standard BNF operators, in addition, *x,** means zero or more comma-separated occur-

[1]For complete syntax of the Alloy grammar, refer to [65].

rences of *x*. Every name ending with *ID* is an identifier and is a terminal. Bold names are terminals as well.

The extended grammar of the framework is only used within the *Mixed-Constraints* block. We represent a set of either Alloy constraints or Java constraints by a *constraintSet*. Each constraint set is separated from the other by the use of a semicolon. Similar to commonly used imperative languages where the semantics of a semicolon is to declare the end of an execution statement, a semicolon in JABAL declares the end of a set of constraints, which are to be solved together. A constraint set can have an optional corresponding annotation. The annotations conform to the following definition given using the familiar style of defining annotations in Java:

```java
public @interface MixedConstraints {
    String[] defs();
    String[] uses();
    String solver();
}
```

The *Mixed-Constraints* annotation has three elements: (1) the *defs* element representing the set of fields which the constraint solver *defines* by solving the corresponding constraint, (2) the *uses* element representing the set of fields which the constraint solver *uses* in order to solve the corresponding constraints, and (3) the *solver* name which is responsible for checking constraints and/or generating values for the fields in the data structure.

JABAL uses the annotations to analyze the role of a predicate in generating certain fields of the data structure and analyzes the information in

module ::= **module moduleID** *paragraph**
paragraph ::= *sigDecl* factDecl* funDecl* predDecl* mixedPred runCmd*

sigDecl ::= [**abstract**] **sig** sigID,+ [**extends** sigID] *sigBody*
sigBody ::= {(varID : sigID),*} [{*alloyConstraints*}]

factDecl ::= **fact** [factID] *alloyConstraints*
funDecl ::= **fun** funID (***var***:*expr*,*) : *declExpr expr*
predDecl ::= **pred** predId (***var***:*expr*,*) *alloyConstraints*

mixedPred ::= **MixedConstraints {** *mixedConstraints* **}**
mixedConstraints ::= [*annotation*] *constraintSet* **;** mixedConstraints | $\emptyset$
constraintSet ::= *alloyConstraints* | *javaConstraints*
alloyConstraints ::= *formula**
javaConstraints ::= **java:** BaseClassID.predicateMethodID() *

annotation ::= **@MixedConstraints {** *use*, *def*, *solver* **}**
use ::= **use = {** *fieldID*,* **}**
def ::= **def = {** *fieldID*,* **}**
solver ::= **solver = "***solverID***"**
fieldID ::= "BaseClassID.instanceVariableID"
solverID ::= **"miniSat"** | **"ZChaff"** | **"Sat4j"** | **"JPF"** | **"JVM"**

formula ::= *expr* [!]**in** *expr*
| [**all** | **no** | **lone** | **one** | **some**] *expr*
| !*formula*
| *formula* **and** *formula*
| *formula* **or** *formula*
| **all** v : type | *formula*
| **some** v : type | *formula*

Figure 2.2: Extended grammar of Alloy supported by JABAL.

expr ::= expr **+** expr
    | expr **&** expr
    | expr **-** expr
    | ∼expr
    | expr.expr
    | ∗expr
    | ˆexpr
    | funID (expr,*)
    | *Var*

Var ::= **var**
    | Var[**var**]

runCmd ::= **run MixedConstraints** [scope]

Figure 2.2: Extended grammar of Alloy supported by JABAL.

them to check and warn for proper user guidance of mixed constraint solving. The Alloy Analyzer uses SAT4J and MiniSat as incremental SAT solvers to enumerate all possible solutions satisfying the model's constraints, while the other SAT solvers only provide one solution satisfying the constraints if any exists. For example, the user can decide to solve a set of Alloy constraints using MiniSat solver, enumerating all possible valuations satisfying those constraints and check for other constrains via ZChaff solver in order to prune out invalid solutions.

The framework permits executing Java code on the data structure as an independent step of the complete generation process. To execute Java code without any automatic generation of values for fields, the user can set the

$$\begin{aligned}
&M\ :\ formula \rightarrow env \rightarrow boolean\\
&X\ :\ expr \rightarrow env \rightarrow value\\
&env = (var + type) \rightarrow value\\
&value = (atom \times ... \times atom) + (atom \rightarrow value)
\end{aligned}$$

$$\begin{aligned}
&M\ [a\ \mathbf{in}\ b]\ e = X\ [a]\ e \subseteq X\ [b]\ e\\
&M\ [\mathbf{!}F]\ e = \neg M\ [F]\ e\\
&M\ [F\ \mathbf{and}\ G]\ e = M\ [F]\ e \wedge M\ [G]\ e\\
&M\ [F\ \mathbf{or}\ G]\ e = M\ [F]\ e \vee M\ [G]\ e\\
&M\ [\mathbf{all}\ v : t \mid F]\ e = \bigwedge\{M\ [F]\ (e \oplus v \mapsto x) \mid (x, unit) \in e(t)\}\\
&M\ [\mathbf{some}\ v : t \mid F]\ e = \bigvee\{M\ [F]\ (e \oplus v \mapsto x) \mid (x, unit) \in e(t)\}
\end{aligned}$$

Figure 2.3: Semantics of formulas in Alloy.

$$\begin{aligned}
&CS : a\ constraintSet\ term\\
&MC : a\ mixedConstraints\ term
\end{aligned}$$

$$\begin{aligned}
&M\ [CS]\ e = \bigwedge\{M\ [F]\ e \mid F \in CS\}\\
&M\ [CS\ ;\ MC]\ e = M\ [MC]\ (e \oplus e') \mid M\ [CS]\ e'
\end{aligned}$$

$$\oplus(e,\ e') = (e \cup e') - \{r \in e \mid (r.var) \in e'.var\}$$

Figure 2.4: Semantics of mixed constraints in JABAL.

solver element in the annotation to *JVM*, indicating that the Java Virtual Machine is running the code. This is useful for testing methods by checking the correctness of the post-state of the data structures upon executing the code. In addition, it is an easy way to modify the state of the data structure before validating the remaining set of constraints.

Fig. 2.3 shows the main semantics for formula operations in Alloy language. There are two functions: M which interprets a formula as true or false, and X which interprets an expression as a value. An *env*, environment, is a mapping of variables into values representing an Alloy instance; it is a set of valuations for the variables in the model to concrete values or relations between atoms. To define the sequential notation of solving constraints, we extend the definition of function M to interpret the use of semicolons between constraints. Fig. 2.4 shows the extended semantics for our mixed notation reflecting the semantics of the semicolon. Solving for a constraint set means that all the functions belonging to the set must be satisfied via the environment. A mixed constraint is a sequence of constraint sets. Each constraint set is solved based on the environment by which the previous constraint set was satisfied.

To solve for all solutions of the mixed constraints, we define a function S, which maps constraints in the context of their corresponding annotations and a concretization of the latest instance into a new instance:

$S : (annotations, constraintSet, instance) \rightarrow instance$

Consider a mixed constraint predicate MC as follows:

$MC = cs_1; cs_2; ...; cs_k;$

Solving MC implies finding all instances, i.e. environments, i_k such that :

$i_0 = \emptyset$ *(empty instance with no bindings of variables)*

$S[cs_1]a_1, i_0 = i_1$

$S[cs_2]a_2, i_1 = i_2$

...

$S[cs_k]a_k, i_{k-1} = i_k$

This property is defined in the semantics by having a mixed constraint *MC* satisfiable by *($e \oplus e'$)* where e' is the instance which satisfied the previous constraint set *CS*. *($e \oplus e'$)* implies an environment where valuations in e are overridden by the valuations of e'.

Finally, running JABAL requires providing the *run* command, as defined in the Alloy language, with *MixedConstraints* as its first parameter and the scope of elements of the model similar to usual Alloy *run* commands.

Chapter 3

Solving Mixed Constraints

3.1 Overview

Our framework improves the efficiency of systematic constraint-based testing by introducing new techniques for solving input constraints — using a combination of solvers that support input constraints written in different programming paradigms. Specifically, JABAL leverages the Alloy Analyzer [63] — a fully automatic SAT-based tool-set for checking Alloy formulas within a given *scope*, i.e., bound on the universe of discourse tool — and the Java PathFinder (JPF) [108] — an explicit state, open-source software model checker for Java — and uses them in synergy for solving structural constraints and enable automated test generation. To optimize solving, we introduce *def-use* annotations, which states the sets of fields defined and used by a constraint.

A key challenge in solving constraints written using a combination of paradigms is to handle the likely differences in data models native to each paradigm. For example, Alloy's data model is entirely based on sets and relations over atoms, whereas Java data-types include primitives, references, and arrays. To bridge this gap in the data models, we use data translations based on a relational model of the program heap: abstraction translations translate Java data structures into Alloy instances and concretization translations translate Alloy instances to concrete Java data structures. Such translations were originally developed for specification-based testing of Java programs using the Alloy tool-set [71].

3.2 Framework

This section describes the framework for writing and solving declarative and imperative constraints in synergy for automatic generation of complex data structures. JABAL allows the user to define constraints declaratively in the Alloy language, in addition to describing constraints imperatively using the Java programming language, which provides flexibility in writing constraints and reduces the learning burden on the user. Additionally, the user can guide solving those constraints using annotations provided by JABAL. This allows the user to benefit from strengths of different constraint solvers.

Fig. 2.1 shows an overview of the framework. It takes as input a set of mixed constraints written in Alloy and Java defined by the *MixedConstraints* construct. The first step of the framework is to divide the constraints into separate groups based on their type and annotations provided by each group.

In the next step, each group of constraints is solved using a solver for those constraints. For Alloy constraints, the user can choose between different SAT solvers, and for Java constraints the user can choose between using the Java PathFinder as a solver or directly running the Java code as given using the Java virtual machine (JVM) to set desired field values of the input partial solution. A solving step may generate several solutions (if desired). Each solution stored and used in turn. Since solutions generated by the Alloy Analyzer are Alloy instances, we apply a concretization translation to convert them into concrete Java data structures. Each partial solution is input for solving the next group of constraints. To support arbitrary interleaving of constraints, we apply an abstraction translation to convert Java data structures to Alloy instances.

Each group of constraints is used to generate values for certain fields of the complete data structure being generated. Based on the annotations on each group of constraints, the input partial solution might be either updated to include the changes on the fields generated or it might be pruned from the set of partial solutions if cannot be updated to satisfy the group's constraints.

3.2.1 Solving mixed constraints

Alloy's relational basis provides a natural embedding of the Java heap into the Alloy data-model. The following translation template shows how our approach models the basic Java types in Alloy:

```
class A {}                    sig A {}
class B extends A {           sig B extends A {
    T x;                          x : T,
    T[] y;                →       y : seq T,
    B z;                          z : lone B
}                             }
```

T represents a primitive data type; B represents a reference to an object type and can be *null*. Not all primitive types supported by Java are supported by Alloy. We enable user defined translations for certain types. For example, for Boolean variables we define an abstract signature Boolean in Alloy and have two other signatures extend it to represent true and false values:

abstract sig Boolean {}

one sig true, false extends Boolean {}

Even though Alloy does not support all primitive types, JABAL can use the mixed constraint solvers to solve for desired primitive values using Java solvers and combine the solutions with Alloy's valuations. Fig. 2.1 shows the framework for integrating outputs from different constraint solvers. Starting with the mixed constraints predicate defined by the user, JABAL performs analysis on the annotations of the constraints within the mixed constraint block and extracts the first set of constraints to be solved. Initially, the partial solution set is empty, so the output of solving the first constraint is stored as the first partial solution. We run a concretization algorithm, based on previous work [71], to update the partial solution. The concretization step converts Alloy's abstract instances into concrete data structures updating the changes to any fields which exist or adding new fields valuations to the partial solution. Once the first set of constraints is solved, JABAL proceeds to solve the next set. It makes sure that the next set of constraints are solved based on the current partial solution. An abstraction translation is used to update the Alloy model with the partial solution in order to achieve the desired output.

3.2.1.1 Analyzing def-use chain

The first step in solving the mixed constraints is analyzing the annotations of every constraint group. JABAL analyzes the *def-use* chain between all the constraints in sequence within the *MixedConstraints* block. Fig. 3.1 shows the algorithm for our analysis. It validates that the *uses* of the first constraint set is empty, and a global set of *defs* is maintained by adding to it the fields that are solved for at each step. Following the first constraint set, it validates that the *uses* set of the consequent constraint set is a subset of the *globalDefs* set.

For example consider the following two predicates with their corresponding annotations:

```
@MixedConstraints (
  uses = {}, defs = {"BTree.root", "BTreeNode.subs",
    "BTreeNode.order"}, solver = "SAT4J")
tree[];

@MixedConstraints (
  uses={"BTree.root","BTreeNode.subs","BTreeNode.order"},
    defs = {"BTreeNode.keys"}, solver = "JPF" )
java: BTree.correctKeysSubsCount();
```

The def-use analysis checks that the global *defs* set, at the time of solving the second constraint, contains all the *uses* elements of the second constraint. Then after solving the second constraint it adds to the *defs* set the *defs* elements of the second constraint. The process is similar to the remaining constraints. As a default setting, if a *defs* set of a constraint is empty then we assume that it generates all elements which have not been previously defined.

```
$Set\ globalDefs \leftarrow \emptyset$
for each $annotation$ in $Annotation\ Sequence$ do
    $localDefs \leftarrow$ definitions using static analysis
    $localUses \leftarrow$ uses using static analysis
    //Check for Errors
    if $annotation.uses \not\subseteq globalDefs$ then
        ERROR def-use chain broken
    end if
    //Check for Warnings
    if $localDefs \not\subseteq annotation.defs$ then
        WARN fields defined not in annotation defs set
    end if
    if $localUses \not\subseteq annotation.uses$ then
        WARN fields used not in annotation uses set
    end if
    if $annotation.defs \cap globalDefs \neq \emptyset$ then
        WARN multiple field generation
    end if
    //Update global defs set
    if $annotation.defs = \emptyset$ then
        $globalDefs \leftarrow globalDefs \cup localDefs$
    else
        $globalDefs \leftarrow globalDefs \cup annotation.defs$
    end if
end for
```

Figure 3.1: Algorithm for def-use analysis.

JABAL also performs a static analysis of the code and reports any fields which are used or defined that are not included in the annotations. It is not necessary to include all the fields in the annotations, however the analysis produces warnings to avoid missing fields or generation of fields accidently. Since solving constraints in an imperative order can update fields independently of previous constraints as the user desires and describes in the model, our framework can *not* claim that solving all the constraints in sequence implies that *all* the constraints are satisfied at the end of the process. This claim cannot be achieved since each constraint set can update fields that might break the validity of a previous constraint. JABAL warns the user about possible overlapping of fields generation or update by multiple constraint solvers.

3.2.1.2 Abstraction

The purpose of the abstraction step is to convert a concrete data structure into Alloy, and enforce the state of that structure into the model. We abstract the partial solution in an Alloy model when the next solver is a SAT solver. The algorithm traverses all the objects and their fields in the partial solution and maps them into corresponding Alloy segments. However, not all of the fields are included in the abstraction. The set of fields which do not belong to neither the *uses* nor the *defs* sets of the annotation of the next constraint set are excluded from the abstraction process. This selection of fields which is done at every abstraction call makes constraint solving faster. For example, having the model of a B-Tree in the example and solving for acyclicity of the structure, the Alloy Analyzer does not have to solve for any key fields, yielding a smaller set of clauses to solve by the SAT solver.

For example, based on the definition in section 2.2, a BTree with one root and two BTreeNode elements as its subs can be abstracted as follows:

```
abstract sig BTree { ... }
abstract sig BTreeNode { ... }

one sig BTree_0 extends BTree {}

one BTreeNode_0 extends BTreeNode {}
one BTreeNode_1 extends BTreeNode {}
one BTreeNode_2 extends BTreeNode {}

fact {
    BTree_0.root = BTreeNode_2
    BTreeNode_0 = BTreeNode_2.subs[0]
    BTreeNode_1 = BTreeNode_2.subs[1]
}
```

Notice the addition of *abstract* keyword to the main signatures of BTree and BTreeNode. Its purpose is to explicitly define those sets by the elements that extends them. Thus there is only one BTree element that can exist, namely the BTree_0 element, similarly for the three BTreeNode elements. The *fact* block represents a constraint that should hold true for all instances generated. This abstraction considers that the constraints at this step use the root and subs fields, and they are not altered.

3.2.1.3 Concretization

The concretization process is required to convert Alloy instances generated by the Alloy Analyzer from their abstract state into concrete Java objects. The main approach is taken from [71], however our approach is different because we need to take into consideration the existence of a partial solution that needs to be updated at each step.

For every element in an Alloy instance, we check if there is a Java object corresponding to that element ID in the partial solution. If the object exists, then we update the instance variables of that object by the relation fields in the Alloy instance. Otherwise, we create a new Java object and map it to a unique ID.

For example, consider the following Alloy instance:

```
BTree = {BTree$0}
BTree<:root = {BTree$0->BTreeNode$0}
BTreeNode = {BTreeNode$0, BTreeNode$1, BTreeNode$2}
BTreeNode<:subs = {BTreeNode$0->0->BTreeNode$1, BTreeNode$0
    ->1->BTreeNode$2}
```

The solution for every signature and relation is a set of valuations. We translate the Alloy instance into Java objects equivalent to the following code[1]:

```
BTree BTree$0 = new BTree();
BTreeNode BTreeNode$0 = new BTreeNode();
BTreeNode BTreeNode$1 = new BTreeNode();
BTreeNode BTreeNode$2 = new BTreeNode();
BTree$0.root = BTreeNode$0;
BTreeNode$0.subs[0] = BTreeNode$1;
BTreeNode$0.subs[1] = BTreeNode$2;
```

3.2.1.4 Maintaining partial solutions

Every solution generated by solving a set of constraints updates the partial solution. To track the changes on the data structures through different steps of solving the constraints, JABAL maintains a map for every object created and a unique ID associated with it. Each of the abstraction and concretization steps search this map table for existing objects by ID to update it. Abstracting the data structure to Alloy uses the map table to name the unique signatures for each data element. This guarantees that the abstract representation maps to the same concrete representation.

[1]We use reflection to create the objects.

Each partial solution can be the input to a constraint set, which upon solving generates a set of other partial solutions. Since this approach can generate numerous possibilities, we try to limit the number of solutions upon reaching a threshold value.

3.2.2 Solving Java constraints using JPF

In this section we describe the use of JPF solver in solving Java constraints. Java PathFinder (JPF) is an explicit state model checker for Java programs. It has been used for test input generation in [109] and for finding errors in complex systems in [10]. We use a similar approach to test input generation using JPF with several optimizations.

After analyzing the *defs* set of the annotations of a Java constraint, JABAL instruments the Java code to automate the generation of values for *defs* fields. Performing analysis on the code, JABAL replaces the use of any field in the *defs* set by a corresponding method call which uses JPF's main Verify class that implements non-deterministic choices over a range of possible valuations. We explain this approach by an example.

Consider the following Java predicate method which checks if the keys in a BTreeNode, defined in Section 2.2, are sorted:

```
@MixedConstraints (
        uses = {"BTree.root", "BTreeNode.subs"},
        defs = {"BTreeNode.keys"}, solver = "JPF"
)
boolean keysAreSorted() {
        for (int i = 0; i < numKeys - 1; i++) {
                if (keys[i] >= keys[i + 1]) {
```

```
                                        return false;
                            }
                }
                return true;
}
```

The annotation for the predicate method specifies that it should be solved after the *root* and the *subs* fields have been defined, which is expected since we cannot solve for *keys* for an undefined structure. JABAL reads the *defs* sets and instruments the code to generate *keys* values using JPF. Every use of the *keys* fields is replaced by a call to *_get_keys(int i)* which is defined as follows:

```
int _get_keys(int i) {
      int choice = Verify.random(intVector.size() - 1);
      keys[i] = intVector.elementAt(choice);
            return keys[i];
}
```

The *intVector* is a vector containing all possible integer values based on the scope for integers defined by the user. The *Verify.random(n)* method returns values [0, n] nondeterministically. If a constraint is not satisfied after calling *_get_keys*, JPF back-tracks and assigns a new value to *keys[i]* until all possible values are explored. We use *Verify.randomBool()* method to return a Boolean value nondeterministically, and *Verify.ignoreIf(cond)* to force the model checker to backtrack when *cond* evaluates to true.

One of the optimizations that we use upon generating object types is lazy initialization [73]. The idea is to initialize fields when they are first accessed. The algorithm nondeterministically initializes a newly accessed field to either null, a reference to a newly created instance of the object with uninitialized fields, or to a reference of an object created during a prior field initialization. Thus, we add to each field in the class declaration a Boolean field to determine whether the field has been previously assigned a value or not. Once a field is accessed we set it to true. For example, the previous code is instrumented as follows[2]:

```
int _get_keys(int i) {
    if (!_keys_initialized[i]) {
        int choice = Verify.random(intVector.size() - 1);
        keys[i] = intVector.elementAt(choice);
        _keys_initialized[i] = true;
    }
        return keys[i];
}
```

Experiments have also shown that setting JPF choice generator to randomized generator can lead to a better performance especially for selecting primitive values.

3.3 Experiments

In this section, we demonstrate potential benefits of JABAL in generating complex data structures. We consider the following two research questions: (1) "Can JABAL perform better than SAT?", and (2) "Can JABAL perform better than JPF?". We perform three case studies tailored at generating different data structures as inputs. The experiments are run on an Intel Core i7

[2]For object references, we instrument a _get_new_field() to help the nondeterministic choice. Refer to [73] for details.

CPU (Quad Core) 2.8GHz with total 8GB RAM.

3.3.1 Comparison of Alloy/SAT and JABAL

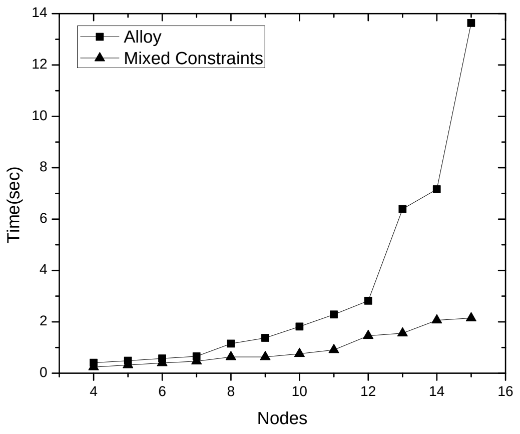
We perform three experiments to compare the performance of JABAL with Alloy/SAT. In the first experiment, we generate sorted singly linked lists where each node in the list contains an integer value. We model the linked list in Java as follows:

```
class List {
        Node head;
        int size;
}
class Node {
    int value;
    Node next
}
```

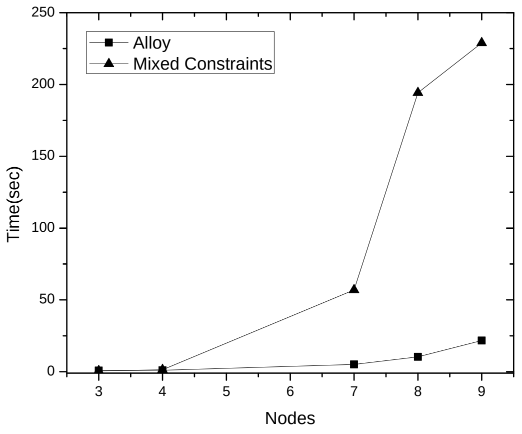
The constraints on the linked list are the following:

1. Acyclic: the list has no cycles.
2. Sorted: the values in the list are sorted, i.e. for each node n except the last, $n.value < n.next.value$.
3. Correct size: the size field value of the linked list is equal to the number of nodes in the list.

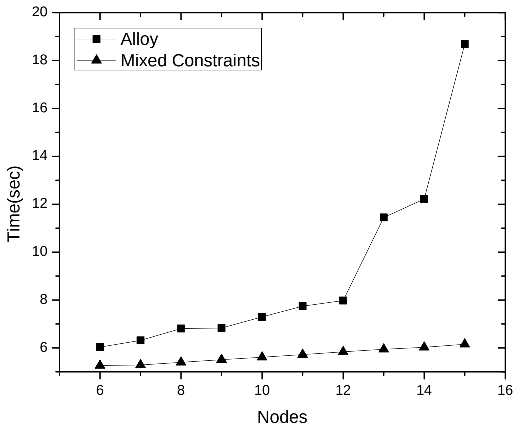
To measure the performance of JABAL compared with SAT, we model the linked list data structure and its constraints in Alloy, and use the Alloy Analyzer to solve it. On the other hand, we run JABAL on mixed constraints



(a) Sorted linked lists



(b) B-Trees of order 3



(c) Linked lists with B-Tree of size 7

Figure 3.2: Comparison of SAT and JABAL.

written using Alloy and Java. We select Alloy to express acyclicity and Java to express the other two properties. The acyclicity constraint defines the *List.head* and *Node.next* fields in the linked list, while the other two constraints define the *List.size* and *Node.value* fields.

Fig. 3.2a shows the results of this experiment. The values shown represent the time needed to generate the first data structure which satisfies all the constraints. We compare results based on the size of the data structure. JABAL outperforms Alloy as the number of nodes increases because it benefits for the fast performance of JPF in solving linear integer constraints on the linked list, and as the number of nodes increases, the integer representation in Alloy expands exponentially yielding poor results for larger lists.

In the second experiment, we generate B-Tree data structures, which were introduced in Section 2.2. We model the B-Tree and its constraints solely in Alloy and compare it to JABAL where properties 1 to 4 are written in Alloy while properties 5 and 6 are written in Java.

Fig. 3.2b shows the results of the second experiment. Alloy performs better than JABAL since JPF, which is used for solving properties 5 and 6, performs a brute force search to solve these constraints.

In the third experiment, we consider generating inputs for a method that has two parameters. Specifically, we consider generating data structures to test a method in the *BTree* class which adds elements from a linked list into the B-Tree receiver object. We use JABAL to generate both inputs simultaneously: (1) the B-Tree receiver objects, and (2) the linked lists parameter objects. For mixed constraints, we express linked lists constraints using Java and the B-Tree constraints using Alloy; solving is done by JPF and Alloy Analyzer. To compare with SAT, we generate both inputs using SAT alone.

Fig. 3.2c shows the results of generating the first B-Tree of order 3 and size 7 with linked list of different sizes. JABAL outperforms SAT in generating the data structures in conjunction since it benefits from SAT to generate B-Trees and JPF to generate linked lists.

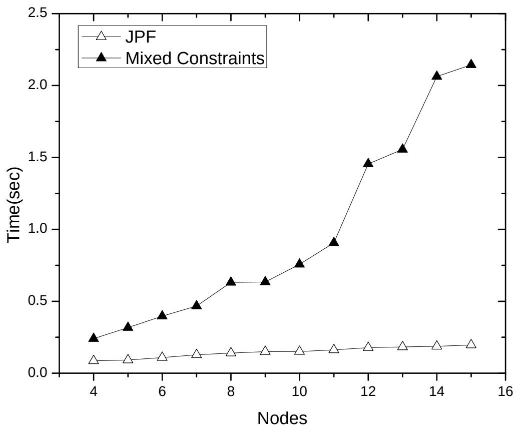
Overall, JABAL provides more efficient solving than SAT in two of three studies.

3.3.2 Comparison of JPF and JABAL

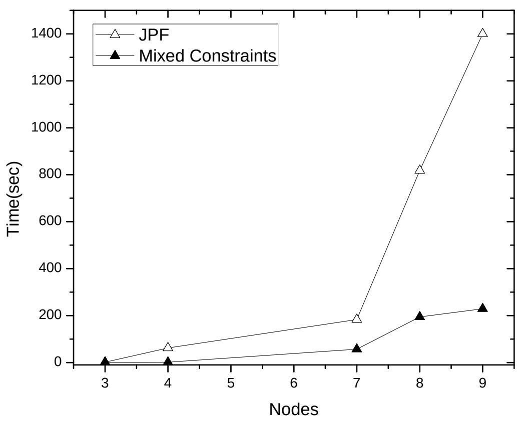
To answer the second research question, we perform three experiments similar to the first set of experiments. However, in these experiments we compare JABAL to JPF. We write the constraints of the data structures purely in Java and use JPF to solve for all the fields and compare it to solving mixed constraints using JABAL.

Fig. 3.3a shows the results of the first experiment in generating sorted singly linked lists. JPF performs better than JABAL due to the optimization techniques described in Section. 3.2.1.3.

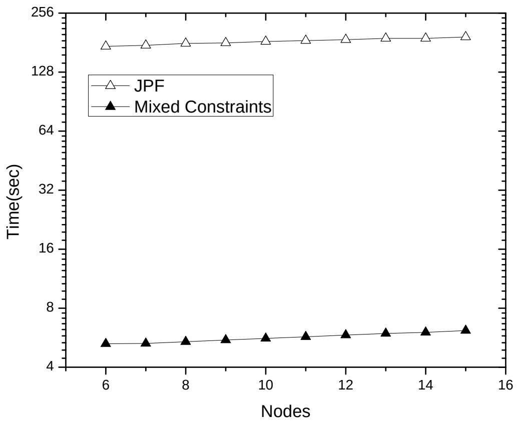
Fig. 3.3b shows the results of the second experiment in generating B-Trees of order 3. JABAL performs better than JPF. The complexity of B-Trees



(a) Sorted linked lists



(b) B-Trees of order 3



(c) Linked lists with B-Tree of size 7

Figure 3.3: Comparison of JPF and JABAL.

make the generation by JPF inefficient, while JABAL is able to benefit from Alloy and JPF combined.

Similar to the first set of experiments, Fig. 3.3c shows the results of generating the first B-Tree of order 3 and size 7 with linked list of different sizes. JABAL outperforms JPF in generating the data structures in conjunction since it benefits from SAT to solve B-Trees and JPF to solve for Linked Lists.

Overall, JABAL provides more efficient solving than JPF in two of three studies.

Chapter 4

Test Summaries for Testing Relational Database Engines

In this Chapter, we discuss test summaries as a basis for systematic black-box testing of relational database management systems (DBMS). There are three fundamental steps in testing a DBMS: (1) generating test queries with respect to a database schema, (2) generating a set of test databases (tables), and (3) generating oracles to verify the result of executing the queries on the input databases using the DBMS.

In the following sections, we discuss the details of our framework which combines the three major steps together. In addition, we present experimental results which show the ability of the framework in detecting bugs in different database management systems.

4.1 Overview

Database management systems have been used widely for decades. They are steadily growing in complexity and size. At the same time reliability is becoming a more vital concern; the cost of user data loss or wrong query processing can be prohibitively expensive. DBMS testing, in general, is a labor intensive, time consuming process, often performed manually. For example, to test the correctness of a query execution, the tester is required to populate the database with interesting values that enable bug discovery, and manually check the execution result of the query based on the input data. Automating DBMS testing not only reduces development costs, but also increases the reliability in the developed systems.

We presents a novel use of test summaries to automate systematic testing of database management systems. There are three fundamental steps in testing a DBMS: (1) generating test queries with respect to a database schema, (2) generating a set of test databases (tables), and (3) generating oracles to verify the result of executing the queries on the input databases using the DBMS.

The insight of our work is that a relational engine backed by SAT provides a sound and practical basis of a unified approach that supports all the three fundamental steps in DBMS testing and allows generation of a higher quality test suite: queries generated are valid, database states generated are query-aware, and expected outputs represent meaningful executions. Thus, each test case checks some core functionality of a DBMS.

We have leveraged our work on ADUSA[1] [1] to provide a framework which fully automates the process of systematic testing of database management systems. ADUSA is a framework for query aware test generation. It uses

[1]ADUSA: Automated database testing using SAT solvers.

```
CREATE TABLE students(     CREATE TABLE grades(
    id int,                    studentID int,
    name varchar(50)           courseID int,
);                             grade int
                           );
```

Figure 4.1: Example database schema.

the Alloy Analyzer which in turn uses SAT to generate test data and test oracles. Syntactically and semantically valid SQL queries combined with query aware data generators and the expected output oracles is the backbone of our framework. The framework builds on that and automates the validation of each test suite on a DBMS, reporting any errors and information to reproduce them.

4.2 Example

In this section we illustrate our testing approach by applying our framework on a test database schema. We start with describing our test input database schema and show how the framework produces the corresponding Alloy specifications which generate (1) SQL queries to test the database, (2) input test data to populate the database, and (3) the expected output of running each query on every set of test data.

Let us consider a sample database schema as shown in Fig. 4.1. These SQL statements create two relations (aka tables): (1) `students` table with two attributes, `id` of type `int`, and `name` of type `varchar`, (2) `grades` table with three attributes, `studentID` of type `int` representing a student ID number, `courseID` of type `int` representing the course ID number and `grade` of type `int` representing the grade which the student earned in that course.

The first step in our framework is to automatically generate valid SQL queries for testing. Let us consider a subset of SQL grammar consisting of selecting up to two table attributes from either one table or cross product of two tables. The terminal strings of the grammar are the table names and attribute names: *students, grades, id, name, studentID,* and *courseID*. In addition, we consider that the grammar allows the use of aggregate functions when selecting a field. We consider `MAX` and `MIN` aggregate functions in this example. Below is the grammar of SQL queries that we consider in this example:

```
QUERY ::= SELECT FROM
SELECT ::= 'SELECT' selectTerm+
FROM ::= 'FROM' (table | table JOIN table)
selectTerm ::= term | agg(term)
table ::= 'students' | 'grades'
term ::= 'id'|'name'|'studentID'|'courseID'|'grade'
agg ::= 'MAX' | 'MIN'
```

After automatically generating the complete Alloy model for this SQL grammar, the Alloy analyzer, based on SAT, converts all Alloy formulas into Boolean formulas and enumerates all possible solutions satisfying the model. We run the output through our concretization program to convert Alloy instances into complete SQL queries. For the grammar in this example, considering up to two SELECT terms, up to two FROM tables, and two aggregate

```
SELECT courseID, studentID FROM GRADES, STUDENT;
SELECT MAX (courseID), MAX (studentID) FROM GRADES, STUDENT;
SELECT MIN (courseID), MIN (studentID) FROM GRADES, STUDENT;
SELECT courseID FROM GRADES, STUDENT;
SELECT MAX (courseID), MIN (NAME) FROM GRADES, STUDENT;
SELECT MAX (courseID), MIN (courseID) FROM GRADES;
SELECT MAX (studentID), MIN (studentID) FROM GRADES;
SELECT MAX (grade), MIN (grade) FROM GRADES;
SELECT grade, MAX (grade) FROM GRADES;
...
```

Figure 4.2: Example of a sample SQL queries generated by our approach.

functions, we get 186 unique non-isomorphic[2] SQL queries generated, which is what we would expect. Fig. 4.2 is a sample subset of the SQL queries generated by our approach for this example.

For each of the SQL queries automatically generated, ADUSA automatically generates test input and test oracle to verify the output of the query upon running it on a database management system. The use of Alloy enables specifying constraints on both the query as well as the results which enables more precise test input generation. Using the same `students` table schema described above, the framework adds Alloy constraints to model the relational properties between the tables of the database schema such as primary and foreign key constraints. The following Alloy specification models the `students` schema representation:

```
one sig student {
```

[2]In this example, two queries are considered isomorphic if they only differ in the order at which the SELECT or the FROM tables are used.

```
    rows: Int -> varchar
} {
    all x: rows.varchar | one x.rows
}
```

ADUSA continues by adding Alloy functions to model the SQL query under test. Each section of the SQL statement is modeled as a separate Alloy function which enforces the constraints on the data selected by the query. For example, the following Alloy paragraphs model the query `SELECT id FROM STUDENT;` :

```
fun query () : Int {
    {select[from[student.rows]]}
}
fun select (rows: Int -> varchar) : Int {
    {rows.varchar}
}
fun from(rows: Int -> varchar) : Int -> varchar {
    {rows}
}
```

Functions (fun) used by Alloy represent named expressions. A fun paragraph takes a relation as input and returns a relations with similar or different arity. As the SQL queries get more complicated, ADUSA adds more functions and predicated to model the WHERE, GROUP BY, and HAVING clauses. After generating the Alloy specification for the schema and the query, ADUSA uses the Alloy Analyzer to find database instances that satisfy the specification, i.e. provide valuations for the types and relations that satisfy all the constraints. The Alloy Analyzer finds all the instances within a given scope, i.e., a bound on the number of data elements of each type to be considered while generation. Below is an example of an instance generated by the Alloy Analyzer:

```
varchar: {varchar$0, varchar$1}
student:{student$0}
rows: {(1, varchar$1), (2, varchar$0), (4, varchar$1)}
$query: {1, 2, 4}
```

In the above instance, the `varchar` set has two elements labeled as `varchar` followed by the element number. The rows relation consists of three tuples, each of which represent a row in the `student` table. The `query` set represents the result of executing the query on the `rows` relation. Since the example query is `SELECT id FROM STUDENT;`, the `query` set holds the `id` attribute of the tuples, thus it is the set of integers {`1, 2, 4`}.

After generating the Alloy Instance, ADUSA translates the instance into `INSERT` SQL queries that are used to populate an empty database. For example, for the above Alloy instance, ADUSA identifies the `rows` relation and generates the following `SQL` statements:

```
INSERT INTO student VALUES (1, varchar$1)
INSERT INTO student VALUES (2, varchar$0)
INSERT INTO student VALUES (4, varchar$1)
```

Once the database is populated with data, the given SQL query is executed on the DBMS and the result is verified with the one found in the Alloy instance. The process is repeated for each generated instance as well as for each generated SQL query.

4.3 Framework for DBMS Testing

In this section, we discuss the general algorithm for our approach. We describe the integration of the automated SQL generator with ADUSA to create a systematic fully automated testing framework. The full framework models the syntax and semantics of both SQL queries and relational databases without considering specific DBMS implementation details.

Figure 4.3 shows the complete framework for DBMS testing. Boxes represent the processing modules; ovals represent the inputs and outputs of these modules. The main components of the framework are divided as follows:

1. The Automated SQL Query Generator component, described in Section 4.4, models the syntax and semantics of valid SQL queries. It includes two main modules:

   (a) The `SQL Model Generator` module which generates an Alloy specification by modeling the user requirements in addition to the main syntax and semantic requirements of valid SQL queries.

   (b) The `Alloy2SQL_Query Translator` module which translates the Alloy instances into SQL query statements that are used to test the database management systems.

2. The ADUSA component which automates the generation of test databases and the expected output for a given test SQL query. It is comprised of the following sub-modules:

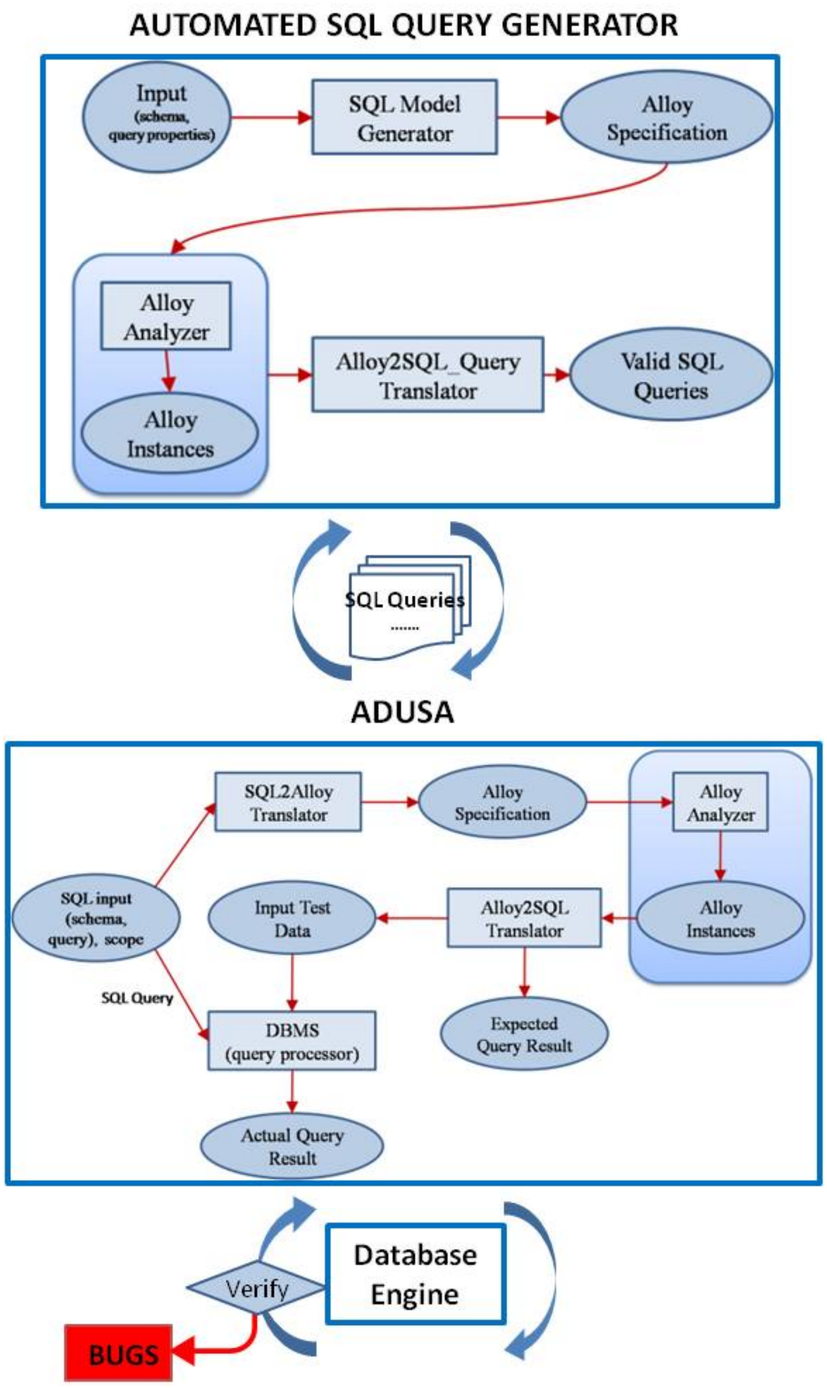


Figure 4.3: Full framework for DBMS testing.

   (a) The `SQL2Alloy` module which generates an Alloy specification by translating a given set of SQL statements representing a database schema and a test query.

   (b) The `Alloy2SQL` module which translates the Alloy instances into SQL statements that are used to populate the database on the DBMS under test.

3. The Alloy Analyzer module which generates instances that satisfy the translated Alloy specification. It used off-the-shelf SAT solvers to find solutions for the formulas generated by the Alloy specifications. Both previous two components use the Alloy Analyzer to find and enumerate solutions for the Alloy models generated.

4. The Verifier module which automates the process of loading the test data into the databases under test and comparing the DBMS with the Alloy query results. It reports any inconsistencies found and provide a report with the status of the database which is used for reproducing any errors.

5. The `FullTest` module which integrates all the modules together providing a benchmark of the queries tested and the ones remaining, plus the number of inconsistencies found and useful testing measurement information.

4.4 Test Summaries for SQL Query Generation

In this section, we present our approach for generating syntactically and semantically correct SQL queries as inputs for testing relational databases using test summaries. Similar to ADUSA, we leverage the SAT-based Alloy tool-set to reduce the problem of generating valid SQL queries into a SAT problem. Our approach translates SQL query constraints into Alloy models, which enable it to generate valid queries that cannot be automatically generated using conventional grammar-based generators.

Given a database schema and properties of SQL queries, which relates different sections of a query together, we automatically generate the corresponding predicates and constraints using Alloy. Then using the Alloy Analyzer, we generate solutions satisfying the constraints, those solutions are translated into concrete SQL queries which can be executed on the database schema.

4.4.1 Automated SQL Query Generator

In this section, we discuss the general algorithm for our approach. We describe the SQL grammar supported and the advances of using SAT in generating valid queries. We also describe the integration of this approach with our previous work on ADUSA, a SAT-based table generator, to automatically generate input queries and table data as well as expected query execution output.

```
QUERY ::= SELECT FROM WHERE GROUP_BY HAVING
SELECT ::= 'SELECT' selectTerm+
selectTerm :: term | aggregate(term)
FROM ::= 'FROM' (table | table JOIN table)
WHERE ::= 'WHERE' term operator (term | value)
GROUP_BY ::= 'GROUP BY' term
HAVING ::= 'HAVING' term operator value
aggregate ::= 'MAX' | 'MIN' | 'AVG' | 'COUNT'
operator ::= '<' | '<=' | '>' | '>=' | '='
```

Figure 4.4: SQL Grammar supported

4.4.2 SQL Grammar

In our approach, we consider a subset of SQL query grammar. The complete grammar supported by our approach is shown in Fig. 4.4.

Next, we describe how to write an Alloy specification to model a database schema and a subset of the SQL query grammar. Then use the Alloy tool-set, based on SAT, to generate syntactically and semantically valid queries. We describe the subset of Alloy language that we use – more details about Alloy can be found in [63, 104].

Alloy is a strongly typed specification language. It assumes a universe of atoms (or elements) partitioned into subsets, where every type is associated with a subset. An Alloy specification consists of a sequence of paragraphs where a paragraph enables defining new types, introducing relations between types, and adding constraints on these types and relations. Being an analyzable relational language, Alloy semantics are closely related to those of relational databases. This enables systematic modeling of relational databases, and automated analysis of relational operations on databases.

Alloy can be used to model a relational database schema. To illustrate, consider the example used in Section 4.2, the schema of the tables is shown in Fig. 4.1. Our approach generates an Alloy specification that represents both `students` and `grades` tables and each of their attributes. To systematically generate Alloy models for all tables, we model a general representation of tables and fields in Alloy as follows:

```
abstract sig FieldNames {}
abstract sig FieldTypes {}
abstract sig Field {
   name : one FieldNames,
   type : one FieldTypes
}
abstract sig TableNames {}
abstract sig Table {
   name : one TableNames,
   fields : some Field
}
```

We use signature paragraphs (`sig`) in Alloy to structure tables and fields of the database schema. For example, the `Table` signature introduces the general structure of a database table. Similarly, the `Field` signature introduces the structure of a Field. We also use signatures to declare new types of elements, in other words sets of elements which have similar use in our model. For example, we declare `TableNames` to store elements used as names of the tables, `FieldNames` and `FieldTypes` to store names and types of fields that are used in the schema.

In addition, Alloy allows us to declare relations between signatures. For example, `name` is a one-to-one relation between a `Table` element and a `TableNames` element. We use a one-to-one mapping because there must exist exactly one name for every table in the database schema. The one-to-one correspondence is set using the Alloy `one` keyword. On the other hand, we use one-to-many mapping between every `Table` element and `Field` elements. This is needed because every table can have more than one field declared for it. The Alloy `some` keyword is used to specify that the fields of a table is a set of minimum one field ensuring that we do not have a table with zero fields. Note that in Alloy, sets do not contain duplicates by definition, thus this guarantees that the set to which `fields` maps to does not contain any duplicate fields.

The `abstract` keyword before `sig` declarations identifies the signature as an empty set and creating elements of this set is constrained to signatures extending the set. In specific, we extend the `Table` and `Field` signatures by modeling unique signatures for every table and field used in the database schema.

We model aggregate functions by creating an abstract signature for all aggregates and extending this signature with the ones that we want to use. If we consider the aggregate functions `MIN` and `MAX` as in the grammar in our example, then the following Alloy code models these aggregates to be used in the query generation:

```
abstract sig AggregateNames {}
one sig MAX extends AggregateNames {}
one sig MIN extends AggregateNames {}
```

Modeling the SQL grammar in Alloy requires modeling each of the SELECT and FROM parts as separate entities. This guarantees the generation of syntactically correct queries. After modeling the query grammar in Alloy, we add constraints over the model which gives the ability to prune out queries which are either not useful or semantically incorrect. The following Alloy code models both the SELECT and FROM sections:

```
sig term {
   field : one Field,
   agg : lone AggregateNames
}
one sig SELECT {
   fields : some term
}
one sig FROM {
   tables: some Table
}
```

The `term` signature represents a term in the SELECT section. Each term element represents a field of a table, thus we create a one-to-one relation called `field` which maps each `term` with exactly one `Field` element. Since some aggregates are allowed to be used, we add a relation `agg` which relates the term with an aggregate. The `lone` keyword specifies that this relation is of degree zero or one, which makes the aggregate term optional. To model the complete SELECT section, we add another signature called SELECT. Since a SELECT statement constitutes of either one or more terms, we create a one to many relation called `fields` inside the `SELECT` signature. Note that the number of terms to be selected is not specified at this point. This constraint is added later on, giving the power to the user to specify the number of terms used in the SELECT section without changing the basic Alloy model. The keyword `one` before the signature `SELECT` forces the set to be a singleton set, meaning that there is only one SELECT statement in every query generated. This size constraint must be changed for generating nested SELECT queries. The `FROM` signature represents elements inside the FROM section. Similar to the `SELECT` signature, there is only one `FROM` element and this element has a one-to-many mapping with the set of tables in the schema. Later we add a constraint to specify what is the total number of tables that can be joined within the FROM section at once. We assume, for the simplicity of this example, that only cross JOIN is allowed between tables in the FROM section.

Our approach reads the database schema and automatically generates signatures and constraints for the tables and fields. First, we populate the `FieldNames` and `TableNames` with elements representing all the names of fields and tables that exist in the database schema. We use the `extends` keyword to extend any of the existing signatures in the model. We also use the multiplicity `lone` for these elements since they can be either singletons when used in a query or empty when not used. The following Alloy code covers all the field names and table names in our example that would be potentially used in queries generated:

```
lone sig id, name, studentID, courseID, grade
        extends FieldNames {}
one sig students, grades extends TableNames {}
```

We then automatically create a signature for each of the fields of the tables. These signatures extend the `Field` type. In addition, for each `Field` type extended we explicitly set the relation constraints, setting the name and type of every field explicitly. Similarly for each table in the schema, we create a signature extending the `Table` type. The code below shows the declarations of such signatures for both fields and tables of the schema. Note that for `fields` relation in the signatures extending table, we use the keyword `in` which guarantees that the field (on the left hand side) belongs to the set of fields for that table.

```
lone sig field_id extends Field {} {
   name = id
   type = intType
}
lone sig field_name extends Field {} {
   name = name
   type = stringType
}
...
lone sig table_student extends Table {} {
   name = students
   field_id in fields
   field_name in fields
}
lone sig table_grades extends Table {} {
   name = grades
   field_studentID in fields
   field_courseID in fields
   field_grade in fields
}
```

The operations used in the code above look similar to the ones we used in declaring the skeleton code for tables and fields. Nevertheless, the difference is significant. First, these signatures do not add any relations to the `Table` and `Field` types. Thus, the empty pair of braces `{}` is used at the end of the declaration. Second, we add Alloy facts to these signatures. Facts resemble explicit constraints that we want to satisfy. Thus we use another pair of braces directly following the signature declaration (facts in Alloy can be written as separate paragraphs as well, in this case we chose to embed them in the signature declarations for the ease of read).

After modeling the main components in Alloy, this guarantees that the SQL grammar is satisfied in the Alloy model. However, we need to add constraints to the model to guarantee semantically correct queries. Looking back at the SQL grammar in this example, and using a grammar based string generator, it would generate queries such as:

```
SELECT courseID from STUDENT;      (1)
SELECT name, name from STUDENT;    (2)
SELECT id from STUDENT, STUDENT;   (3)
```

These queries adhere to the syntax of the grammar in our example. They are syntactically correct but not semantically. Query (1) selects a field which does not belong to the table selected in the FROM section. It causes an error in execution when run on a most database systems. To prune out similar queries in our query generation we add Alloy constraints to the model. The constraint is added by introducing a new fact paragraph as follows:

```
fact field_in_table {
   all f: term.field | some t: FROM.tables |
 f in t.fields
}
```

The above Alloy fact paragraph reads as: *for any of the fields of term elements, there exists a table t in FROM tables, where the field belongs to t's fields.* The `all` quantifier stands for universal quantification, the `some` quantifier stands for existential quantification. The dot operation `'.'` represents a relational join, e.g., `FROM.tables` returns the domain of the `tables` relation and the `t.fields` returns the image of an element `t` in the `fields` relation.

Query (2) is semantically correct. It would run and execute on any database system, however we would prefer selecting different fields in the SELECT section to get more meaningful queries for testing. We add a new fact to the model to enforce this property:

```
fact unique_select_terms {
   all a, b : SELECT.fields.term |
  (a.field = b.field and a.agg = b.agg ) => a=b
}
```

The above Alloy fact paragraph reads as follows: *for any two elements in the terms set of SELECT.fields image, if both are related to the same field element and both have the same aggregate element, this implies those two elements are identical.* The `'='` operator in Alloy is not an assignment operator, it is a boolean operator which checks if the elements on the right and left hand sides are the same. The `'and'` operator is a logical conjunction which evaluates to true if and only if both sides of the equation are true. The `'=>'` operator is the implication operator as used in mathematical logic operators. Thus we specify in this constraint that if any two fields belonging to the set of selected fields have the same table field and aggregate, then make these selected fields unique, guaranteeing that the SELECT section of the query does not have duplicate terms.

Similar to query (2), query (3) is syntactically and semantically correct, but seldom is the case that queries require joining the same table to itself. Adding a constraint to not join the table with itself in the grammar is complex and could be done manually by enumerating all possible combinations of tables to be cross joined and explicitly mentioning those in the grammar. In our example, since we only have two tables to pick from, we can update the grammar easily to not chose the same table twice. Anyway, in a more general case where we have more tables to pick from, enumerating possible table joins is tedious and prone to mistakes. In our approach, we add a constraint to the model to guarantee not to pick same table twice. Similar to specifying unique SELECT terms, we specify this fact as: *for any two tables in the tables image of the FROM element, if those tables map to the same table element then they must be identical* :

```
fact unique_tables {
   all a, b: FROM.tables |
   a.table = b.table => a = b
}
```

4.4.3 SQL GROUP BY Constraints

Modeling the GROUP BY section requires declaring a new signature paragraph in Alloy. We add a one-to-many relation to the GROUPBY signature named `fields` which maps to a set of `term` elements. The multiplicity of the `fields` relation is at least one which is enforced by the Alloy `some` keyword. The following Alloy code models the GROUPBY section of the query:

```
lone sig GROUPBY{
   fields : some term
}{
   fields in SELECT.fields
   #fields.agg = 0
}
```

The main constraint in the GROUP BY section is that the terms used within it should be a subset of the terms used within the SELECT section, not considering the ones used with an aggregate function. Adding this constraint to a conventional grammar based generator is complex. One way to do so, is by marking each term in the SELECT section with a unique alias (using SQL 'as' keyword), and then having the terms in the GROUP BY section group be selected out of those alias names. In our approach, we add a fact to the GROUPBY signature to indicate that the terms in the `fields` mapping belong to the terms in the `SELECT` fields. Then we add another constraint forcing the number of aggregate function in the GROUPBY terms to be exactly zero, guaranteeing that the terms are out of the ones which are not aggregated. For example, query (1) is valid while query (2) is not because `NAME` is not selected in the SELECT clause without an aggregate fun:

```
SELECT ID,MAX(NAME) FROM STUDENT GROUP BY ID;(1)
SELECT ID,MAX(NAME) FROM STUDENT GROUP BY NAME;(2)
```

4.4.4 SQL HAVING Constraints

We introduce two new signatures for the HAVING section: `HAVING` and `havingTerm`. The `havingTerm` represents the grammar "`aggregate_function(term) operator value`". Each of the `havingTerm` elements is related to one `term` element, one `Operator` element, and one `Value` element. The `HAVING` signature contains the `fields` relation mapping it to a non-empty set of `havingTerm` elements. The following Alloy code models the HAVING section of the query:

```
abstract sig Operator{}
abstract sig Value{}
sig havingTerm{
    field : one term,
    operator : one Operator,
    value : one Value
}
lone sig HAVING{
   fields: some havingTerm
}{
   fields.field in SELECT.fields
   all f: fields.field | #f.agg = 1
}
```

The constraint of HAVING clause is that the terms used should be a subset of the terms used in the SELECT queries with an aggregation function. The fact following the `HAVING` signature declaration ensures two properties: (1) all the terms in the HAVING clause are subset of the terms used in the SELECT clause and (2) for any term in the HAVING clause, the term is used with an aggregation function. For example, query (1) is valid while query (2) is not because `MAX (NAME)` is not selected in the SELECT clause:

```
SELECT NAME, MAX (ID) FROM STUDENT GROUP BY
    NAME HAVING MAX (ID) > 5;            (1)
SELECT NAME, MAX (ID) FROM STUDENT GROUP BY
    NAME HAVING MAX (NAME) > 5;          (2)
```

4.4.5 SQL WHERE Constraints

We introduce two new signatures for the WHERE clause: `whereTerm` and `WHERE`. The `whereTerm` elements represent a clause containing a term compared to another term or another constant value. The `operator` relation maps the `whereTerm` element to a single operator. The `WHERE` signature maps the where clause to a set of `whereTerm` elements. We discuss the relation between the where terms in the discussion section. The following code models the SQL WHERE clause and their constraints in Alloy:

```
sig whereTerm {
    leftTerm : one term,
    operator : one Operator,
    rightTerm : one term + Value
}{
    leftTerm != rightTerm
}
lone sig WHERE {
    fields: some whereTerm
}
fact whereTerms {
   all n:
    (WHERE.fields.leftTerm +
    WHERE.fields.rightTerm).field
    | some t: FROM.tables | n in t.fields
   all a, b: whereTerm | a.leftTerm =
    b.leftTerm and a.rightTerm = b.rightTerm
    and a.operator = b.operator => a = b
}
```

We constrain comparing the same element with itself using the fact '`leftTerm != rightTerm`'. We constrain that the terms in the WHERE clause belong to one of the tables in the FROM clause inside the `whereTerms` fact paragraph. The fact indicates that for any field in the either left or right side terms of the where term, there exists a table in the FROM section which the field belongs to. Another fact statement prunes out have same where terms by specifying that for any two where terms, if they have the same fields on the left and right hand side and the same operator, then these two fields are identical.

4.4.6 Alloy2SQL_Query Translator

The Alloy Analyzer tool-set compiles an Alloy model into a boolean formula and uses SAT technology to solve it. It iterates over all possible solutions for the Boolean formula. For every solution, it converts it into an Alloy instance. In our approach, we take the Alloy instances and convert them into a valid SQL queries that are used for testing databases and applications.

Fig. 4.5 shows a graphical representation of a SAT solution of a SQL model mapped into Alloy elements. The graphical image is produced by the Magic lay-out available in the Alloy Analyzer tool-set. The instance is concretized into: SELECT MAX (NAME) FROM STUDENT;.

An Alloy instance generated by the Alloy analyzer is a set of valuations

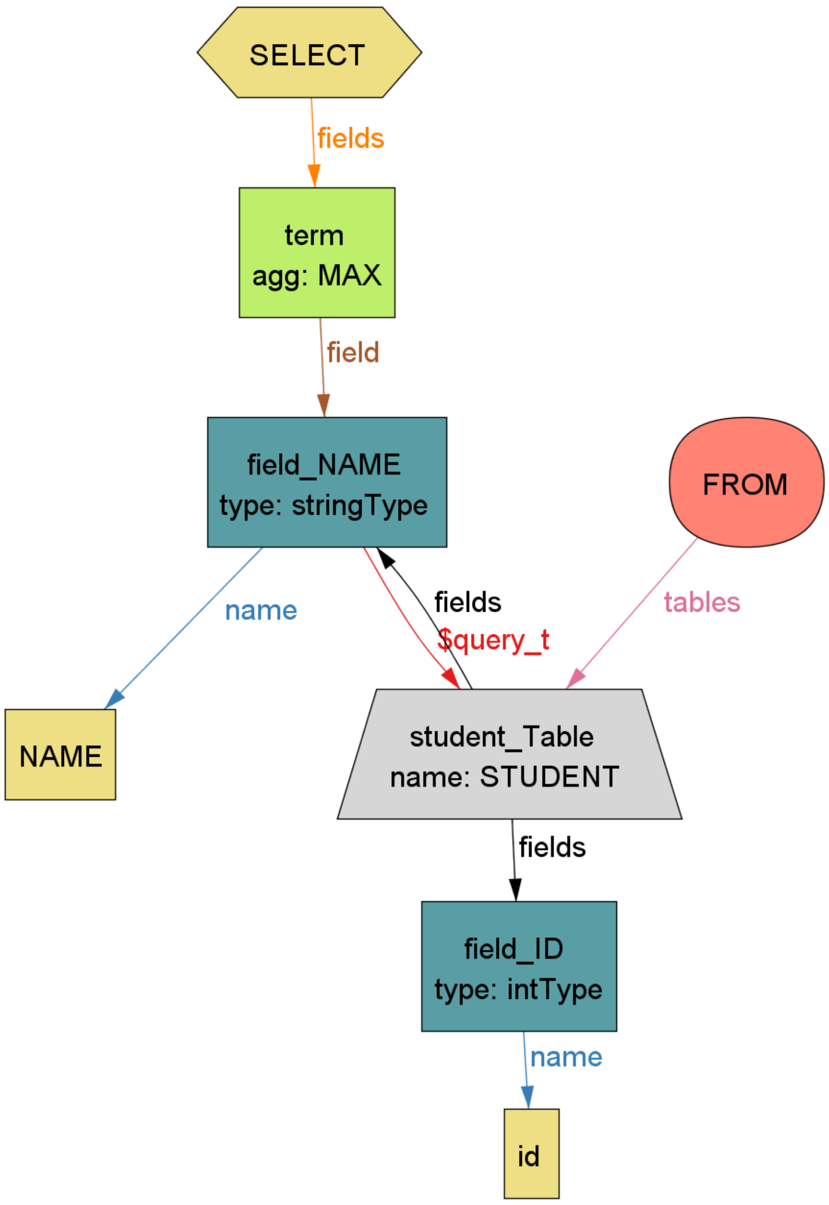


Figure 4.5: Sample Alloy output.

```
for all sig : Instance.Signatures
   for each valuation of sig in the solution
      Create a Java object for the valuation
      //map the Alloy object to Java object
      map.add (AlloyObject, JavaObject)
for all sig : Instance.Signatures
   for all field : sig.fields
      //need a loop for non-singleton field
      for every valuation of field
         sourceObject = map.get (field.source)
         targetObject = map.get (field.target)
         sourceObject.setField(field.name,
               targetObject)
print map.values()
```

Figure 4.6: Algorithm for translating an Alloy instance into a SQL query.

assigned to the signatures and relations declared by the Alloy model. To translate an instance into a SQL query, we first identify the signatures associated with the fields, tables, aggregate functions and other singleton elements which serve as the initial parts of the grammar. Then we iterate over relations on the signatures setting the corresponding field values for every relation. We use Java classes to represent each part of the SQL grammar. Then we use a mapping from Alloy objects into Java objects. We set the relations between signatures by calling setter methods in the Java classes corresponding to the field being set. Fig. 4.6 illustrates the basic algorithm for creating a concrete SQL query out of an Alloy instance.

4.4.7 Case Studies

In this section we discuss the use of our framework in different case studies. We perform tasks for generating SQL queries based on different subsets of the SQL grammar.

In each case study we use our approach to enumerate all possible valid queries for a given schema. We compare the approach by applying it to different subsets of the SQL grammar. We consider the same schema presented in Fig. 4.1 consisting of the two tables: `student` and `grades`. The 3 subsets of SQL grammar that we consider are presented in Table 4.1. Case#1 consists of only SELECT and FROM clauses. For each of the tests, we consider two cases: (1) up to one table in the FROM section, and (2) up to two tables in the FROM section. So, queries generated in (1) are inclusive to (2).

Table 4.2 shows the solving time of each of the case studies. The #Tables is the maximum number of tables in the FROM section. Primary variables and total variables are the Alloy variables used in generating the Boolean formula. The clauses are the Boolean clauses. Solving time is the SAT time to generate the first possible solution for the Boolean formula (next solutions take negligible time). Concretization time per query, is the processing time our approach does to concretize an Alloy instance into a SQL query in ms. The #Queries is the total number of queries generated for the specific case study.

The total number of queries increases drastically for Case#2, this is

Case#	SQL Grammar

1	QUERY ::= SELECT FROM
	SELECT ::= 'SELECT' selectTerm+
	FROM ::= 'FROM' (table | table JOIN table)

2	QUERY ::= SELECT FROM WHERE
	SELECT ::= 'SELECT' selectTerm+
	FROM ::= 'FROM' (table | table JOIN table)
	WHERE ::= 'WHERE' term operator (term | value)

3	QUERY ::= SELECT FROM GROUP_BY HAVING SELECT ::= 'SELECT' selectTerm+ FROM ::= 'FROM' (table | table JOIN table) GROUP_BY ::= 'GROUP BY' term HAVING ::= 'HAVING' term operator value
*	selectTerm ::= term | agg (term) term ::= 'id' | 'name' | 'studentID' | 'courseID' | 'grade' agg ::= 'MAX' | 'MIN' table ::= 'students' | 'grades'

Table 4.1: The SQL grammar used in each case study. (The * indicates common terminal values.)

Case#	#Tables	Primary Vars	Total Vars	Clauses	Solving Time(ms)	Concret. Time(ms)/Query	#Queries
1	1	155	1586	2648	375	4.03	66
1	2	155	1586	2652	343	3.53	186
2	1	247	3013	5135	390	3.86	3456
2	2	247	3011	5129	422	5.13	27081
3	1	263	3209	5524	437	4.61	26
3	2	263	3210	5528	422	4.08	76

Table 4.2: Solving time of each of the case studies.

because the fact that the WHERE clause can contain terms from the tables which are not constrained by the SELECT statement, and the fact that each term can be related to either another term or a value, thus the number of possible queries increases. Case#3, using up to one table in the FROM section, generates the minimum amount of queries. This is because both constraints for GROUP BY and HAVING must be satisfied in all the queries generated. In the grammar for Case#3, the GROUP BY and HAVING clauses are mandatory, thus limiting the output space.

4.5 Test Summaries for Database Test Generation

We present a novel approach for effective black-box DBMS testing based on our work in the master's thesis [1]. Our approach uses model-based testing to perform (1) *query-aware* database generation to construct a useful test input suite that covers the various scenarios for query execution and (2) test oracle generation to verify query execution results on the generated databases. As an enabling technology, we use Alloy and the Alloy Analyzer. Alloy's relational basis provides a natural fit for modeling relational databases and query operations. Given an input database schema and an input SQL query (manually written by the user, or automatically generated by our automated SQL query generator), our approach formulates Alloy specifications to model both of the inputs, and then uses the Alloy Analyzer to generate all databases that satisfy the given schema under a ceratin bound, as well as the expected result of executing the given query on each of the generated databases. Each input/oracle pair is then used to test the query execution on the DBMS.

Our framework automates both test input and test oracle generation. By incorporating both the schema and the query during the analysis, our approach performs query-aware data generation where executing the query on the generated data produces meaningful non-empty results. We evaluated our approach by testing the query execution correctness of three DBMSs including two open source systems, HSQLDB [102] and MySql [98], and one commercial DBMS, Oracle 11g [88]. Experimental results show that ADUSAwas able to detect (1) a non-reported bug in Oracle 11g, (2) bugs that are previously reported in the MySql bug repository, and (3) bugs that we injected in HSQLDB.

The following sections show an example illustrating the testing approach and the effectiveness of using Alloy for both test input and oracle generation.

4.5.1 Example

We describe how to generate an Alloy specification to model a database schema and a test query, and use the Alloy specification to test a DBMS. Alloy can be used to model a relational database schema. To illustrate, consider the student table described in the database schema in Fig. 4.1. We systematically generate an Alloy specification to model the schema as follows:

```
sig varchar{}
one sig student {
    rows: Int -> varchar
}
```

Alloy specifications can also be used to model operations on relational databases. For example, consider the following SQL query:

```
SELECT DISTINCT id
FROM student
WHERE (id=1 OR (id>=3 AND id<=5));
```

The above query is a selection SQL statement that queries the `student` relation, and returns all the `id` elements that are either equal to 1, or between 3 and 5. The following Alloy paragraphs model the SQL query.

```
fun query () : Int {
    {select[where[from[student.rows]]]}}
fun select (rows: Int -> varchar) : Int {
    {rows.varchar}}
fun where(rows: Int -> varchar) : Int -> varchar {
    {x1: rows.varchar, x2: x1.rows | condition[x1]}}
pred condition(x1: Int) {
    {eq[x1, 1] or (gte[x1, 3] and lte[x1, 5] )}}
fun from(rows: Int -> varchar) : Int -> varchar {
    {rows}}
```

After generating the Alloy specification for the schema and the query, ADUSA uses the Alloy Analyzer to find database instances that satisfy the specification, i.e., provide valuations for the types and relations that satisfy all the constraints. The Alloy Analyzer finds all the instances within a given scope, i.e., a bound on the number of data elements of each type to be considered while generation. Below is an example of an instance generated by the Alloy Analyzer:

```
varchar: {varchar$0, varchar$1}
student:{student$0}
```

```
rows: {(1, varchar$1), (2, varchar$0), (4, varchar$1)}
$query: {1, 4}
```

In the above instance, the `varchar` set has two elements labeled as `varchar` followed by the element number. The rows relation consists of three tuples whereby two tuples satisfy the condition of the `condition` predicate. The `query` set represents the result of executing the query on the `rows` relation. The `query` set holds the `id` attribute of the tuples that satisfy the `condition` predicate. For this example, the result is the set of integers {`1, 4`}. After generating the Alloy Instance, ADUSA translates the instance into `INSERT` SQL queries that are used to populate an empty database. For example, for the above Alloy instance, ADUSA identifies the `rows` relation and generates the following `SQL` statements:

```
INSERT INTO student VALUES (1, varchar$1)
INSERT INTO student VALUES (2, varchar$0)
INSERT INTO student VALUES (4, varchar$1)
```

Once the database is populated with data, the given SQL selection query is executed on the DBMS and the result is verified with the one found in the Alloy instance. This process is repeated for each generated instance.

4.5.2 Query-aware Test Generation

By incorporating information from the SQL query and the schema, our approach performs effective query-aware generation where the generated test cases are guaranteed to produce meaningful results upon query execution as opposed to query unaware generators where the execution results are highly likely to be empty.

To illustrate the effectiveness of the query-aware generator in ADUSA, we used it to generate a set of databases for testing the following query:

```
SELECT *
FROM student
WHERE id < 3 and name = 'John'
```

Figure 4.7 shows an example of 3 instances generated by ADUSA. The integer values generated by ADUSA are closely related to the predicate (`id < 3`) described in the query. Based on the given predicate, ADUSA partitions the integer space into two regions: integers with values less than three and integers with values greater than or equal to three. ADUSA then generates tuples with integer values from each of the regions, as well as tuples with integer values from both regions. The same approach is performed for the string types. ADUSA generates strings that are either equal to "John" or not. The generated instances are then constructed using values from these spaces. For example, the first instance from Figure 4.7 only contains integers with values less than three, and strings with values equal or unequal to "John"; the second instance only contains integers with values greater than or equal to three; and the third instance contains both.

Executing the query on the generated input results in one tuple for the first instance, no tuples for the second, and two tuples for the third instance.

```
Instance 1 - rows: {(0, varchar), (1, John)}
Instance 2 - rows: {(4, John), (6, varchar), (3, varchar)}
Instance 3 - rows: {(0, John), (3, varchar), (2, John)}
```

Figure 4.7: Alloy instances for a database table generated using ADUSA.

The use of the Alloy Analyzer enables enumerating different instances that cover the spaces for the data types involved in the query constraints.

4.5.3 Test Oracle Generation

Almost all academic and commercial test data generators don't provide a mechanism for verifying the result of a query execution on the generated data. An obvious approach to do this is by using a trusted (golden) DBMS as an oracle. This process includes running the query on both the test and the golden DBMSs and matching the results to assure correctness. Using this approach requires the existence of a more advanced DBMS that supports the features under test as well as the correct implementation of those features.

A key advantage of using Alloy as a modeling language is the ability to generate test oracles. While performing data generation, the Alloy Analyzer automatically labels a subset of the generated data as the query result. Such feature provides an automatic and efficient way to verify the execution result of the DBMS under test. A more subtle advantage is that we can even specify constraints on query results, and generate input databases that satisfy these constraints, e.g., cardinality constraints on the results' size.

4.6 Experiments

In this section we discuss the results of experimenting our full framework with Oracle11g database management system. We perform tasks for generating SQL queries based on different subsets of the SQL grammar and automatically verify the output of the DBMS. Our experiments successfully identified new queries which result in erroneous output.

We describe the results of full tests which illustrate the functionality of the framework while testing Oracle11g (Release 1) [88]. The first set of tests shows the performance and complexity of the Alloy models as the number of selected tables increases. The second set of tests focuses on queries which revealed bugs in the Oracle DBMS. Our previous work on ADUSA was successfully able to reproduce and provide a counter example for a bug in Oracle11g. Nevertheless, this bug was only reproduced using a specific SQL query. Using our automated framework, we were able to identify 5 new queries which reveal erroneous output in Oracle11g which we were not previously aware of, and the required data to reproduce each of them.

4.6.1 Generating full tests

First we demonstrate the effectiveness of our framework in providing a complete test cycle for testing databases. Our framework generates valid SQL queries for test and each query is modeled in Alloy in addition to the database schema. Then it produces test inputs to populate the database and the expected output of running the query on each test input.

In the following tests, we consider a database schema constituting of 3 tables: (1) `student` table with `ID` (primary key) and `Name` fields, (2) `course` table with `cID` (primary key) and `Name` fields, and (3) `department` with `ID` (primary key) and `Name` fields. We consider all fields of the tables to be of varchar type. Given this schema, we consider automatically generating all valid SQL queries satisfying the following grammar:

```
QUERY ::= SELECT FROM
SELECT ::= 'SELECT' selectTerm+
selectTerm :: term | aggregate(term)
FROM ::= 'FROM' (table |
  table NATURAL JOIN table |
  table NATURAL JOIN table NATURAL JOIN TABLE)
aggregate ::= 'COUNT'
term ::= 'ID' | 'NAME' | 'CID' | *
```

We add to the constraints while generating SQL queries that no table is selected twice in the FROM clause, this guarantees that a table is not crossed joined by itself. In addition, we do not have to distinguish fields with same names between tables since NATURAL JOIN joins tables based on common fields names[3]. In addition, we add a specific constraint for the `COUNT` aggregate function; we specify that there are no other terms selected if `COUNT` is used. Another constraint added is that if the $\star$ term is selected then no other terms can be selected at the same time to guarantee a correct SQL query syntax.

[3]Our approach supports tables with same field names by creating alias names for the fields of the tables selected, thus every field is uniquely represented as table_name.field_name.

Using the schema and subset SQL grammar discussed above, our automated query generator produces 57 unique queries. To show the results, due to lack of space, we group the queries into 7 groups. Each group is identified by the tables in the FROM clause. We show the average time needed to generate a database instance and verify the correctness of execution by comparing it to the oracle (expected result). Table 4.3 shows the details of the test suites execution. The tests were run on an Intel Core 2 Duo 2.0GHz, 2GB RAM with Java Runtime Environment 1.6.0.

Table 4.3 shows the details of testing Oracle11g with sets of queries; for each query generated, ADUSA enumerates all test cases and checks the DBMS output for correctness. The FROM Clause Set identifies the queries by the tables in the FROM clause. The `'x'` identifies NATURAL JOIN between tables; S, D, and C were used as abbreviations of table names for simplicity. #Queries is the total number of queries within the FROM Clause set. Total #DB tests is the total number of database instances which ADUSA generated for testing the database. Avg. #tests query is the average number of database instances generated for each query by ADUSA. Primary variables and clauses are the average variables and clauses in the Alloy model generated by ADUSA to produce a test. Average time per query is the average time ADUSA takes per query to enumerate all possible database instances, delete data from the database for every instance, populate the database with new database instance data, query the database, and verify the correctness of the DBMS output. Total time is the total time consumed by ADUSA to verify the correctness of

FROM Clause Set	#Queries	Total #DB tests	Avg. #tests/ Query	Prim. Vars	Clauses	Avg. Time(ms)/ Query	Total time(sec)	#Failures
STUDENT (S)	6	3036	506	30	468	4072.8	24.4	0
COURSE (C)	6	3036	506	30	468	4312.5	25.8	0
DEPARTMENT (D)	6	3036	506	30	468	4221.2	25.3	0
S x D	6	5096	850	39	1252	7721.5	46.3	0
S x C	11	11000	1000	44	1531	9642.1	106.1	0
D x C	11	11000	1000	40	1486	8836.7	97.2	0
S x C x D	11	11000	1000	48	3485	12670	139.4	988

Table 4.3: Results of running our framework on Oracle 11g.

all the queries in a given FROM Clause set. The scope of variables used by ADUSA in all the tests above is 3 varchar and 4 bits for Int.

We set a threshold of 1,000 database instances to test per query. In most cases, in less than 10 seconds ADUSA was able to verify 1,000 test cases for correctness for each query. It is worth mentioning that for each test case there is an overhead time consumed in emptying the database, inserting new elements into it, and finally querying it. The time required for concretizing the oracle (expected output) from an Alloy instance into real data and comparing it to the actual DBMS output is negligible. For each query, we establish a database connection once and close it after all instances generated by ADUSA for that query has been tested. Note that in Table 4.3 the average time (ms) per query is the time elapsed to test all instances generated for one query. In almost cases, the number of database instances is more than 500.

In addition to testing Oracle11g, we ran the same tests on MySQL 5.0. The most noticeable difference is the time consumed for running the tests. It took in almost all cases, half the time to run the tests on MySQL compared to Oracle11g. Which points out that the overhead of database connection and querying is a bottle neck.

4.6.2 Experimenting with Oracle11g

Table 4.3 shows that out of the total 11,000 tests on the FROM set which includes `student NATURAL JOIN course NATURAL JOIN department`, there were 988 tests which revealed failures. We experiment more on these queries showing bugs. We examine the relevance of the scope of variables to finding a bug faster verses complexity of the Alloy model. Increasing the scope used by ADUSA can significantly blow up the number of database instances generated. On the other hand, having a richer set of test cases may reveal bugs faster.

The set of test queries with the NATURAL JOIN of three tables is as follows:

```
SELECT DISTINCT id FROM ...
SELECT DISTINCT id,NAME FROM ...
SELECT DISTINCT id, NAME, cID FROM ...
SELECT DISTINCT NAME FROM ...
SELECT DISTINCT NAME,cID FROM ...
SELECT DISTINCT cID FROM ...
SELECT DISTINCT id, cID FROM ...
SELECT COUNT (DISTINCT cID) FROM ...
SELECT COUNT (DISTINCT id) FROM ...
SELECT COUNT (DISTINCT NAME) FROM ...
SELECT COUNT (*) FROM ...
```

All these queries select from `student NATURAL JOIN course NATURAL JOIN department`. They constitute all possible valid queries adhering to the constraints and grammar we specified for these tests. Using ADUSA framework to test these queries on Oracle11g we were able to identify bugs in 6 queries.

Table 4.4 shows the queries which revealed unexpected wrong output. The scope is the maximum number of varchar elements used to populate the database with test instances. #DB tests is the number of database test instances, generated by ADUSA, considered for each query. #Failures is the

#	Query	scope	#DB tests	Total time(ms)	#Bugs
1	SELECT DISTINCT NAME,cID FROM ...	3	1000	13500	229
2	SELECT COUNT (DISTINCT cID) FROM ...	3	1000	11703	229
3	SELECT COUNT (DISTINCT NAME) FROM ...	3	1000	11593	228
4	SELECT DISTINCT NAME FROM ...	3	3798	75812	1
		4	2148	36500	1
5	SELECT DISTINCT cID FROM ...	3	2721	32828	1
		4	1000	17141	1
		4	2000	33265	139
6	SELECT COUNT (*) FROM ...	3	1000	11938	302
		3	4000	47984	1139
		4	4000	80188	2414

Table 4.4: Queries which revealed bugs in Oracle11g.

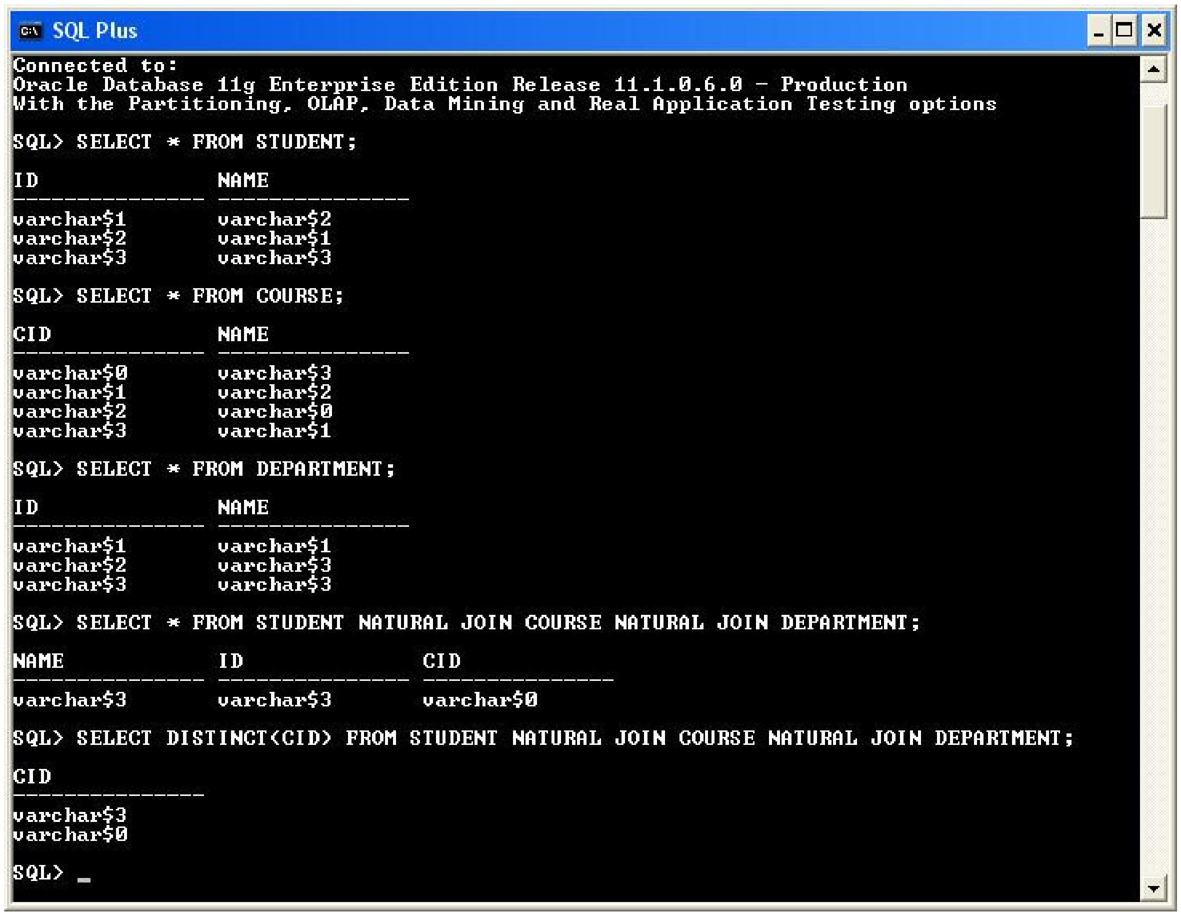
number of tests which produced an inconsistence result with the expected output. The first 5 queries were newly discovered using our approach. The 6th query was discovered in our previous work on ADUSA. To verify that a bug exists, we used two methods. First, for every failure detected, ADUSA provides us with efficient data needed to reproduced it. We use the same data to test the query on a different database, for instance MySQL 5.0, which has the same database schema saved on it as well. For all the queries which revealed failures, none of them showed any inconsistencies with MySQL 5.0. Second, we ran random manual verifications. For random counter examples, we manually analyzed the data, predicted the output and verified our analysis. Indeed, our manual verification results were consistent with ADUSA's.

Table 4.4 shows the number of database instances generated and corresponding inconsistencies found. An interesting query is Query#4, where ADUSA was able to find an inconsistency after enumerating 3,798 instances with scope of exactly 3 varchar; and it took 2,148 instances to reproduce the bug with a scope of 4 varchar. This indicates that a tester can easily come up with a test suite out of the thousands of possible tests and not detect the bug. Trying to manually generate data to reproduce the bug would have taken hours, while exhaustive bounded checking can guarantee that up to a given scope all possible instances have been verified. Nevertheless, this keeps the possibility of finding a bug with bigger scopes.

We compare running the tests with scope of 3 versus 4 for the last 3 queries in Table 4.4. The results show that using a bigger scope there was a higher possible of detecting the bug faster. For example, tests with scope of 4 on Query#5 showed only 1 inconsistency in the first 1,000 tests, but it revealed 138 more inconsistencies in the next 1,000 tests. While using scope of 3, the first inconsistency was detected after 2,721 tests. On the other hand, using a bigger scope puts a load on the clauses generated by Alloy and running the tests with bigger scopes is a little slower, but compared to the database overhead the time to run the tests with higher scopes is negligible. The time consumed increased significantly in most cases because of the blow up in possible database instances to generate. Originally we used 1,000 instances as a threshold to stop ADUSA from generating more instances. As our experience in finding erroneous queries increase, we think that a higher threshold might be needed for higher confidence in the results.

Fig. 4.8 shows a screen shot of the data produced by ADUSA to generate a counter example for Query#5 of Table 4.4 on Oracle11g. The expected output of the query is 'varchar$0' while Oracle11g reports 'varchar$3' and 'varchar$0' as output which is wrong. It is worth mentioning that without using our framework, we were not aware of any bugs found in the first 5 queries of Table 4.4 and it would have been hard to find and reproduce them.

Previously, having known that Query#6 produces a bug from our previous work, we suspected a bug in the implementation of the `COUNT` aggregate function, but using our full framework for automating tests enabled us to detect new queries producing erroneous output. The queries show that the bug is not related to the aggregate function and it could be that multiple bugs are



```
Connected to:
Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> SELECT * FROM STUDENT;

ID              NAME
--------------- ---------------
varchar$1       varchar$2
varchar$2       varchar$1
varchar$3       varchar$3

SQL> SELECT * FROM COURSE;

CID             NAME
--------------- ---------------
varchar$0       varchar$3
varchar$1       varchar$2
varchar$2       varchar$0
varchar$3       varchar$1

SQL> SELECT * FROM DEPARTMENT;

ID              NAME
--------------- ---------------
varchar$1       varchar$1
varchar$2       varchar$3
varchar$3       varchar$3

SQL> SELECT * FROM STUDENT NATURAL JOIN COURSE NATURAL JOIN DEPARTMENT;

NAME            ID              CID
--------------- --------------- ---------------
varchar$3       varchar$3       varchar$0

SQL> SELECT DISTINCT(CID) FROM STUDENT NATURAL JOIN COURSE NATURAL JOIN DEPARTMENT;

CID
---------------
varchar$3
varchar$0

SQL> _
```

Figure 4.8: Snapshot of an erroneous query output in Oracle11g detected by our framework. The counter example is automatically produced. The correct query execution should return 'varchar$0'.

related in these queries.

Chapter 5

Clustered Test Execution

In this Chapter, we present a novel technique for efficiently executing suites of unit tests, where several tests in a suite may contain a common initial segment of execution – a property often exhibited by systematically generated suites, e.g., those for bounded exhaustive testing [18, 51, 71, 89]. Our insight is that we can *cluster* execution of such tests by defining *abstract-level undo* operations [41, 91], which allow a common execution segment to be performed once, and its result to be shared across the tests, which then perform the rest of their operations. Thus, unlike existing techniques for test execution, distinct tests that execute different program paths to explore different behaviors do not have to be executed separately (one by one), and their common (initial) operations do not have to be performed repeatedly (once for each test).

5.1 Overview

During the last decade, much progress has been made to address these practical shortcomings of testing. Frameworks, such as JUnit [13], allow automated test execution and error reporting, and have become an integral part of development environments. Novel techniques, such as TestEra [71], Korat [18], DART [51], and RANDOOP [89], allow automated generation of test inputs – a largely manual task in traditional testing – and have provided the basis for systematic approaches, e.g., *bounded exhaustive* testing, where a program is checked against all inputs within a given bound on input size. While such bounded exhaustive approaches enhance our ability to find bugs in real programs [82], they require executing a large number of tests, which can be a time consuming and expensive task, especially for tests that execute operations on external resources, such as a file system or a network.

We present our framework for clustered execution of unit tests for Java programs. Given a set of tests, our framework analyzes the tests to build a *cluster tree* – a trie [99] structure that represents the sharing of initial code segments across the tests. Each node represents a code segment in a test. Thus, each path in the cluster tree represents a test case. A depth-first traversal of the cluster-tree forms the basis of test execution: visiting a node represents executing the corresponding code segment. The search backtracks by performing user-defined abstract-level undo operations, which enable sharing of results for initial segments of execution common to different tests.

Abstract-level undo operations provide a powerful mechanism for backtracking, which enables clustered test execution. While backtracking is a primitive idiom in model checking, general purpose model checkers, such as Java PathFinder [108] and SpecExplorer [28], do not directly support clustered test execution, since either they need to store and retrieve entire program states, which is not feasible for programs that perform operations on external resources, or they re-execute from start, which amounts to running the tests one by one. Thus, backtracking operations on external resources necessitates the usage of undo operations on those resources.

5.2 Example

To illustrate our approach, we present an example of using checkpoint-based clustering of unit tests for database management systems. We consider black-box database tests. We first describe an example database schema and present several corresponding unit tests. Then we describe how our approach clusters those tests using check-point undo operations on the database system.

Let us consider the following SQL statement for describing our database schema containing one table:

```
CREATE TABLE student (
  s_id int,
  s_name varchar(50),
  PRIMARY KEY (s_id)
);
```

The above SQL statement creates a `student` table with two attributes, `s_id` of type `int`, and `s_name` of type `varchar`. The statement also specifies the `PRIMARY KEY` property on the `s_id` attribute which restricts the number of tuples in the `student` table with the same `s_id` attribute to one. That is, every student entry in the table has a unique id.

Based on the above schema, let us consider the four unit tests in Figure 5.1. Each test initially calls a `emptyTables()` method which empties the database from any data in the tables. We call this method to make sure that each test runs on an empty database in order to verify the test results. Then each test inserts test data into the database and runs a SQL SELECT query on the database and asserts some properties of the result.

As the four tests show, the initial six statements are common between them all. Commonly a *setUp()* method, available in JUnit framework [13], is used to encapsulate those statements in one method and run it prior to every test. Nevertheless, *setUp()* methods should be manually written and do not provide any performance improvement since it is run for each test.

Even though the four tests share some initial execution trace, traditional state-based checkpointing, such as recording the program state at the point where execution paths differ between two tests, cannot be applied in this scenario. For example, a state-full model checker stores the program state at line 7 then retrieves it for each test instead of executing the same code again. However, in our example, doing so results in erroneous output since the state of the database is not stored in the Java heap space and cannot be retrieved by a simple storage of the program state.

Our approach overcomes this limitation by introducing abstract undo operations for clustering tests. The programmer defines undo operations at the level of method calls which restore the state of the heap or external memory, such as a database. The undo operations are meant to be defined at the

```
@Test public void test_1 {
  emptyTables();
  StudentDBManager db = getDBManager();
  db.addToStudent(1, "name1");
  db.addToStudent(2, "name2");
  db.addToStudent(3, "name3");
  db.addToStudent(4, "name4");
  ResultSet rs = db.executeQuery("SELECT COUNT(*) FROM student");
  assertTrue(rs.getFirstResult().equals("4"));
}
```

```
@Test public void test_2 {
  emptyTables();
  StudentDBManager db = getDBManager();
  db.addToStudent(1, "name1");
  db.addToStudent(2, "name2");
  db.addToStudent(3, "name3");
  db.addToStudent(4, "name4");
  db.deleteStudentByID(1);
  ResultSet rs = db.executeQuery("SELECT COUNT(*) FROM student");
  assertTrue(rs.getFirstResult().equals("3"));
}
```

```
@Test public void test_3 {
  emptyTables();
  StudentDBManager db = getDBManager();
  db.addToStudent(1, "name1");
  db.addToStudent(2, "name2");
  db.addToStudent(3, "name3");
  db.addToStudent(4, "name4");
  db.deleteStudentByID(4);
  db.deleteStudentByID(3);
  ResultSet rs = db.executeQuery("SELECT MIN(s_id) FROM student");
  assertTrue(rs.getFirstResult().equals("1"));
}
```

```
@Test public void test_4 {
  emptyTables();
  StudentDBManager db = getDBManager();
  db.addToStudent(1, "name1");
  db.addToStudent(2, "name2");
  db.addToStudent(3, "name3");
  db.addToStudent(4, "name4");
  db.deleteStudentByID(4);
  db.deleteStudentByID(1);
  ResultSet rs = db.executeQuery("SELECT MAX(s_id) FROM student");
  assertTrue(rs.getFirstResult().equals("3"));
}
```

Figure 5.1: Example: four unit tests for testing a DBMS.

```
class addToStudentUNDO extends UndoOperation {
  StudentDBManager db;
  int s_id;

  public addToStudentUNDO(StudentDBManager db, int s_id,
      Sring name) {
    thid.db = db;
    this.s_id = s_id;
  }

  public void undo() {
    //check database is still accessible...
    if (db == null) {
      db = getDBManager();
    }
    db.deleteStudentByID(s_id);
  }
}
```

Figure 5.2: A class definition implementing undo semantics of `addToStudent` method.

operational abstract level benefitting from the semantics of the code rather than the low level data and heap values. For example, consider the undo operation definition in Figure 5.2 for the `addToStudent` method.

An instance of `addToStudentUNDO` class *undoes* the operational semantics of the `addToStudent` method. The undo operation is defined in a class whose constructor takes the receiver object of the original `addToStudent` method call, and other parameters identical to the parameters required to do a `addToStudent` method call. Since the student table defined in our schema has `s_id` as a primary key, then deleting the student entry with the same `s_id` restores the state of the database to an equivalent state prior to adding the entry in the first place. Thus we only need to store the `s_id` value to perform the undo operation. Even though this undo operation does not retrieve the original state of the database in terms of the internal data structures representation and file indices, an equivalent state is restored which contains the same entries in the database tables. In other database application contexts, different definitions for undo operations might be needed. In Section 5.4, we explore using database native save-point/roll-back operations to restore the state of the database.

Given the required definitions of undo operations by the user, based on the semantics of the tests, our approach performs a source-code analysis of the tests and builds an execution trie, similar to a prefix-tree [99] of the source code of the tests. We define a *checkpoint* to be the point where two or more tests have different execution statements, irrespective of pure methods. A method is annotated by the user as pure if it does not change the program state, thus it is not required to revoke its effects on the program in order to execute subsequent tests.

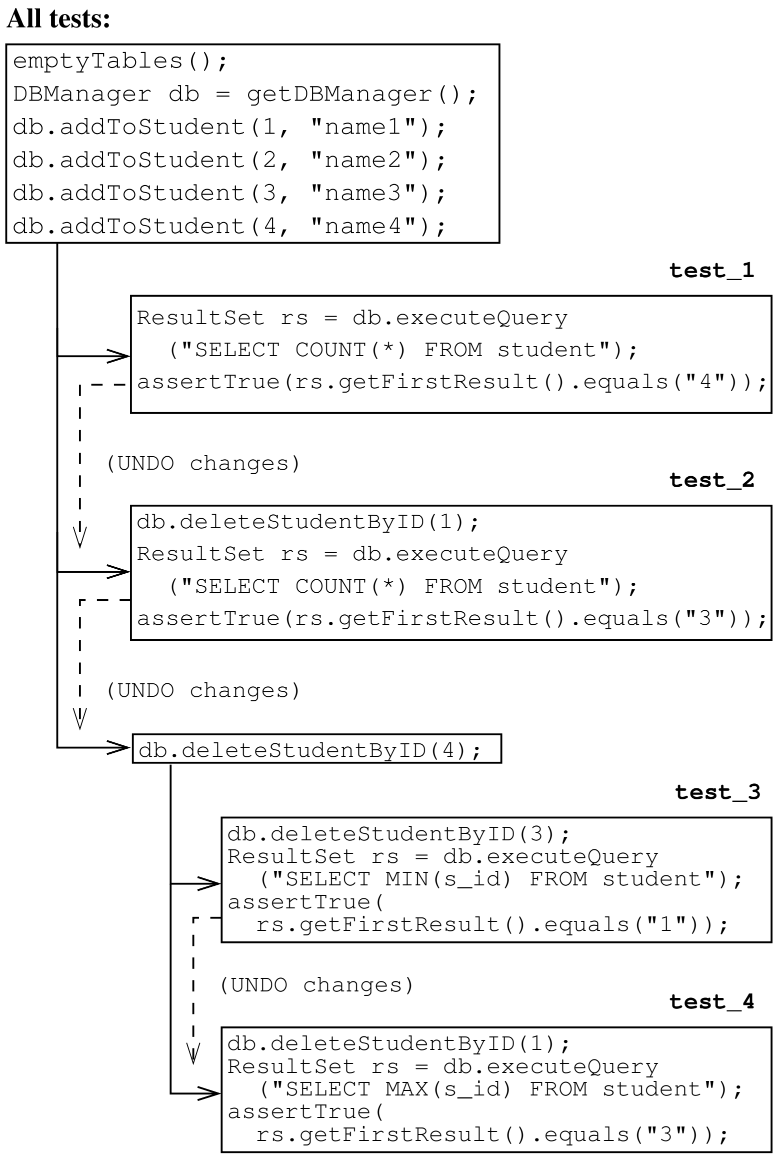
Figure 5.3 shows the equivalent execution trie for the four example unit tests. Running the four tests can now be seen as one test which executes the four tests consecutively. The operational semantics of the tests are maintained by running undo operations which restore the state of the database at each checkpoint. For example, to run `test_3` after executing `test_2`, we execute the undo operations corresponding to the previous step, specifically we run the undo operation for line 8 of `test_2`. This restores the student entry which had `s_id` equals 1, and thus restoring the database state equivalent to executing lines 1 through 7 of `test_3`.

In our simple example, our automated approach for clustering the four unit tests was able to reduce the number of database calls from 29 to 17, including the undo operation database calls. Section 5.4 shows how this approach scales on larger tests in different experiments and the significant speed-ups gained.

5.3 Framework

This section describes our approach for clustering tests based on state storage and retrieval using abstract undo operations. We first define the interface for *Undo* operations, then we describe the algorithms for clustering tests together and executing them.

Our approach for clustering tests is similar to systematic search algorithms [72, 93] in its essence. Model checkers, for instance, perform a bounded exhaustive exploration of the search space by exploring nondeterministic choices of the search variables. Some model checkers, example [108], store the state of the program, including the heap, stack, and static memory, at every *choice point*, i.e. the program statement where a nondeterministic choice is performed. Upon reaching a *termination point*, i.e. a program statement that specifies the end of a search path, the model checker retrieves the last stored program state and continues with the next possible choice in the search. This step is called the *back tracking* step. Figure 5.4, gives an abstraction of the



```
emptyTables();
DBManager db = getDBManager();
db.addToStudent(1, "name1");
db.addToStudent(2, "name2");
db.addToStudent(3, "name3");
db.addToStudent(4, "name4");
```

test_1

```
ResultSet rs = db.executeQuery
  ("SELECT COUNT(*) FROM student");
assertTrue(rs.getFirstResult().equals("4"));
```

test_2

```
db.deleteStudentByID(1);
ResultSet rs = db.executeQuery
  ("SELECT COUNT(*) FROM student");
assertTrue(rs.getFirstResult().equals("3"));
```

```
db.deleteStudentByID(4);
```

test_3

```
db.deleteStudentByID(3);
ResultSet rs = db.executeQuery
  ("SELECT MIN(s_id) FROM student");
assertTrue(
  rs.getFirstResult().equals("1"));
```

test_4

```
db.deleteStudentByID(1);
ResultSet rs = db.executeQuery
  ("SELECT MAX(s_id) FROM student");
assertTrue(
  rs.getFirstResult().equals("3"));
```

Figure 5.3: Example: clustered execution of unit tests.

storing/retreiving process for backtracking program execution. White nodes are the choice points and black nodes are terminations points. Backtracking takes place after executing a termination point to go back to a choice point, or from a choice point to a prior choice point if all nondeterministic choices have been explored. In general, this approach can be used for combining tests that share program execution. However, applications that interact with external memory, such as database applications and web applications, cannot have their program state retrieved by a conventional state storage/retreival approach.

In the context of running unit tests, our search space is the set of tests available. The choice points are the specific statements where two or more tests differ, and the termination points are the end of each test. A simple straight forward approach for storing the program state by taking a snapshot

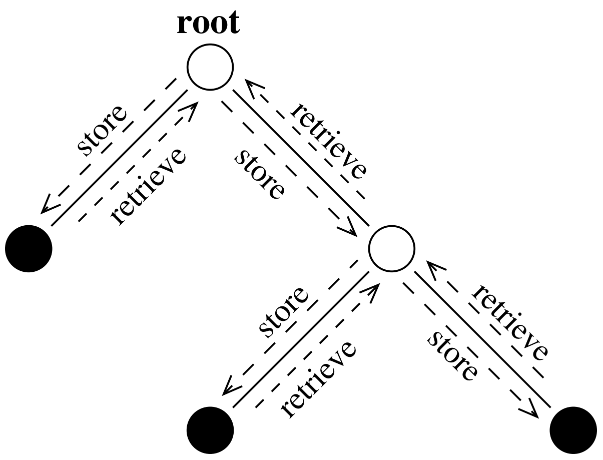


Figure 5.4: Backtracking process: White nodes are choice points while black nodes are termination points. Backtracking takes place after executing a termination point to go back to a choice point, or from a choice point to a prior choice point if all nondeterministic choices have been explored.

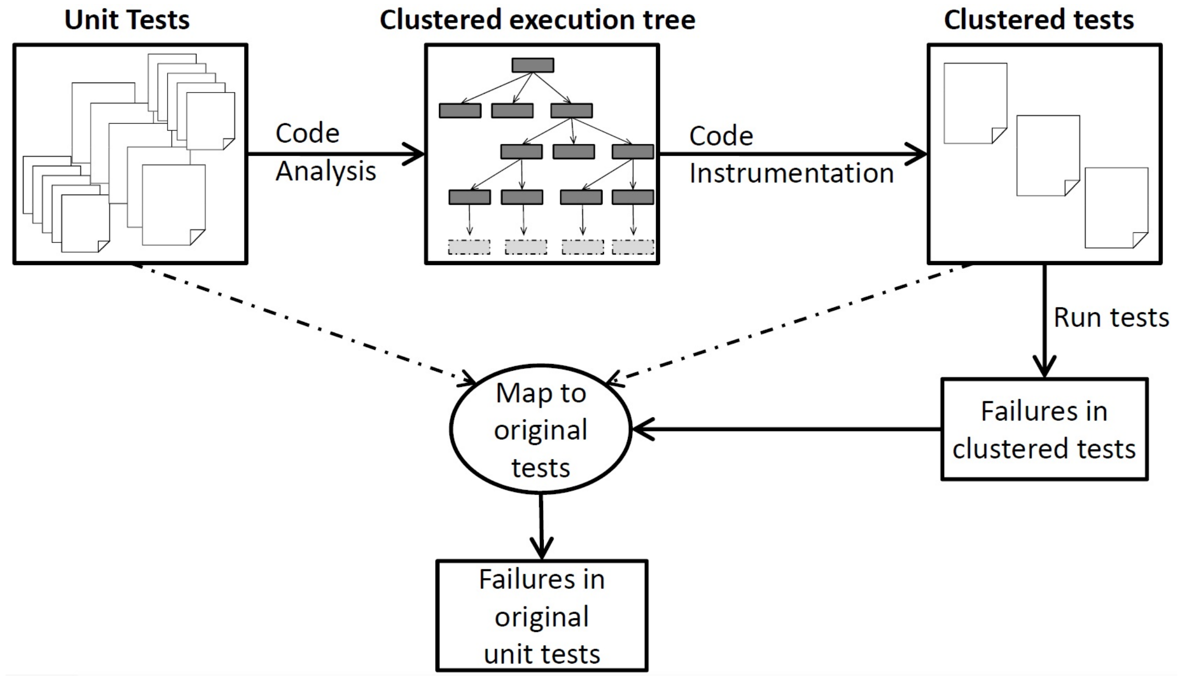


Figure 5.5: Clustered test execution framework.

of the program at each choice point imposes a huge overhead. In the cases where storing the program state can be used for back tracking, i.e. there is no external resources affected by the program execution, our experiments showed that the overhead of using a full state storage and retrieval mechanism was usually as slow as rebuilding the state though re-execution of the test code.

Our framework of clustering and executing unit tests is described in Figure 5.5. The framework takes as input a set of unit tests which are likely to share code execution paths. The framework then analyzes the tests and builds a tree representation of common paths between the tests. Then it instruments a list of clustered tests out of the original set of tests. The clustered tests include the instrumented code necessary for storing and backtracking states between the tests. Finally, it runs the clustered tests and reports passes and failures. The reports are mapped to the original unit tests for a better view of the results.

The following sections explain the different parts of the framework and the algorithms used in each.

5.3.1 Undo Operations

We propose an alternative approach for storing and backtracking a program state. Our approach is based on the *Command* design pattern [48], where a command is an entity in the program that can perform two operations: (1) the *execute* operation which executes the intended behavior of the command and (2) the *undo* operation which reverses the effects of a previous call to *execute*. Using this technique, a state of a program can be stored by maintaining a history list of executed commands. Retrieving the program state is achieved by traversing the list backwards and executing the *undo* operation of each command in the list. Each command stores the required information in order to be capable of reversing its execution.

In the *Command* design pattern, a command has both its *execute* and *undo* code implementation embedded within one command instance. However, in the context of running tests, it is more flexible and easier for a developer to define the *undo operations* at the method level execution. Thus, the user defines the undo operations in separation of the original code which might have

```
// An interface for undo operations
public interface UndoOperation {
  public void undo();
}

// A template class definition for implementing the
// UndoOperation interface.
class className_MethodName_UNDO implements UndoOperation {
  // store the data required to undo the operation
  private type data;

  // A constructor for the undo operation
  // The first parameter is the receiver object of the
  // original method call, followed by the list of
  // parameters which the original method has been called.
  public className_MethodName_UNDO (className recevierObject
     , original method parameters ...) {
    // store required data in the private fields
    this.data = ...;
  }

  // The undo method gets called to reverse the effects
  // of executing the original method
  public void undo () {
    // undo the operations
    ...
  }
}
```

Figure 5.6: The `UndoOperation` interface includes the main *undo* method that needs to be defined, followed by a template class definition for a general implementation of the interface.

been previously implemented. Figure 5.6 shows the general `UndoOperation` interface. It includes the `undo` method that needs to be defined in any class implementing the interface. Figure 5.6 shows as well a general definition of a class implementing the `UndoOperation` interface. The class name constitutes of the original method's class name followed by the method's name and the `UNDO` word. The constructor of a class implementing the interface includes the original receiver object which calls the original method, followed by a list of the parameters which the original method is called with. Required information to reverse the effect of executing the original method is stored in private fields which are later used when calling the `undo` method.

In addition to undo operations specified by the user at the method levels in the tests, we automatically generate undo operations for object field accesses within the test code. The undo operation would be specific to the field of the receiver object and it stores the field's value just before being updated. To illustrate consider the example in Figure 5.7. Figure 5.7(a) shows how the undo operation for updating the `score` field in the `Player` class is inserted before the field update. The code generated for backtracking is explained further in Section 5.3.3. In order to backtrack to a state prior to updating the value of the field, its value is temporary stored in the `Player_score_UNDO` instance as shown in Figure 5.7(b). Calling `Player_score_UNDO.undo()` restores the value of the `score` field.

5.3.2 Code Analysis

Our approach for clustering tests together is based on source code analysis. Starting with the root of every test we examine matching statements in other tests. This is done similar to building a prefix tree structure, also called

```
// Backtracking field accesses
@Test public void test() {
    Player bill = new Player();
    bill.score = 100;
    ...
    // Store the value of score for backtracking
    addUndoOperation(
        new Player_score_UNDO(bill, bill.score);
    // update the score field
    bill.score = 85;
    ...
}
```

(a) Automatic field access undo operations

```
// The undo operation for field accesses
public class Player_score_UNDO implements UndoOperation {
  Player player;
  int score;

  // store the player receiver object and old field value
  public Player_score_UNDO(Player player, int score) {
     this.player = player;
     this.score = score;
  }

  // undo field update by restoring the old field value
  public void undo() {
     player.score = score;
  }
}
```

(b) UndoOperation definition of object field

Figure 5.7: Auto generated *undo operations* for field accesses within the test code.

trie structure. Tries are often used for storing strings or other sequences and allows a fast search. It is often used to suggest words that begin with a given prefix fast. Figure 5.8 shows an example of a prefix tree. Each node stores a character. Words are formed by traversing paths from the root to leaf nodes. Similar to storing characters at nodes, we store common code execution statements at each node. Once two or more tests have different code a branching into another trie layer is created. The branching points are the choice points. In addition to comparing method calls and field accesses, we check for *pure* methods. Those methods do not change the program state. Such methods can be declared by the user using *@Pure* annotation. Static analysis of the code can check field accesses and updates, and we can conclude if a program mutates the state of the heap. However, such techniques cannot detect, for instance, if a SQL statement updates the state of the database or simply queries it without any change without the guidance of the user. Thus, we do not consider pure methods as choice points for building the tests trie, instead we merge different tests with different pure methods together, and they are run once in the node they belong to.

Clustering tests into a trie divides the execution code of each test into the nodes of the trie. Each unique full test is the augmentation of the code execution from the root to a leaf node. Clearly, not all tests have code in common, thus analysis could result in multiple clustered tries.

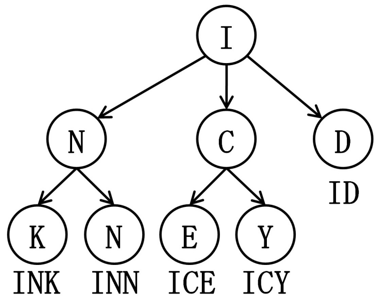


Figure 5.8: Prefix-tree, each path in the tree represents a word.

5.3.3 Code Instrumentation

The trie structure generated from combining common sections of tests together is the backbone of instrumenting the final clustered tests. At this level, we traverse the prefix trees and generate tests corresponding to the same choice and terminating points in the tree. The idea is to do a *pre-order* traversal of the tree, i.e. recursively traverse the root, left sub-tree, and then the right sub-tree. We perform two main operations while traversing the tree: (1) store the execution steps, i.e. keep a list of operations executed while going from a parent node to a child node, and (2) backtrack the operations, i.e. run the undo operations stored in the first step while going from a child back to a root node.

Figure 5.9 shows the algorithm for instrumenting the test code out of a prefix tree of clustered tests. The initial parameters are the prefix tree root reference and a *StateManager* object. The `traversePrefixTree` method traverses the tree in a depth-first-search way and generates the instrumented code using a global `codeManager` object. The key idea is that for each node we

```
// Instrument the prefix tree into a storage/retrieval
// clustered execution test.
// The parameters are the root of the prefix tree and
// the StateManager object responsible for storing
// and backtracking the program state.
traversePrefixTree(Node root, StateManager sm) {
  // Create a new state for each choice point
  State state = sm.addNewState();
  // Add the instrumented code for this node to the test
  Code stmts = instrumentCode(root.getCode(), state);
  codeManager.append(stmts);
  // Base case for recursive call
  if(root.isLeaf()){
    // Backtrack since we reached end of path
    State lastState = sm.backtrackLastState();
    codeManager.append(lastState.getBackTrackCode());
    return;
  }
  // Depth first traversal of nodes to visit all tests
  for (Node child : root.getChildren()) {
    traversePrefixTree(child, sm);
  }
  // Backtrack to a previous choice point
  State lastState = sm.backtrackLastState();
  codeManager.append(lastState.getBackTrackCode());
}
// Create and add undo operations before each non-pure
// statement. Store the operations in the state object.
Code instrumentCode(Code stmts, State state) {
  Code nodeCode = new Code();
  for(Statement stmt : stmts) {
    if (!stmt.isPure()) {
      UndoOperation undo = stmt.initUndoOperation();
      state.push(undo);
      nodeCode.append(stmt.getUndoOperationCode());
    }
    nodeCode.append(stmt);
  }
  return nodeCode;
}
```

Figure 5.9: Algorithm for instrumenting the clustered tests out of the prefix tree.

```
// Manage the program states across the execution
// paths of the tests.
class StateManager {
  Stack<State> programStates = new Stack<State>();
  State currentState = new State();

  State addNewState() {
    programStates.push(currentState);
    currentState = new State();
    return currentState;
  }

  State backtrackLastState() {
    State temp = currentState;
    currentState = programStates.pop();
    return temp;
  }
}

class State {
  Stack<UndoOperation> undoStack =
     new Stack<UndoOperation>();

  void push(UndoOperation op) {
    undoStack.push(op);
  }
  // Generate the backtracking code out of the undo
  // operations in the state stack
  Code getBackTrackCode() {
    Code backTrackCode = new Code();
    while(!undoStack.isEmpty()) {
      Statement undoStmt = undoStack.pop().getUndoStmt()
      backTrackCode.append(undoStmt);
    }
    return backTrackCode;
  }
}
```

Figure 5.10: StateManager responsible for maintaining a local state for each node in the prefix tree.

maintain a local state. This state is backtracked once all the subsequent statements, in the children's nodes, have been backtracked. The `instrumentCode` method is responsible for injecting code that maintains the state by initiating *UndoOperations* prior to every non-pure statement. References to those undo operations created are stored in a state object which in turn injects the code to backtrack the state by calling the `undo` methods of each operation.

Figure 5.10 shows the `StateManager` class responsible for instrumenting the code for maintaining the local states for each node in the prefix tree. The state manager maintains a stack of `state` objects which are pushed into the stack as the tree is traversed from the root to the leaf nodes. At each node a new *State* object is created. Each *State* object maintains a stack of *UndoOperations* which are pushed as the statements of the tree node are instrumented. To instrument the backtracking code, the state pops each *UndoOperation* from the stack and instruments a call to its *undo()* method.

5.3.4 Running Clustered Tests

As a final step of the framework, we run the clustered tests as unit tests itself. We report all passes and failures in the tests. Since the tester is interested in the original unit tests used, we map the failures and their specific lines of code to the original test's lines of code which result in the failure. This is achieved by creating a map between the original test code and the instrumented code in Section 5.3.3. The map is a one-to-one correspondence and thus a failure in the clustered test is mapped directly to the failure in the original test. If the failure happens at the level of clustered code, i.e. in non-leaf nodes of the prefix tree, we use the state manager to keep track of which test was running at the time of the failure. Usually we report all tests that share this code to fail. This gives the user a better understanding of the failure and a list of all tests that need to be updated in case the problem is in the tests themselves. However, in case of a badly implemented undo operation, subsequent tests following the bad backtrack might fail which gives the use a hint that the problem is in the undo operations.

5.4 Case Studies

In this section, we demonstrate the potential benefits of our approach. We consider the following research questions:

1. **RQ1:** How does common execution path locations in the test code affect clusters execution time?

2. **RQ2:** How does the complexity of the code and its location relative to the test affect the clusters execution time?

3. **RQ3:** How does the number of *checkpoints* in the tests affect clusters execution time?

To answer these questions, we performed three case studies. The experiments were run on an Intel Core i7 CPU (Quad Core) 2.8GHz with total 8GB RAM.

5.4.1 RQ1: Database Case Study

In this study, we show the effectiveness of our approach on running database unit tests. In common black-box database engine tests, each test initially populates the database schema with test data then runs SQL commands on top of the data and verifies the results. Those commands can change the state of the database by adding, updating, or removing data from the database. Thus each test must clear the database from any data and populate it again to run. Our approach clusters the tests using abstract undo operations that store and retrieve the state of the program and the database.

We use two approaches for defining the *Undo* operations for database statements. The first approach is similar to the example is Section 5.2, where each `INSERT` statement, wrapped within `addToStudent()` method, is associated with an undo operation, `addToStudentUndo()` method. Figure 5.2 shows the definition of the undo operation. This approach is valid since we can reverse the effect of adding a student with a unique `s_id` by deleting the student with that `s_id`. However, in the case where the attributes of the table are not unique or where multiple rows of the table are affected by one operation on the database, such techniques might fail and we would need to store a big set of data to reverse the operation. So, our second approach is to use the database built-in *save-point* and *roll-back* operations. Once a data set is inserted into the database, a save-point is created, later on we can reload the state of the database at that point by calling the database roll-back function. Figure 5.11 shows an implementation of the undo operation using this approach. A save-point is created before the database is altered and thus we can undo the operations by rolling back to that exact point.

```
class addDataSetUNDO extends UndoOperation {
  Savepoint sp;

  public addDataSetUNDO(StudentDBManager db, ...) {
    sp = db.getConnection().setSavepoint();
  }

  public void undo() {
    //check database is still accessible...
    if (db == null) {
      db = getDBManager();
    }
    db.getConnection().rollback(sp);
  }
}
```

Figure 5.11: A class definition implementing database undo operation using a *save-point/roll-back* approach.

To control this experiment, we generate tests that are clustered into a full tree structure, as described in Section 5.3.2, of three levels. That is, each test execution is divided into exactly 3 nodes containing different SQL statements that alter the state of the database and some assertions on SELECT queries on the data. To answer our first research question, we generate database tests which have common execution paths at different levels of the tree. We run the tests on Oracle 11g database [88]. Figure 5.12 shows the case where the different execution paths are varied at the end of the tests. That is, we fix the common test executions to the initial parts of the tests and we vary

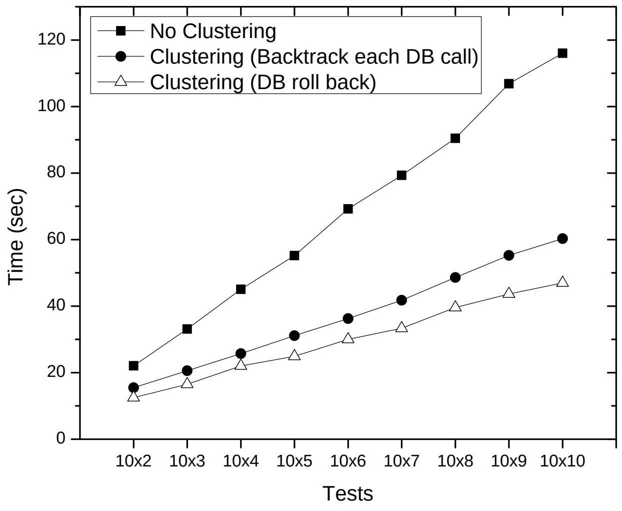


Figure 5.12: Clustering DB tests by increasing the difference in execution paths at the end of the tests. Each cluster is represented by the product of the root's number of children and each child's number of children. For example, 10x7 is a cluster with a root of 10 children, each of which has 7 other children as leaf nodes.

the operations at the end of the test. On the other hand, Figure 5.13 shows the case where the different execution paths at the beginning of the tests are varied. That is, we add more tests that have different execution codes at the beginning of tests rather than the end.

For both tests, in Figure 5.12 and 5.13, a cluster tree is represented by a product or the branch factor of its nodes. For example, a 10x7 cluster tree has a root of 10 children and each child has 7 other children as leaf nodes, to give a total of 70 tests. Figure 5.12 shows that as the number of tests with differences at the end of the tests increases, the speed-ups increase. This is the expected result since we are adding more tests that need less backtracking

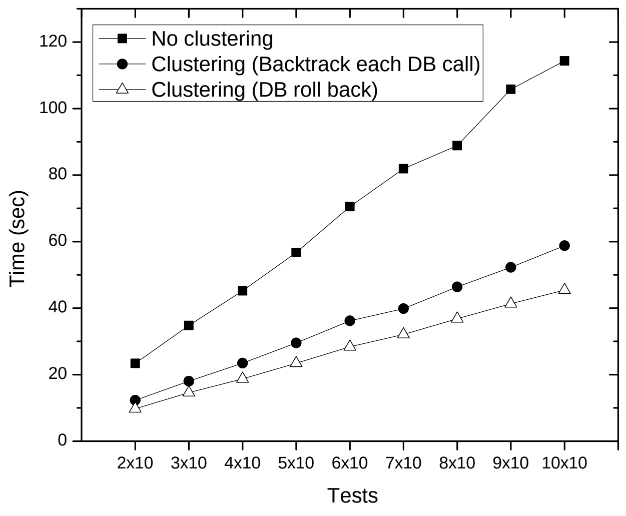


Figure 5.13: Clustering DB tests by increasing the difference in execution paths at the beginning of the tests.

operations to run. The tests showed a speed-up increase from 1.59X to 2.5X for clustered execution using roll-back operations as the second level branching factor increased from 2 to 10, and an increase from 1.36X to 1.95X for clustered execution with undo operation for each database statement.

Figure 5.13 shows that as the number of tests with differences at the beginning of the tests increases, the speed-ups are not affected. However, the speed-ups in this experiments were as much as the maximum speed-ups achieved in the previous experiment. The speed-up for clustered execution using roll-back operations was about 2.5X across all clusters and about 1.93X for the clusters with undo operation for each database statement. The results were expected since as we add more tests that share less code with previous tests and thus we maintain the same speed-up.

Table 5.1 shows the difference between the two backtracking approaches we used. The first approach adds an undo operation for each database statement. For a set of 100 unit tests, we ran a total of 110,000 database statements including a total of 55,000 undo operations. The root level operations do not need to be back-tracked. The speed-up was 1.97X compared to running the tests without any clustering. The second approach of using save-points has a roll-back operation for each 5,000 database statements and thus the undo operations were mere 110 for all the tests. The speed-ups were 2.53X. Even though we used one undo operation for every 5,000 database operations, the database still does a complex internal work to roll-back the operation.

5.4.2 RQ2: Web-services Case Study

In this case study, we use web-service calls to reflect the complexity of a test. A web-service call sends a request over the internet to a remote server and waits for a response. We use Google Maps web-services [54] to request driving directions between two cities. The request is made over *https* connection protocol and the response is a string representation of an XML document. We generate tests that do different web-service calls each and perform different XML additions, deletions, and combinations of other XML documents. We wrap the XML reader and writer with proper undo operations based on the tag IDs and names.

To answer our second research question, we generate tests that do web-service calls at different locations in the tests. A web-service call is usually

Approach	# Tests	# DB Calls	# Undo's	Time (sec)	Speed-up
No cluster	100	200,000	0	114	0
Clustering (1)	100	110,000	55,000	58	1.97X
Clustering (2)	100	56,110	110	45	2.53X

Table 5.1: Clustering DB unit tests using two approaches: (1) backtracking using *Undo* operations for each database statement, and (2) backtracking using DB *save-point* and *roll-back* operations.

an expensive operation since it requires a network connection. We generate three clusters of 4 levels; the root is at level 0 and leaf nodes at level 3. Each cluster tree is full and contains 20 tests, and each test calls a total of 10 web-service calls. Table 5.2 shows the results of running the tests. The first cluster contains tests that have web-service calls distributed across the 4 levels of the cluster tree. The second cluster adds more complexity to the root instead of the other nodes. The third cluster does 7 web-service calls at the root level and one call at each of the other level, putting a major complexity on the beginning of the cluster.

As expected, the higher the complexity is at lower levels of the cluster tree, i.e. at levels closer to the root, the better the speed-up gains. Table 5.2 shows that as the complexity of the root increased from 10% of the total web-service calls of a test to 70%, the speed-up increased from 1.8X to 6.7X.

5.4.3 RQ3: Data Structures Case Study

In this study, we answer our third research question. We do so by generating full clustered trees of up to 7 levels. The tests manipulate data structures, in specific, we use the Java *LinkedList* implementation and add undo operations for the *add* method. Each test adds 10,000 random integers to the data structure at random indices.

Table 5.3 shows the results of our experiment. We have three sets of clusters with order of 3, 4, and 5, where the *order* is the number of children for each node in the full cluster tree. The *Tree Height* is the height of the tree,

Cluster #	# Nodes	# Tests	# Web-service Calls				Total # Web-service Calls	Avg. Time / Test (sec)	Time (sec)	Speed-up
			level 0	level 1	level 2	level 3				
No cluster	–	20	10	0	0	0	200	0.613	12.27	0
1	37	20	1	3	3	3	106	0.335	6.69	1.83X
2	37	20	4	2	2	2	74	0.246	4.91	2.5X
3	37	20	7	1	1	1	42	0.140	2.79	4.39X

Table 5.2: Clustering tests with 10 web-service calls each. The tests are clustered into a trie with 4 levels; the root is at level 0 with 5 children. Level 1 and 2 have 2 children and leaf nodes are at level 3. We vary the clusters by adding the common complexity to the root instead of the leafs.

and it represents the number of nodes or checkpoints for each test from the root till the leaf node of the test. The # *Nodes* is the total number of nodes in the cluster tree. The table shows that as the number of checkpoints increases the speed-up gain is higher. This is because the increase in the number of checkpoints means tests share more execution path sections and thus benefits from the relatively small backtrack to run. In contrast, if we have the same tests with less checkpoints, it means that the branching factor of the nodes is higher and each backtrack is more expensive than the previous scenario. The table also shows that as the *order* increases, and thus increasing the number of checkpoints however on the breadth level instead of the depth level, the speed ups are slightly affected. This is also intuitive since adding a branch to a node means that the test has execution path difference at higher positions in the code. This is similar to the results in Section 5.4.1 (Figure 5.12), however in this test the order increases the branching factor to all nodes of the tree rather than just the root node.

Order	Tree Height	# Nodes	# Tests	Cluster	# Operations	# Undo's	Time (sec)	Speed-up
3	4	40	27	NO	270,000	0	0.497	N/A
				YES	197,500	97,500	0.428	1.16X
	5	121	81	NO	810,000	0	1.503	N/A
				YES	482,000	240,000	1.119	1.34X
	6	364	243	NO	2,430,000	0	4.547	N/A
				YES	1,211,667	605,000	2.769	1.64X
	7	1,093	729	NO	7,290,000	0	13.696	N/A
				YES	3,121,429	1,560,000	6.742	2.03X
4	4	85	64	NO	640,000	0	1.167	N/A
				YES	422,500	210,000	0.882	1.32X
	5	341	256	NO	2,560,000	0	4.687	N/A
				YES	1,362,000	680,000	2.813	1.67X
	6	1,365	1,024	NO	10,240,000	0	18.679	N/A
				YES	4,548,334	2,273,334	9.190	2.30X
	7	5,461	4,096	NO	40,960,000	0	74.496	N/A
				YES	15,601,429	7,800,000	32.271	2.31X
5	4	156	125	NO	1,250,000	0	2.248	N/A
				YES	777,500	387,500	1.618	1.38X
	5	781	625	NO	6,250,000	0	11.191	N/A
				YES	3,122,000	1,560,000	6.467	1.73X
	6	3,906	3,125	NO	31,250,000	0	56.352	N/A
				YES	13,018,334	6,508,334	26.344	2.14X
	7	19,531	15,625	NO	156,250,000	0	282.164	N/A
				YES	55,801,429	27,900,000	113.614	2.48X

Table 5.3: Clustering tests manipulating large data structures. The *order* is the number of children for each node in the clustered tree. The *Tree height* is the height of each cluster tree, where root is at height 1. The *# Operations* is the total number of data structure operations executed in all the tests in the cluster, including *undo* operations if any. *# Undo's* is the total number of *undo* operations executed for running the cluster.

Chapter 6

Related Work

6.1 Mixed-Constraints

The JABAL framework provides a novel notation for mixed constraints to facilitate writing constraints by supporting a combination of declarative and imperative paradigms. It provides an analysis technique for solving mixed constraints using a combination of solvers, where each solver is designed for constraints written using one particular paradigm. To our knowledge, JABAL is the first framework for writing and solving constraints in a mixed paradigm.

Narayanan's Master's thesis [85] presented an approach for mixing declarative and imperative constraints within the Java language. However, the work presented there lacks the use of def-use chain analysis, and doesn't support formulation of imperative style Alloy formulas. Our work also mixes constraints at the Alloy level, keeping the Java source code intact and benefiting from static analysis and systematic input generation without the need of explicit field annotations to be added to the constraints.

A recent paper [49] presents annotations for Alloy models to guide solving of Alloy constraints using a variety of dedicated solvers, including an Integer constraint solver and a String constraint solver. However, the focus of that work is on constraints written purely in Alloy. In contrast, JABAL supports mixed constraints, annotations for def-use, and incremental solving using solvers designed for constraints in different programming paradigms.

Annotations in JML (Java modeling Language) [75] support writing constraints, including quantifiers, on top of Java methods. However, they do not support test input generation neither checking constraints using other solvers.

Incremental solving for Alloy models, where a solution to one formula is fed as a partial solution to efficiently solve another formula, was introduced in Uzuncaova's doctoral work [106,107], which also applied it in the context of test input generation for product lines. Their work did not support annotations, rather used a heuristic def-use analysis to prioritize constraints purely written in Alloy. In their work, the structure of a product line was leveraged for incremental solving.

Model checkers [38] have been used in analyzing software programs and verifying concurrent and distributed systems [30]. Recently, Alloy has been gaining popularity for modeling and analyzing software systems [1,67]. These techniques involve modeling both the program inputs and as well as the computations in Alloy. The TestEra framework [69] for specification based testing generates test cases from Alloy specifications of data structures. JABAL uses a similar approach used in TestEra for translating Alloy solutions into concrete data structures. However, JABAL does not only translate abstract solutions into concrete ones, but it maintains a partial solution and updates it based on the concretization.

The Java PathFinder model checker [108], has been used to find errors in a number of complex systems [8]. In [109], Visser et al. show how JPF can be used for test input generation as well. We use their technique in our approach for solving constraints written in Java. However, JPF solver in JABAL takes a partial solution as input and updates it as required. Another efficient constraint-based testing framework for Java programs is Korat [81]. Korat uses Java predicates to generate test input. Based on the predicate execution, it monitors fields accessed while generating candidate inputs. We are investigating the use of Korat as a Java constraint solver in JABAL.

In [47], Frias et al. present DynAlloy which is an extension to Alloy to describe dynamic properties of actions in software systems. This allows the user to check for properties in execution traces. They also present the notion of partial correctness assertions which are used to describe properties regarding executions. JABAL, on the other hand, provides a similar notion for verifying dynamic properties as well. We can write constraints in Alloy, execute Java code, then check the properties of the output state. However, our purpose of using JABAL has been focused on test input generation.

Significant research has been done in the field of cooperate constraint solving [103]. For instance, the Satisfiability Modulo Theories (SMT) solvers [33, 37] combine solving logical formulas from different background theories. However, the techniques introduced are effective at the level of combining these theories to help solve formulas. In our research, we propose mixing constraint solvers at the level of abstract representation using different paradigms. The solvers that we use can benefit from the research in the field of cooperate constraint solvers, thus helping solve Alloy or Java constraints faster. In this context, El Ghazi and Taghdiri present in [50] an approach for solving Alloy models using SMT solvers. However, still their approach supports Alloy language and does not support using different programming paradigms.

Combining an imperative language with a model checker has been introduced in the SPIN [60] model checker. SPIN allows writing constraints in Promela (Process Meta Language) and allows the insertion of C code into the model. This integration is feasible since SPIN generates C-code for a problem-specific model checker, however it does not support the use of different solvers to solve separate constraints. In [24], the authors combine Alloy with temporal logic by combining the logic into one BDD representation. However, they do not solve constraints written in Java imperatively and they only support integer and Boolean primitive types.

6.2 Systematic Database Testing

Our framework for systematic black-box testing of database engines uses Alloy [64, 65] and the Alloy Analyzer [66] as means for test input and oracle generation. However, the use of Alloy for modeling and analyzing software systems is not new. Alloy has been previously used to analyze software systems [67, 70]. These techniques involve modeling both the program inputs as well as the computations in Alloy. Alloy has been also used for specification based testing. TestEra [69,71] and Korat [18,81] are the first two frameworks to provide systematic generation of structurally complex tests from constraints.

Our framework unifies the generation of (1) syntactically and semantically correct SQL queries for testing, (2) meaningful input data to populate test databases, and (3) expected results of executing the queries on the DBMS. Previous work has addressed each of these three steps, but largely in isolation of the other steps [90, 95]. While a brute-force combination of existing approaches to automate DBMS testing is possible in principle, the resulting framework is unlikely to be practical: it generates a prohibitively large number of test cases, which have a high percentage of tests that are redundant or invalid, and hence represent a significant amount of wasted effort. Some approaches, such as [20], target generating queries with cardinality constraints. Integrating query generators with data generators, however, is still either specialized [90], or sometimes not possible [20]. Several academic and commercial tools target the problem of test database generation [19,32,61,96]. Nevertheless, they do not support query generation nor test oracle generation. Recent work in query aware input generation [78] takes a parameterized SQL query as input and produces input tables and parameter values, but does not generate an oracle. Recent approaches introduced query-aware database generation [16, 17]. These approaches use information from queries as a basis to constrain the data generator to generate databases that provide interesting results upon query executions. Query-aware generation is gaining popularity in both DBMS and database application testing [42, 112] but requires providing queries and test oracles manually.

Our framework uses a similar approach to that of TestEra in using Alloy to specify the input and for checking the correctness of the output. However, it differs from TestEra in two key ways. (1) TestEra is specialized for testing programs that take linked data structures as inputs, whereas our framework targets testing database management systems. (2) TestEra requires the programmer to learn a new specification language in order to specify the input. On the other hand, we only require the input described in SQL (the language of the application under test) and systematically the Alloy specifications are generated.

The use of Alloy and the Alloy Analyzer enables the unification of the three key approaches for database testing: generation of SQL queries, test databases, and test oracles.

A popular framework for query generation is the Random Query Generator (RQG) [44], which uses the SQL grammar as a basis of query generation - in the spirit of production grammars. Given a grammar RQG generates random queries and tests databases by running the tests against two or more databases and comparing their results. Since the query generation is purely grammar-based, it generates a large number of invalid queries as well as redundant ones. Moreover, validating that the queries generated are syntactically correct is hard and sometimes impossible to ensure [44].

Test database generation is a well studied problem [16, 19, 32, 61, 96].

Several approaches perform data generation by analyzing a given database schema [31, 32]. Such approaches aim at generating large databases that are used as benchmarks for performing various analysis on databases.

Our approach for data generation is query-aware and targets generating a large set of small databases for exhaustively testing a DBMS system. Unlike other approaches which use constraint solvers [17], or object modeling language [96], our framework uses the Alloy Analyzer which in turn uses SAT to generate its data. The use of Alloy enables specifying constraints on both the query as well as the results which enables more precise test input generation.

Several database testing approaches target transaction testing, i.e., checking the effect of executing a sequence of related SQL queries on a database. For example, a recent tool, AGENDA [34], uses state validation techniques to verify the consistency of the database after executing a transaction. Our approach does not target transaction testing. The framework primarily considers SQL selection statements which unlike transactions do no update the state of the database.

6.3 Clustered Test Execution

Several approaches exist to optimize unit tests and regression testing, such as identifying special and common unit tests for object-oriented programs in [114] and incremental regression testing in [7]. However, there is no work, to our knowledge, that clusters existing unit tests for efficient execution. Our framework does not try to minimize the number of unit tests available nor tries to detect un-necessary or redundant tests. Instead, it analyzes similar tests to optimize the time required to run them.

Narayanan's Master's thesis [84] presented an approach for clustering unit tests using the Java PathFinder as a tool for state storage and backtracking. However, using JPF imposed a lot of overhead for running it's own JVM. In addition, their approach doesn't support backtracking of external resources state. On the other hand, our work uses user-defined undo operations that enables back-tracking of external resources and it does source code instrumentation that runs on the default Java JVM.

Different approaches for state storage and retrieval has been used in various contexts [108, 115]. The straight forward technique is to store the full program state. Other techniques use state comparisons, example [76], which incrementally updates a state by comparing it to previously stored one. In [115], a check-pointing technique is based on static analysis of the program to minimize the size of the state to save at each checkpoint. Those techniques can be integrated with our approach, however, the main difference is in our use of undo operations which give us the ability to retrieve the program state that updates external resources.

The idea of using Undo operations has been explored in model checkers [91], however our use of undo operations serves a different purpose for running clustered tests rather than exploring the search space. In addition, rather than performing the undo operations at the concrete heap levels we leverage abstract undo operations at higher levels of abstraction as the example shows in Section 5.2. Undo operations has been explored in the context of structural constraint solving in [41]. They systematically explore generation of data structures using undo operations for back tracking. Nevertheless, their application and implementation are different. We use this approach for clustering unit tests and our code instrumentation does not require using *TABLESWITCH* byte code commands.

In the context of running unit tests, Xie et al. in [113] propose an approach for detecting redundant object oriented unit tests using different comparison techniques. Even though two tests can have different executions they might have the same object representation as a final result. Their techniques can be useful, however it is focused on object oriented tests and we can not apply it to check the equivalence of two tests that use external resources. We believe that the statement sequences in the test define the test itself and changing the sequence implies running a different test. Incorporating their approach in our framework is a possibility. We might want to check for partial redundant tests and use it as a base for checkpointing the clustered tests. However, we use code instrumentation to generate the clustered tests whereas in [113] they require running the tests and comparing objects in the tests dynamically at run time.

Chapter 7

Conclusion

This chapter provides a summary of our main contributions on using test-summaries for efficient and effective testing of relational applications, followed by a discussion and future directions.

7.1 Summary

We presented novel approaches to improve the usability, effectiveness, and efficiency of systematic constraint-based testing. We further applied those techniques to the domain of relational applications.

To improve the usability of systematic constraint-based testing, we presented our technique for mixed declarative and imperative formulation of structural constraints. We introduced a notation that supports writing specifications describing input constraints using a combination of declarative and imperative programming styles. In specific, our approach enables the user to freely write and mix constraints using the expressive declarative Alloy language and the imperative object oriented Java language. We presented a new notation using a semi-colon and Java-style annotations in the Alloy grammar. The semi-colon notation provided the means to sequential analysis of Alloy formulas. A Formula separated by semi-colons assumes that the previous formula has been satisfied. In addition, annotations provided the tool necessary for defining input generation steps, and the analysis of user defined combined formulas.

To improve the effectiveness of systematic constraint-based testing, we presented our technique for solving input constraints using a combination of solvers that support different classes of input constraints written using different programming paradigms. Using the notation we previously defined, we were able to leverage the Alloy Analyzer to solve constraints written in Alloy and the Java PathFinder to solve constraints written in Java. We use those solvers in synergy for automated test input generation of complex data structures. This technique gives the user the power of choosing the appropriate solver to solve individual constraints. In addition, we can use the Java virtual machine to execute and perform operations on the defined system as part of a partial constraint solving.

To improve the efficiency of running systematically generated suites of tests, such as those generated by bounded-exhaustive testing techniques, we presented our approach of clustered execution by clustering common path executions using abstract level undo operations. Our approach executes initial common segments between tests once and shares its results among them, then it executes user-defined abstract undo operations to undo the specific test changes. This approach enables us to cluster tests that not only execute on heap data structures, but also those which affect external resources. The basis of our approach is checkpoint backtracking as a means to retrieve program state at choice points. However, we don't store the program state at each checkpoint, instead we only store undo-operations required to restore the state. Undo operation are defined by the user to undo the effect of a state change. This enables the user to define undo operations on external resources such as database systems which is hard to achieve is previous typical checkpointing techniques.

We presented experimental results that show the applicability and effectiveness of our approaches on different sets of commonly used data structures. We showed the potential use of combined declarative and imperative constraint formulation and solving for test input generation. This approach not only provides an easier way of writing constraints, but it also provides potential speed ups in test generation times. On the other hand, we also showed that clustering tests that share initial common execution paths can provide great speed-ups in execution time. It can also be used to minimize calls to expensive external resources by storing output to be shared with different tests.

In addition, we presented our framework for systematic testing of database engines using test summaries. Our database testing framework uses test summaries to automate the generation of syntactically and semantically valid SQL queries, data to populate the database, and test oracles to check the correctness of executing the SQL queries on the populated data. We model SQL queries and their execution semantics in Alloy. We were able to add constraints in Alloy defining the constraints on SQL queries which are hard to define with conventional grammar-based query generators. Thus, we were able to generate syntactically and semantically correct SQL queries in a bounded-exhaustive fashion. In addition, the natural map of Alloy relational logic to relational database logic enabled us to define SQL query execution logic which provided the means to generate test input and expected output as well. This framework provided a full automated system of generating the three main artifacts for testing database management systems. Our approach was able to reproduce and find new bugs in commercial and open source database management systems.

7.2 Discussion

In this section, we discuss some characteristics and limitations of our approaches.

7.2.1 Mixed-constraints

We presented a notation for mixed-constraint formulation and solving. This technique gives the user the ability to mix between different programming paradigms. Even though it provides flexibility in formulation constraints, it can also yield worse constraint solving time. This is due to the fact that constraint prioritization affects solving time. There is not an automatic way to know what constraints are better solved using a specified solver. However, with experience using different solvers, we can have an intuition of what solvers suite certain constraints.

We presented our approach for solving mixed constraints using the Alloy Analyzer and JPF. This technique imposes limitations of both of the solvers. Alloy Analyzer usually works well with small scope of variables, and JPF uses exhaustive bounded algorithms to examine all possible valuations of variables. Thus, the way user defines constraints both in Alloy and Java has an impact on solving times.

7.2.2 Clustered Execution

Clustering suites of tests that share common initial segments can provide more efficient execution times. However, this technique works the best with systematically generated tests since they usually have many common segments. On the other hand, for small tests it might not be as useful. For tests that only have few execution statements, it might be faster to execute each test separately. Nevertheless, our approach is also useful in minimizing expensive external resource calls, such as web-services and database calls, and can be necessary in real systems testing.

7.2.3 Database Testing

In the field of testing relational application, we presented several techniques for SQL query generation and test input and oracle generation. Using Alloy requires specifying a scope to the variables used in its formulas. This scope is used to limit the number of elements of each data type declared in a database schema. In our experiments, we notices that this scope works well with small numbers. However, for large values, i.e. to generate larger databases, the Alloy Analyzer significantly performs poorly. While this limitation prevents our framework from generating large databases, the query-aware generation still enables generating a collection of databases that are large enough to cover a range of interesting scenarios for testing SQL queries. The case studies presented in our example illustrate how by covering the different combinations of tables with small number of records we were still able to discover new bugs in database management systems.

On the other hand, the Alloy language provides a built-in notation for the integer data-type which simplifies the analysis of SQL queries that manipulate integers. Alloy types however are uninterpreted, and Alloy does not provide a representation for complex types such as, varchar, date, and time datatypes. Those data-types need a user-defined defined module.

7.3 Future Work

There are many directions for future work for each of our approaches.

7.3.1 Mixed-constraints

In mixed-constraint solving, we see the opportunity of using other constraint solvers such as Korat [81] for solving constraints written in Java. In addition, we can integrate other solvers dedicated for different constraint types such as dedicated integer constraint solvers. As a future work, we define an interface for a constrained solver that can be used in our system. Having API's for solvers that implement our interface makes it easy to plug-in solvers and using multiple solvers in synergy.

In our experiments, we used JPF as a test generation tool. We can also integrate the use of JPF in multi-threaded programs. For example, we can use Alloy to model certain data structures constraints such that they would impose a certain multi-threaded execution tree that can be verified by JPF.

We also see the opportunity of doing further analyzes to suggest to the user a certain solving constraint prioritization solution. Using some heuristics we can guess suitable solvers for certain data-types.

7.3.2 Clustered Execution

We see multiple directions for enhancing our clustered execution techniques. Currently, the analysis of test code is based on the source code. This direct source code comparison works well for systematic generation of tests. However, we can use better analysis of equivalent code segments. Research in source code comparison in code version systems might be useful.

We can also use object equivalence at runtime to determine common execution code. For example, if executing different code segments result in the same object heap structure, then we mark these code segments as equivalent. A simple approach can use the equal method on objects to detect equivalence. However, this approach might require run-time analysis of the heap memory.

Another future direction of clustered execution is to use the prefix-tree generated for parallel execution of clustered tests. The prefix tree provides a natural forking tree that can be easily parallelized.

7.3.3 Database Testing

The main direction for future work in the our approach for automated database testing is to extend the supported SQL grammar. Our approach showed how to use Alloy and the Alloy Analyzer to model a subset of SQL query grammar and its constraints to ensure the validity of the syntax and semantics of queries generated. Having an extensible framework, we can systematically add support for a larger subset of SQL grammar. For example, we showed how to integrate in our framework the types for table attributes; these types can be used to add type checking constraints in the WHERE, GROUP BY, and HAVING clauses. SQL transactional grammar can be extended as well. DELETE statements can be introduced by modifying the grammar as: `DELETE FROM TABLE WHERE term in (SELECT term FROM table WHERE condition).` The constraint that the term to be deleted is the same as the term to be selected is simple to write in Alloy. Nested SELECT statements can be extended by ensuring that the inner SELECT statements can have access to the outer SELECT terms but not vice-versa using the same approach. Along with extending the grammar, we can also extend the complex data types used in common SQL queries. Operations on these data types can be defined by the user in a simple interface.

Another future direction for database testing is to automatically suggest appropriate scope values for different data types in the database schema under test. This can be based on heuristics or empirical studies by reproducing bugs in different database engines and analyzing the scope properties for which bugs can be reproduced.

Bibliography

[1] Shadi Abdul Khalek, Bassem Elkarablieh, Y.O. Laleye, and Sarfraz Khurshid. Query-aware test generation using a relational constraint solver. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 238 –247, Sept. 2008.

[2] Shadi Abdul Khalek and Sarfraz Khurshid. Automated SQL query generation for systematic testing of database engines. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 329–332, Antwerp, Belgium, 2010.

[3] Shadi Abdul Khalek and Sarfraz Khurshid. Systematic testing of database engines using a relational constraint solver. In *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, pages 50 –59, March 2011.

[4] Shadi Abdul Khalek and Sarfraz Khurshid. Efficiently running suites of unit tests using abstract undo operations. In *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Hiroshima, Japan, Nov. 2011 (to appear).

[5] Shadi Abdul Khalek, Vidya Priyadarshini, and Sarfraz Khurshid. Mixed-constraints for test input generation. In *Proceedings of the 26th IEEE/ACM*

International Conference on Automated Software Engineering (ASE), Lawrence, Kansas, Nov. 2011 (to appear).

[6] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. TestEra: A tool for testing java programs using Alloy specifications. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lawrence, Kansas, Nov. 2011 (tool paper to appear).

[7] H. Agrawal, J.R. Horgan, E.W. Krauser, and S.A. London. Incremental regression testing. In *CSM'93: Proc. of Conference on Software Maintenance.*

[8] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Rosu, and Willem Visser. Experiments with test case generation and runtime analysis. In *Proceedings of the abstract state machines 10th international conference on Advances in theory and practice*, ASM'03, pages 87–108, Berlin, Heidelberg, 2003. Springer-Verlag.

[9] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. Integrating boolean and mathematical solving: Foundations, basic algorithms and requirements. In *Proceedings of the Joint international Conference on Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, volume 2385 of *Lecture Notes in Artificial Intelligence.* Springer, 2002.

[10] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of the 8th international SPIN workshop on Model checking of software.*

[11] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[12] Clark W. Barrett, David L. Dill, and Aaron Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Proceedings of the 4th International Workshop on Frontiers of Combining Systems, FroCoS'2002 (Santa Margherita Ligure, Italy)*, volume 2309 of *Lecture Notes in Computer Science*, pages 132–147, apr 2002.

[13] Kent Beck and Erich Gamma. *Test infected: Programmers love writing tests. Java Report*, July 1998.

[14] Boris Beizer. *Software Testing Techniques.* International Thomson Computer Press, 1990.

[15] G. Bernot, M.C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387 –405, November 1991.

[16] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *ICDE*, pages 506–515. IEEE, 2007.

[17] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: generating query-aware test databases. In *SIGMOD Conference*, pages 341–352, 2007.

[18] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. July 2002.

[19] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1097–1107. VLDB Endowment, 2005.

[20] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006.

[21] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015, pages 150–153, Paphos, Cyprus, 2010. Springer, Springer.

[22] Simon Burton. Towards automated unit testing of statechart implementations. Technical report, 1999.

[23] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30:775–802, June 2000.

[24] Felix Sheng-Ho Chang and Daniel Jackson. Symbolic model checking of declarative relational models. In *Proc. of the 28th international conference on Software engineering*.

[25] Juei Chang and Debra Richardson. Structural specification-based testing: Automated support and experimental evaluation. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 285–302. Springer Berlin / Heidelberg, 1999.

[26] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with adl. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '96, pages 62–70, New York, NY, USA, 1996. ACM.

[27] Lori A. Clarke. *Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation.* PhD thesis, University of Colorado at Boulder, 1976.

[28] Lev Nachmanson Wolfram Schulte Nikolai Tillmann Colin Campbell, Wolfgang Grieskamp and Margus Veanes. Testing concurrent object-oriented systems with Spec Explorer. In *FM'05: Proc. of Formal Methods.*

[29] Sylvain Conchon and Sava Krstic. Strategies for combining decision procedures. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Warsaw, Poland)*, volume 2619 of *Lecture Notes in Computer Science*, pages 537–553. Springer-Verlag, April 2003.

[30] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000.

[31] Datanamics. Db date generator. `http://www.datanamic.com/datagenerator/index.html`.

[32] IBM DB2. Test database generator. `www.ibm.com/software/data/db2imstools/db2tools/db2tdbg/`.

[33] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] Yuetang Deng, Phyllis Frankl, and David Chays. Testing database transactions with agenda. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 78–87, 2005.

[35] Roong-Ko Doong and Phyllis G. Frankl. The astoot approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3:101–130, April 1994.

[36] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438 –444, 1984.

[37] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, 2006.

[38] O. Grumberg E. M. Clarke and D. A. Peled. *ModelChecking.*

[39] Eclipse.org. The Eclipse Foundation open source community website. `http://www.eclipse.org/`.

[40] Larry Greenemeier Elena Malykhina and Paul McDougall. The high cost of data loss.`http://www.informationweek.com/showArticle.jhtml?articleID=183700367`. march 2006.

[41] Bassem Elkarablieh, Darko Marinov, and Sarfraz Khurshid. Efficient solving of structural constraints. In *ISSTA '08: Proc. of the International Symposium on Software Testing and Analysis.*

[42] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 151–162, 2007.

[43] S. Estévez-Martín, A. J. Fernández, T. Hortalá-González, M. Rodríguez-Artalejo, F. Sáenz-Pérez, and R. del Vado-Vírseda. A proposal for the cooperation of solvers in constraint functional logic programming. *Electron. Notes Theor. Comput. Sci.*, 188:37–51, July 2007.

[44] MySQL Forge. Random query generator. `http://forge.mysql.com/wiki/RandomQueryGenerator/`. 2009.

[45] Stephan Frank, Petra Hofstedt, and Dirk Reckmann. Solution strategies for multi-domain constraint logic programs.

[46] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, and Lorenzo Strigini. Evaluating testing methods by delivered reliability. *IEEE Trans. Softw. Eng.*, 24:586–601, August 1998.

[47] M.R. Frias, J.P. Galeotti, C.G.L. Pombo, and N.M. Aguirre. Dynalloy: upgrading alloy with actions. In *Proc. of the 27th International Conference on Software Engineering, 2005.*

[48] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[49] Svetoslav Ganov, Sarfraz Khurshid, and Dewayne E. Perry. Annotations and domain specific solvers for alloy, Submitted for publication to ASE 2011.

[50] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *17th International Symposium on Formal Methods (FM)*, June 2011.

[51] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proc. of the ACM SIGPLAN conference on Programming language design and implementation.*

[52] Eugene Goldberg and Yakov Novikov. Berkmin: A fast and robust sat-solver. *Discrete Appl. Math.*, 155:1549–1561, June 2007.

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

[54] Google Inc. Google Maps API Web Services. `http://code.google.com/apis/maps/documentation/webservices/`.

[55] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 206 –215, July 1988.

[56] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from java classes. In *In ECOOP*, pages 431–456. Springer, 2003.

[57] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[58] Petra Hofstedt. Cooperating constraint solvers. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 520–524, London, UK, 2000. Springer-Verlag.

[59] Petra Hofstedt and Peter Pepper. Integration of declarative and constraint programming. *Theory Pract. Log. Program.*, 7:93–121, January 2007.

[60] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[61] Kenneth Houkjaer, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246. VLDB Endowment, 2006.

[62] Radu Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 22–41, London, UK, 2002. Springer-Verlag.

[63] Daniel Jackson. Alloy: A lightweight object modeling notation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*.

[64] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Founda-*

tions of software engineering: twenty-first century applications, SIGSOFT '00/FSE-8, pages 130–139, New York, NY, USA, 2000. ACM.

[65] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis.* 2006.

[66] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 730–733, New York, NY, USA, 2000. ACM.

[67] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. August 2000.

[68] JUnit.org. Resources for Test Driven Development. `http://www.junit.org/`.

[69] Sarfraz Khurshid and Darko Marinov. Checking Java implementation of a naming architecture using TestEra. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.

[70] Sarfraz Khurshid and Darko Marinov. Using TestEra to check the Intentional Naming System of Oxygen. In *MIT Student Oxygen Workshop*, Gloucester, MA, July 2001.

[71] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 2004.

[72] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03: Proc. of 9th Conference on Tools and Algorithms for Construction and Analysis of Systems.*

[73] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

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

[75] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Jml: A notation for detailed design, 1999.

[76] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN'01: Proc. of the 8th international SPIN workshop on Model checking of software.*

[77] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Proc. of Static Analysis Symposium*, 2000.

[78] Nikolai Tillmann Margus Veanes and Jonathan de Halleux. Qex: Symbolic sql query explorer. In *Microsoft Research Technical Report MSR-TR-2009-2015, October 2009*.

[79] Mircea Marin and Tetsuo Ida. Collaborative constraint functional logic programming system in an open environment. In *IEICE Transactions on Information and Systems*, pages 223–230, 2003.

[80] Sun Microsystems. Java 2 Platform, Standard Edition, v1.3.1 API Specification. `http://java.sun.com/j2se/1.3/docs/api/`.

[81] Aleksandar Milicevic, Sasa Misailovic, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 771–774, Washington, DC, USA, 2007. IEEE Computer Society.

[82] Vilas Jagannath Sarfraz Khurshid Viktor Kuncak Milos Gligoric, Tihomir Gvero and Darko Marinov. Test generation through programming in UDITA. In *ICSE'10: Proc. of 32nd International Conference on Software Engineering*.

[83] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Design Automation Conference, 2001. Proceedings*, 2001.

[84] Sowmiya Narayanan. Clustered test execution using Java PathFinder. Master's thesis, Unniversity of Texas at Austin, 2009.

[85] Vidya Narayanan. Milao: A novel framework for mixed imperative and declarative formulation and solving of structural constraints. Master's thesis, Unniversity of Texas at Austin, 2009.

[86] Peter Neubauer. B-trees: Balanced tree data structures.`http://www.bluerwhite.org/btree/`. 1999.

[87] Hideaki Nishihara, Koichi Shinozaki, Koji Hayamizu, Toshiaki Aoki, Kenji Taguchi, and Fumihiro Kumeno. Model checking education for software engineers in japan. *SIGCSE Bull.*, 41:45–50, June 2009.

[88] Oracle Corp. Oracle database engine. `http://www.oracle.com/technology/software/products/database/index.html`.

[89] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA'07: In Proc. of the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion.*

[90] Meikel Poess and Jr. John M. Stephens. Generating thousand benchmark queries in seconds. In *VLDB: Proceedings of the Thirtieth International Conference on Very Large Databases*, pages 1045–1053, 2004.

[91] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *FSE'03:*

Proc. of the ACM SIGSOFT symposium on The Foundations of Software Engineering.

[92] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic software, 2003.

[93] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *FSE'05: Proc. of 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering.*

[94] Hossein M. Sheini and Karem A. Sakallah. A progressive simplifier for satisfiability modulo theories. In *In Proc. SAT06, volume 4121 of LNCS*, pages 184–197. Springer, 2006.

[95] Donald R. Slutz. Massive stochastic testing of sql. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 618–622. Morgan Kaufmann, 1998.

[96] Yannis Smaragdakis, Christoph Csallner, and Ranjith Subramanian. Scalable automatic test data generation from modeling diagrams. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 4–13, New York, NY, USA, 2007. ACM.

[97] Een N. Sorensson, N. An extensible sat-solver. In *In: Proc. 6th Int. Conf. Theor. and Applications of Satisfiability Testing (SAT)*, page

502518, 2003.

[98] MySql Open source database. `http://www.mysql.com/`.

[99] R. Rivest T. Cormen, C. Leiserson and C. Stein. *Introduction to Algorithms*. MIT-Press and McGraw-Hill, second edition, 2001.

[100] Yasuyuki Tahara, Nobukazu Yoshioka, Kenji Taguchi, Toshiaki Aoki, and Shinichi Honiden. Evolution of a course on model checking for practical applications. *SIGCSE Bull.*, 41:38–44, June 2009.

[101] G Tassey. The economic impacts of inadequate infrastructure for software testing. In *National Institute of Standards and Technology. `http://www.informationweek.com/showArticle.jhtml?articleID=183700367`*, May 2002.

[102] The hsqldb Development Group. HSQL database engine. `http://www.hsqldb.org/`.

[103] Cesare Tinelli and Christophe Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1):291–353, January 2003.

[104] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. pages 632–647. 2007.

[105] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[106] Engin Uzuncaova. *Efficient specification-based testing using incremental techniques.* PhD thesis, University of Texas at Austin, 2008.

[107] Engin Uzuncaova and Sarfraz Khurshid. Constraint prioritization for efficient analysis of declarative models. In *In Proc. of the 15th Intl Symposium on Formal Methods*, 2008.

[108] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 3–, Washington, DC, USA, 2000. IEEE Computer Society.

[109] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proc. of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[110] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[111] Visser et al. Symbolic execution for software testing in practice preliminary assessment. In *In Proc. of the 33rd International Conference on Software Engineering, May, 2011. (To appear).*

[112] David Willmor and Suzanne M. Embury. An intensional approach to the specification of test cases for database applications. In *ICSE '06:*

Proceeding of the 28th international conference on Software engineering, pages 102–111, New York, NY, USA, 2006. ACM.

[113] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 196–205, Washington, DC, USA, 2004. IEEE Computer Society.

[114] Tao Xie and David Notkin. Automatically identifying special and common unit tests for object-oriented programs. In *ISSRE'05: Proc. of the 16th IEEE International Symposium on Software Reliability Engineering.*

[115] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *FSE'07: Proceedings of the ACM SIGSOFT symposium on The Foundations of Software Engineering.*

[116] Pixley C. Yuan J. and Aziz A. *Constraint-Based Verification.* Springer, 2006.

Vita

Shadi was born in Venezuela on July 1st 1985. He received the Bachelor of Science degree in Computer Science with distinction from the American University of Beirut in Lebanon in 2006. In 2007, he joined the University of Texas at Austin pursuing his doctorate degree. During his studies at UT, he received his Masters of Science Degree in Software Engineering in 2009. His major fields of interest are automated software testing and verification. Shadi worked fulltime at Clifton Myers Interprises Offshore in 2006 at Beirut before joining UT, then he interned at National Instrument in Austin, Summer 2008, Yahoo! Inc. in Sunnyvale, Summer 2009, and at Google Corp., Summer 2010.

Permanent address: 923 E. 41st St.
Austin, Texas 78751

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.