A Lightweight Approach to Model-Based Acceptance Testing Using Alloy Specifications

Darioush Jalalinasab, Masoumeh Taromirad, and Raman Ramsin, *Member, IEEE*

Abstract—Testing is an important part of the software development process, estimates show up testing uses up to 50% of resources on projects. Unfortunately, determining what tests to write is a difficult task and is often left to the discretion and experience of the individual developer.

Model-Based Testing (MBT) avoids this bias by taking a systematic approach to test generation. However, MBT requires models not usually developed in normal development processes. These models must then be kept up to date with changes to code, a practice agile development approaches often avoid.

This paper shows how a minimal set of structural and behavioral models can be used to automatically generate acceptance tests. We begin by choosing models intuitive and familiar to ordinary practitioners, then describe the process of transforming them to executable test cases.

A non-trivial case study demonstrates the feasibility of the proposed approach. We compare conventional test quality metrics (e.g. line coverage, mutation score) to those of manually developed tests. Finally we discuss faults discovered in the case study as a result of using this approach.

Index Terms—Model Based Testing (MBT), Model Driven Development (MDD), Software Testing, Minimal Modelling, Static Analysis

1 Introduction

Testing plays an invaluable role in software development as a quality assurance measure, estimates show up testing uses up to 50% of resources on projects [1]. However, tests are costly to develop, and they can become complex and difficult to maintain [2]. Unfortunately, determining what tests to write is a difficult task and is often left to the discretion and experience of the individual developer.

Automated Model-Based Testing (MBT) offers a promising solution to this problem [3]: automatically generating tests from high-level behavioural models side-steps the need to write and maintain tests manually. This generative approach to testing reduces the cost of creating and maintaining tests. The models are at a higher level of abstraction (making changes easier), and automated generation process obviates the need of making a single change in many test scripts. As an added benefit, MBT takes a systematic approach to test generation, so it is easier to determine what is covered more confidently.

However, practices prescribed by MBT are not particularly synergetic with those prescribed by lightweight software development processes [2], as they generally discourage the proliferation of models and dealing with the burden of keeping them up-to-date with code changes. MBT requires models different than those developed during general software analysis and design, or may require additional details not typically included (e.g., describing properties in temporal logic languages [4]), thus making things seem worse.

This research introduces a minimalistic set of structural and behavioural models that can be used as a basis for MBT in the context of a lightweight software development process. The models are intuitive and accessible to practitioners: no particular modelling or model-based testing knowledge is required. These models are either normally produced during the development process, or have been kept very simple, if they are needed only for test generation.

Lightweight methodologies typically focus on unit testing, in part due to associated technical difficulties with acceptance testing (i.e, end to end testing) [5], [6]. For example, statistics show that requirements specifications are still commonly captured in the form of documents. In this context, existing studies on the opportunity of benefiting from MBT in lightweight approaches and vice versa (e.g., [7], [8], [9], [10]), by large, have focused on unit testing. Our approach focuses on acceptance testing. A comprehensive study on the application of MBT in lightweight processes has been provided in our earlier work [11].

Automated acceptance testing techniques are generally either (1) script-based, or (2) based on system behavioural models. In *script-based techniques*, developers convert informal requirements, one by one, from natural language to executable tests by writing test cases in a scripting language or library specified by a framework. The FIT [1] and Cucumber [2] frameworks, used in Acceptance Test Driven Development (ATDD) [12] are examples of such techniques. This approach suffers from a low abstraction level and results in brittle, high-maintenance tests which must be

- *D. Jalalinasab was with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. Email: jalalinasab@ce.sharif.edu.*
- *M. Taromirad and R. Ramsin are with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. Email: {m.taromirad,ramsin}@sharif.edu.*

Manuscript received July 23, 2018

1. FitNesse http://fitnesse.org
2. Cucumber http://cukes.info

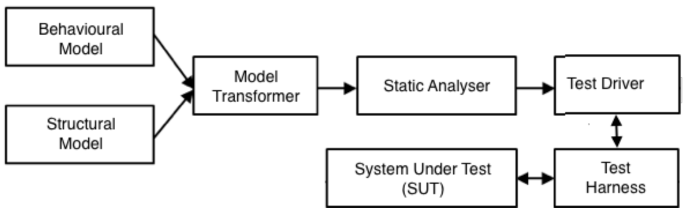


Fig. 1. Overall structure of the proposed framework.

developed and maintained individually. Techniques *based on system behavioural models* generate tests from formal models of system behaviour, such as use cases (e.g., [13], [14], [15]). These models implicitly define many valid execution paths for the system under test (SUT). Deriving tests based on behavioural models addresses the abstraction level problem of script-based testing. However, the require models can be complex and inappropriate for lightweight processes.

In an aim to combine the best of both approaches, this paper presents an acceptance testing technique based on system behavioural models, using simple models, thus remaining practical for application in lightweight processes.The proposed method uses class diagrams and use cases, that are considered as the most useful models in software development [16], [17].

why is this relevant?

Fig. 1 shows the overall structure of the proposed testing framework that has been tailored to data-oriented systems. Tests are generated from (1) a structural model of the domain (class diagram), (2) use cases, as the behavioural model, and (3) a static analyser (i.e., Alloy [18]). We use Alloy as a specification language [18], and the Alloy Analyser as the analysis engine. Alloy is a formal specification language based on first order logic, optimized for automated analysis. The models are automatically translated into Alloy specifications. The solutions to the Alloy models are translated into executable test cases which the test driver and test harness components use to drive the SUT.

Contributions. This paper makes the following contributions:

- *Lightweight model-based acceptance testing.* We develop a lightweight testing framework with detailed description on how the approach is used from testers' perspective.
- *Behavioural modelling DSLs.* We develop a set of simple DSLs for behavioural modelling.
- *Implementation.* We provide the DSL used to create test models, the code that performs the model transformation to Alloy's native language, and the test driver and harness code necessary to execute test cases on the SUT.
- *Case-study.* We evaluated the proposed framework's applicability by demonstrating the feasibility of applying it to a medium-sized business application with reasonable resources. We evaluated the effectiveness of the approach by comparing conventional test quality metrics (line coverage and mutation score) to manually developed tests and also discuss bugs uncovered during the case study.

is this really a research contribution?

Outline. The remainder of this paper is organized as follows. Section 2 explains the running example used to illustrate the testing framework. Section 3 provides an overview of the proposed testing framework. Sections 4 and 5 describe the details of creating test models and test generation and execution, respectively. Section 6 presents the evaluation of the approach. Finally, the paper concludes with a discussion of limitations, and an outline of the related research and future work.

are we going to share

2 Running Example

The paper will refer to a simple library system as a running example to illustrate the framework and relevant models. In this simple library system, each book has a unique name and there is only one copy of each book. Each member can borrow at most two books at the same time. The use cases for the system follow:

1) Add Book
   - Input: book name
   - Pre-condition(s): the name is not empty and the system does not have a book with the specified name.
2) Add Member
3) Borrow Book
   - Pre-condition(s): the book is not already borrowed and the member has borrowed at most one book.
4) Return Book
5) Delete Book
   - Pre-condition(s): the book is not already borrowed.
6) Delete Member
   - Pre-condition(s): the member has no borrowed books.

3 Framework Overview

Fig. 2 shows an overview of the proposed testing framework.

From testers' perspective, the following steps are involved in using our framework:

1) Developing a domain model that is specified using EMF [19].
2) Describing use cases with the internal DSL.
3) Describing the initial state of the system described with the given DSL.
4) (Optionally) describing invariants of the domain objects.
5) Identifying and optionally partitioning the input data types using the internal DSL.
6) Specifying the test scope.
7) Developing the test harness.
8) All of the above models are automatically translated into Alloy's native modelling language.
9) Alloy Analyser solves the model, yielding a set of execution traces. Each trace is a test case and includes the required input values.

10) The test driver executes each test case on the SUT using the test harness. The SUT is reset to the initial state before running the next test case.

The *primary models* include use cases (translated from natural language to the specified DSL) and an EMF class diagram. Additional *subsidiary models* describe the initial state of the SUT, and optionally describe invariants between object states. The tester develops an SUT specific *test harness* to mediate interaction between the test generation framework and the SUT. The test harness consists of a *data generator*, a *command executor*, and an *inspector*.

Next, we discuss the components of the framework and the required models.

3.1 Structural Model

The framework requires the tester to provide an EMF class diagram, conforming to Ecore metamodel. This structural model includies domain entities essential to analysis. These entities typically represent the SUT's persistent data. In case the internal state of a domain object is relevant to the use cases, objects can optionally have a label. An *object label* is determined based on its internal data or the relations it has with other objects. Labels determine whether an object is allowed to participate as input to use cases (see Section 4.2). For example, an book may have a "Borrowed" or "Returned" label, and only *returned* books can participate in the borrow use case. Labels are similar to the "State" pattern described in [20].

Defining "object labels" avoids exposing internal representations to the model finder (making analysis of the model computationally infeasible) and yet retains a degree of flexibility in a modelling and verifying behaviours of the SUT.

3.2 Behavioural Model

The tester develops a behavioural model of the SUT based on system use cases. This model describs effects of the use case on domain objects. Generally, use cases do not have the required formalism which makes them insufficient for automatic test generation [21]. Our approach utilizes an internal domain specific language (DSL) within Java for describing use cases. This DSL provides the required formalism, yet remains familiar for the developer; the behavioural model is constructed via chaining methods in a language that the developer is already familiar with (i.e., Java).

Benefits of using a DSL as a modelling language include:

- developers are already familiar with the programming language. This allows existing IDEs and tools to be used for creating and maintaining the model,
- the use cases are modelled using textual format which is similar to programming and hence, creating behavioural model by our DSL would be familiar to developers (e.g., by using familiar text editors), and
- other testing frameworks such as JUnit [21], jMock[3], and Selenium [4], have been successful in using DSLs for specifying tests.

3. jMock http://jmock.org
4. Selenium - https://www.seleniumhq.org/

The introduced DSL supports modelling of

1) input parameters for each use case,
2) pre-conditions for executing a use case,
3) different execution paths for each use case, and
4) the effects of use case execution on domain objects (post-conditions).

Use cases are inherently not object-oriented, however we assume that mapping use cases to the domain model (e.g., mapping the input parameters and describing the effects of executing use cases) is relatively straightforward for data-intensive systems. Object labels can be used in pre- and post-conditions to help the tester gain some flexibility in this mapping.

3.3 Test Data

Generating test data is one the main issues in automatic test generation [22]. Input domain partitioning is a well known approach for generating test data [23]. This approach partitions asks the tester to partitoin the input domain such that it is enough to test the system with only a single representative of every and each partition. This approach is particularly suited to the systems not involving complicated data manipulation and processing.

The framework allows the tester to specify input data types and their partitioning in a Java based DSL. The tester then provides a data generator (as executable code) for each partitioning according to the architecture prescribed by the framework.

3.4 Subsidiary Models

In addition to the aforementioned models, the testing approach relies on three additional subsidiary models.

3.4.1 Initial State

The initial state of the system is needed for model solving (static analysis). The objects present in the initial state and (optionally) their labels and relations are specified in our DSL.

3.4.2 Object Labels Rules

Certain constraints on relations between objects (e.g., number of involved objects in a relation) cannot be specified in the behavioural model DSL, as they do not pertain to a single use case. The framework includes a DSL for defining such dependencies separately.

3.4.3 Test Goals

The tester specifies the test goal and the search scope (used in static analysis). There are two built-in test goals. Solutions obtained using the first goal are traces representing successful execution of use cases. Solutions obtained using the second goal attempts to run use cases when the pre-conditions are not met, in an attempt to discover bugs. The search scope determines the maximum length of execution traces, and is an inherit limitation to our apptoach, as we utilize a finite model generator (i.e., Alloy). The tester has the flexibility to force the inclusion or exclusion of a specific use case from execution traces. This may be useful, eg, to force a login in data-driven systems where most functionality is only accessible after logging in. Section 5.8 provides more detail on how test goals and search scope are defined.

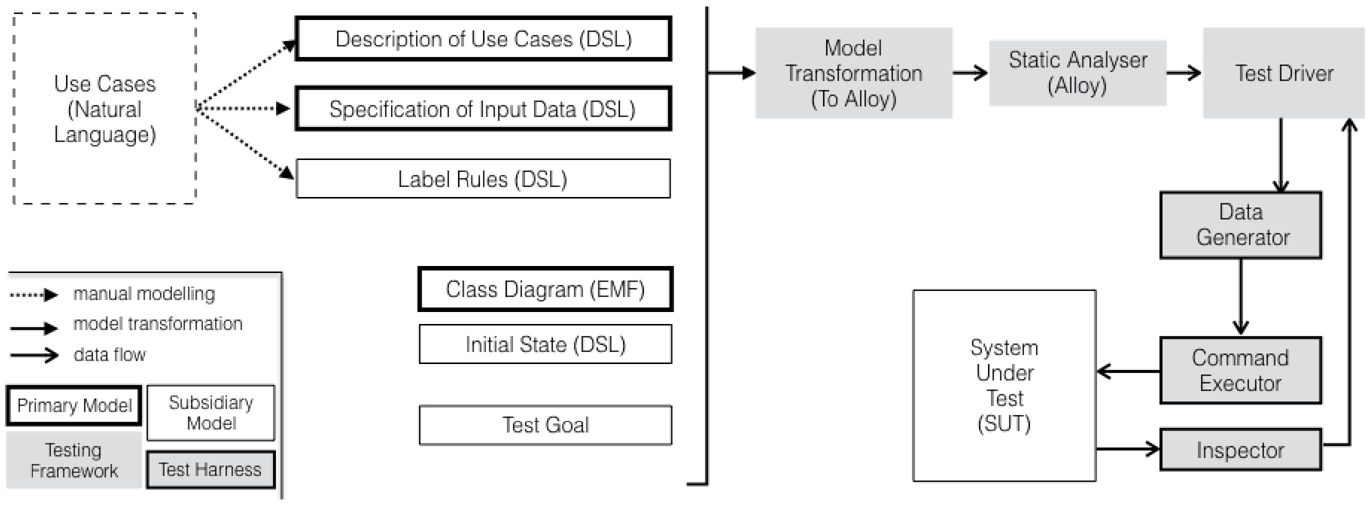


Fig. 2. The architecture of the proposed testing framework.

3.5 Model Transformer: to Alloy

This component translates the structural and behavioural models into Alloy's native modelling language.

3.6 Static Analyser: Alloy

The static analyser solves for a set of execution traces based on the the generated Alloy models. Alloy has a high-level specification language compared to CTL[5], LTL[6], and first-order logic model analysis tools. The lower level of abstraction makes such languages not suitable targets for translating from high-level descriptions of system behavior.

Alloy has its own specification language for software specification which is built on the set theory and relational calculus. Most of other approaches based on set theory (e.g., Z [24]) have weak execution support. One advantage of using Alloy is its ability to enumerate all possible solutions for a given set of constraints for which it is also called a Model Finder. Alloy converts the specification to a SAT problem[7], which is then solved using a SAT solver. The solution (if exists) is mapped back to a relational model. Using SAT solvers gives rise to the need for a finite search space that limits the solver to finding small instances of the solution space. Searching within finite bounds is justified by the "small scope hypothesis", the fundamental premise that underlies Alloys analysis. The hypothesis claims "most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or small instance makes no difference. So if the analysis considers all small instances, most flaws will be revealed" [25]; systems that fail on large instances almost always fail on small ones with similar properties. This claim has not been proved and is fundamentally a claim about the assertions that arise in practice.

5. CTL Computation Tree Logic
6. LTL Linear Temporal Logic
7. Boolean SATisfiability problem

3.7 Test Driver

The test driver uses the test harness to executes solutions to the Alloy model on the SUT. Each solution is a test case that includes: (1) a number of use cases with their input parameters, and (2) a description of the final expected *state* of the SUT if the test case is run from the initial state. The state of the SUT consists of a snapshot of the objects of the domain model at a particular moment.

The driver instructs the harness to reset the SUT to its initial state before running each test case. The driver uses the harness to run the use cases one by one (after generating required input parameters). After the execution of each use case, the current state of the SUT is compared to the expected state of the system. Inconsistencies result in a failed test, and test execution stops. Otherwise, the next use case is executed. Fig. 3 shows this workflow.

3.8 Test Harness

The test harness is an SUT-specific piece of code developed by testers to utilise the testing framework. This component mediates all communications between the test driver and the SUT, including executing a use case and observing SUT behaviour. The test harness includes:

1) the *command executor*, which is responsible for executing use cases on the SUT. This component also includes machinery to generate representative data points from specified input partitions as needed.
2) the *inspector*, which provides a list of domain objects and their labels to the test driver.

4 Creating Test Models

The framework relies on the tester to provide the primary and subsidiary models. This section describes how these required models are created or generated in the context of the example library system.

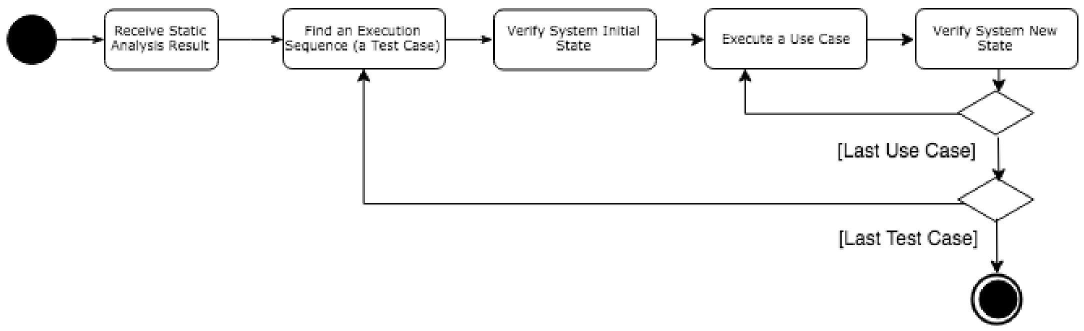
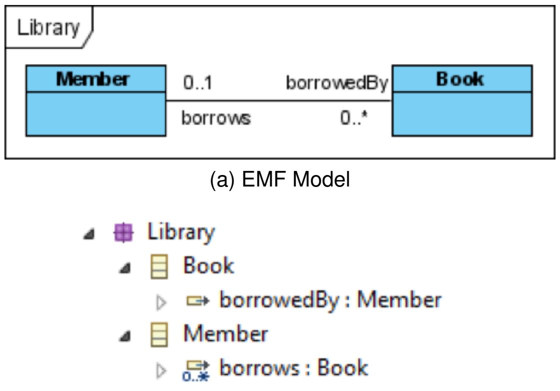

Receive Static Analysis Result
Find an Execution Sequence (a Test Case)
Verify System Initial State
Execute a Use Case
Verify System New State
[Last Use Case]
[Last Test Case]

Fig. 3. The workflow of the test driver.

4.1 Structural Model

The structural model consists of a simplified EMF class diagram. Only the structure and relations are necessary, the internal data fields of classes are not modelled. Fig. 4 shows the structural model of the example library system.



(a) EMF Model

(b) Class Diagram

Fig. 4. Structural model of the example library system.

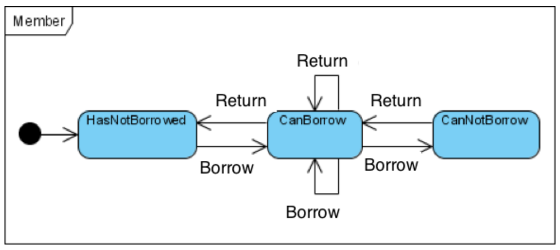
The Eclipse IDE[8], is capable of tranlating the diagram to Java code. This generated code is referenced in the DSL defining the behavioural model. The EMF model is also transformed into a partial Alloy specification (See Section 5.3 for details of this transformation).

4.2 Behavioural Model

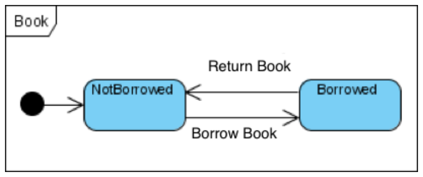
The behavoiral model allows the tester to describe the effect of each use case on domain objects in terms of their labels and their relations. Labels can be identified by considering the state-based behaviour of the system, as implied by use cases. For example, in the library system, the class `Book` has two labels: *Borrowed* and *NotBorrowed*, and the class `Member` has three labels: *HasNotBorrowed*, *CanBorrow*, and *CanNotBorrow*. Fig. 5 shows the labels and illustrates how they relate to use cases. Fig. 6 shows the description of "Borrow Book" use case using the DSL.

Each use case is defined with a Java class with a `@SystemOperation` annotation. A use case description specifies one or more execution paths that are defined as a method using the signature shown in line 3 of Fig. 6. These methods are annotated as `@Description` and return an `ModelExpectations` object. The core of the behavioural modelling is performed by configuring this (line 5 to 12).

8. Eclipse - http://eclipse.org



(a) Member



(b) Book

Fig. 5. State diagram showing the object labels in the library system.

First (lines 5-7), the parameters and the pre-conditions of the use case are defined. Line 5 indicates the use case operates on an object of type `BOOK` which must have the `NotBorrowed` label, and names it `thisBook`). This is done by chaining the `parameter` and `inState` methods. Lines 6 and 7 describe a parameter of type `Member` and its permissible labels in a similar fashion. All-caps class names directly reference the Java code generated by EMF.

The second part (line 9 to 12) describes the post-conditions of the execution path, and includes changes to object labels. A specific state (line 9) or an arbitrary state (line 10) may be specified. Changes in relations are also described (line 11).

4.3 Test Data

This sections describes how data is generated for use cases that need input data. As mentioned in Section 3.3, we use the input partitioning approach for data generation. The partitioning of input parameters are defined by the tester, based on the system requirements, and are introduced to the framework. For example, the "Add Book" use case requires

```
@SystemOperation
public class BorrowBook {
  @Description public ModelExpectations describe(Context context) {
    return context.modelingExpectations()
        .parameter("thisBook", BOOK).inState("NotBorrowed")
        .parameter("thisMember", MEMBER)
        .inState("CanBorrow", "HasNotBorrowed")

        .expectStateChange("thisBook", "Borrowed")
        .allowStateChange("thisMember")
        .expectAddToRelation(MEMBER__BORROWS,
          "thisMember","thisBook");
  }
}
```

Fig. 6. The behavioural modelling for "Borrow Book" use case using our DSL.

the input parameter *bookName* to be automatically generated. A book name is a string that has "empty", "duplicate", and "unique" partitions.

An input data type is defined by implementing the DataFactory interface in which each partition is represented by a method annotated as @Partition and returns an object of type PartitionDescription. Fig. 7 shows the definition of the BookName data type and its partitions, namely "empty", "unique", and "nonUnique" (lines 31-41). Each partition extends PartitionDescription and implements the generate method that is the main method for generating data for that partition. For example, lines 2-6 introduce the "empty" partition that returns an *empty* string in its generate method.

this is way more de-tail than any-one would ever care about, also the code isn't that use-ful, can we cut this?

```
01 public class Bookname implements DataFactory {
02   public class EmptyPD extends PartitionDescription {
03     public Object generate(SoftwareSystem system) {
04       return "";
05     }
06   }
07
08   public class UniquePD extends PartitionDescription {
09     int lastNumber = 1;
10
11     public Object generate(SoftwareSystem system) {
12       return "book-" + this.lastNumber++;
13     }
14   }
15
16   public class NonUniquePD extends PartitionDescription {
17     @Override
18     public List<DataParam> getDataParams() {
19       ArrayList<DataParam> retVal = new ArrayList<DataParam>();
20       retVal.add(new DataParam(BOOK, "book"));
21       return retVal;
22     }
23
24    public Object generate(SoftwareSystem system, DomainParam dp) {
25      LibrarySoftwareSystem sys = (LibrarySoftwareSystem) system;
26      Book book = sys.instance.books.getBookById(dp.lookup());
27      return book.getName();
28    }
29   }
30
31   @Partition public PartitionDescription empty() {
32     return new EmptyPD();
33   }
34
35   @Partition public PartitionDescription unique() {
36     return new UniquePD();
37   }
38
39   @Partition public PartitionDescription nonUnique() {
40     return new NonUniquePD();
41   }
42 }
```

Fig. 7. Defining BookName data type and its partitioning.

If the data generation requires access to the domain objects, this dependency is specified in the getDataParams method that returns a list of required objects. This list is then passed to the generate method, in addition to the SoftwareSystem object. The data generation for the "nonUnique" partition, in Fig. 7, requires a book name that already exists in the system. This is done in lines 18-21 and the returned object that is used in the generate method, lines 24-28, as the generated (non-unique) book name. Fig. 8 shows how a data type and its partitioning are used in modelling a use case. In line 4, calling the method input and referring to the BookName data type define a parameter for the use case. The method inPartition restricts the permissible partitions for that parameter.

4.4 Test Harness

Testers implement the SoftwareSystem interface to introduce an SUT to the framework. This object provides methods for executing commands and inspection. The SoftwareSystem interface includes a reset method, which is responsible for bringing the system into its initial state.

4.4.1 Inspection

The test harness must return a list of the existing objects and their associated labels for each class in the domain model. Each object has a unique identifier (ID) managed by the test harness and the SUT. Accordingly, an abstract super class (AbstractDomainObjInspector) is provided that is inherited by the test harness. Fig. 9 shows an example implementation of this super class and the inspector for the Member class.

The @Inspector annotation with the name of the target type (line 1) allows the framework to identify the inspector for the target domain class (e.g., "Member" in Fig. 9) . The constructor, line 5-7, takes an object implementing SoftwareSystem. The getObjectState returns the labels of the object with the input ID, and the getObjectList returns the IDs of all the objects of the target type. The body of these two methods are implemented for the SUT.

way too much de-tail.

4.4.2 Command Execution

The test harness implements the Command interface (Fig. 10) for each use case so that it is possible to run the test

```
01 @SystemOperation public class AddBook {
02   @Description public ModelExpectationsInterface describe(Context context){
03     return context.modelingExpectations()
04       .input("bookName", new Bookname()).inPartition("unique")
05       .expectNew(BOOK, "NotBorrowed");
06   }
07
08   public Command execute(final InputParam _bookName) {
09     return new Command() {
10       @Override
11       public void execute(SoftwareSystem system) {
12         String bookName = (String) _bookName.generate(system);
13         LibrarySoftwareSystem library = (LibrarySoftwareSystem) system;
14         library.instance.books.addBook(bookName);
15       }
16     };
17   }
18 }
```

Fig. 8. Using the data types and their partitioning in use case modelling.

```
@Inspector(type="Member")
public class MemberInspector extends AbstractDomainObjInspector {

  LibrarySoftwareSystem library;
  public MemberInspector(SoftwareSystem system) {
    this.library = (LibrarySoftwareSystem) system;
  }

  @Override
  public String getObjectState(int appId) {
    Member m = library.instance.members.getMemberById(appId);
    if (m.getBorrowedCount() == 0 ) {
      return "HasNotBorrowed";
    } else if (m.getBorrowedCount() < 2) {
      return "CanBorrow";
    } else {
      return "CanNotBorrow";
    }
  }

  @Override
  public Set<Integer> getObjectList() {
    HashSet<Integer> retval = new HashSet<Integer>();
    for (Member m : library.instance.members.getAll()) {
      retval.add(m.getId());
    }
    return retval;
  }
}
```

Fig. 9. An example inspector, for the "Member" class.

cases on the SUT. Each use case description class has an `execute` method that returns an object of type `Command`. Calling this method would not execute the use case, it just creates a `Command` object which, later on, is used for test case execution by the framework during the test execution process.

```
public interface Command {
  public void execute(SoftwareSystem system);
}
```

Fig. 10. `Command` interface.

An implementation of the `command` method, for the "Borrow Book" use case, is shown in Fig. 11, line 7-20. Considering the use case description in Fig. 6, the use case has two parameters which are passed to the `execute` method as objects of type `DomainParam`. These objects have a method, called `lookup`, which returns the ID of the object. The IDs are required in use case execution.

Fig. 8 shows how a data type is used in defining the implementation of the `execute` method (or creating the `command` object) for the "Add Book" use case which requires an input parameter. In line 8, the input parameter is passed to the `execute` method, as an object of type `InputParam` which has a `generate` method (line 12). Calling this method will call the `generate` method of the related partition and hence, results in generating appropriate test data.

4.5 Subsidiary Models

This section elaborates on the subsidiary models introduced in Section 3.4. The (user defined) test goals are directly specified in Alloy models and thus not discussed further here.

4.5.1 Initial State

This model specifies the objects in the initial state, with their labels and the relations. Fig. 12 specifies an initial state for the example library system, consisting of one book and one member that has borrowed the book.

4.5.2 Object Labels Rules

Some constraints on relations (e.g., number of involved objects in a relation) cannot be specified in the behavioural model. For example, in the modelling of the post-conditions of the "Borrow Book" use case (Fig. 6), the label of the object `thisMember` can not be determined a-priori to use case execution. Such dependencies are defined a subsidiary model. Fig. 13 illustrates the rules applicable to the "Member" class and its labels. In line 3, the `setFor` method specifies the class the rules apply to. Line 4 indicates that if the given relation (`MEMBER_BORROW`) is empty then the label is `HasNotBorrowed`. Similarly, line 5 says if the cadinality of the given relation (`MEMBER_BORROW`) is equal or greater than two then the label is `CanNotBorrowed`. Line 6 indicates that the label would be `CanBorrow` if none of the other cases apply.

4.6 Introducing the Models to the Framework

Each category of models described above is introduced to the framework by placing them in a package together. The testing framework determines the name of this package by

```
@SystemOperation
public class BorrowBook {
  @Description public ModelExpectationsInterface describe(Context context) {
    // snip
  }

  public Command execute(final DomainParam _book, final DomainParam _member) {
    return new Command() {
      public void execute(SoftwareSystem system) {
        final int book = _book.lookup();
        final int member = _member.lookup();

        // SUT Specific code
        LibrarySoftwareSystem library = (LibrarySoftwareSystem) system;
        library.instance.books.getBookById(book).borrow(
            library.instance.members.getMemberById(member));
        // SUT specific code
      }
    };
  }
}
```

Fig. 11. Implementation of the `Command` interface for "Borrow Book" use case.

```
public class InitClass {
  @InitConditions public void conditions(InitContext context) {
    context.addObject(MEMBER, "CanBorrow", "member");
    context.addObject(BOOK, "Borrowed", "book");
    context.addToRel(MEMBER__BORROWS, "member", "book");
  }
}
```

Fig. 12. An initial state model for the library system.

```
public class MemberStateRules {
  @RuleDef public Rules rules(RuleContext context) {
    return context.rules().setFor(MEMBER)
      .whenRelCount(new None(), MEMBER__BORROWS).thenInState("HasNotBorrowed")
      .whenRelCount(new Gte(2), MEMBER__BORROWS).thenInState("CanNotBorrow")
      .elseInState("CanBorrow");
  }
}
```

Fig. 13. Modelling of rules applied in "Member Class".

calling methods of the `SUT` interface (Fig. 14), then finds the required classes (and methods) using Java reflection.

```
public interface SUT {
  String getRequirementsPackage();
  EPackage getEInstance(); // EMF Model
  SoftwareSystem getSystem();
  String getDataPackage();
  String getHarnessPackage();
  String getInitPackage();
  String getRulePackage();
}
```

Fig. 14. `SUT` interface.

5 Test Generation and Execution

The previous section introduces the required models and explains how to produce them. This section explains the test generation and execution workflow. The testing framework has been built upon Alloy [18] and the Alloy Analyser (AA) [26] with the aim of testing a system at the use case level (i.e., acceptance testing). AA provides automatic analysis of Alloy specifications by generating instances that satisfy the constraints expressed in the specification.

The key idea behind the framework is to use Alloy to express the structural and behavioural model of the system specifying the use cases, the invariants of inputs and outputs, and pre-/post-conditions of executing use cases on the system under test. In this context, all the test models, introduced in the previous section, are automatically translated into Alloy specifications which are then used by AA to automatically generate, for a given scope, all non-isomorphic [27] instances for that specification. Next, the testing framework translates these instances to real execution paths and concrete inputs, which form the executable test cases for the SUT.

In the following, we first describe the basics of the Alloy specification language and the Alloy Analyser; details can be found in [18], [25], [26]. Then, we explain the model transformations into Alloy.

5.1 Alloy

Alloy is a strongly typed language that assumes a universe of atoms partitioned into subsets, each of which has a basic type. An Alloy specification is a sequence of paragraphs that can be of two kinds: signatures, used for construction of new *types*, and a variety of formula paragraphs, used to record *constraints*.

A *signature* paragraph represents a basic (or uninterpreted) type and introduces an independent top-level set. A signature declaration may include a collection of relations (that are called *fields*) along with the types of the fields and constraints on their values; a *field* represents a set relation between the signatures. *Formula* paragraphs are formed from Alloy expressions, specifying constraints of the desired solutions, such as *facts* and *assertions*. A `fact` is a formula that takes no arguments and need not be invoked explicitly; it is always true. An assertion (`assert`) is a formula whose correctness needs to be checked, assuming the facts in the model.

The Alloy Analyser [26] is an automatic tool for analysing models created in Alloy. Given a formula and a *scope*–a bound on the number of atoms in the universe–the analyser determines whether there exists a model of the formula (that is, an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it.

Since first order logic is undecidable, the analyser limits its analysis to a finite scope. The analysis is based on a translation to a boolean satisfaction problem, and gains its power by exploiting state-of-the-art SAT solvers. The models of formulae are termed *instances* or *solutions*. Each atom is in a signature and the relations between atoms are instances of their fields.

5.2 Generated Alloy Model

The test models described in Section 3 are automatically translated into Alloy specifications consisting of six `modules`, that are defined in separate Alloy files (.als). These modules and their dependencies are shown in Fig. 15, and include:

1) `system_def_auto`: translation of the domain structural model (EMF diagram),
2) `operations_def_auto`: translation of the system behavioural model (use case descriptions) and definition of the labels,
3) `rules_def_auto`: translation of the object labels rules,
4) `init_def_auto`: translation of the initial state model,
5) `input_def_auto`: translation of the input partitionings, and
6) `main`: the main module specifying the test goals and the entry point for the static analysis (i.e., the starting predicate).

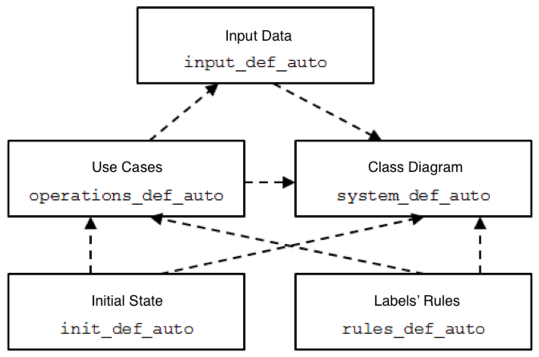


Fig. 15. Generated Alloy specifications and their dependencies.

The rest of this section describes the details of the model transformation for each module.

5.3 Structure Model

The EMF model of the system is automatically translated into an Alloy specification, using the Xpand[9] model-to-text transformation language. Fig. 16 shows the automatically generated Alloy specification for the structure of the example library system. Each class is translated into a signature. Each association is represented by a field in the respective signature (lines 9-11 and lines 17-19). The multiplicity of a relation affects the definition of the fields. Abstract signatures are used for modelling abstract classes. Class

```
01 one sig System {
02   book: Book -> State,
03   member: Member -> State,
04 }
05 abstract sig DomainObjState{}
06 abstract sig DomainObj {
07   state : DomainObjState -> State,
08 }
09 sig Book extends DomainObj {
10   rel_borrowedBy : Member lone-> State,
11 }
12 fact {
13   all s: State | rel_borrowedBy.s = ~(rel_borrows.s)
14 }
15 abstract sig BookState extends DomainObjState {}
16 fact {Book.state.State in BookState}
17 sig Member extends DomainObj {
18   rel_borrows : Book -> State,
19 }
20 fact {
21   all s: State | rel_borrows.s = ~(rel_borrowedBy.s)
22 }
23 abstract sig MemberState extends DomainObjState {}
24 fact {Member.state.State in MemberState}
```

Fig. 16. Alloy specification of the structure of the example library system.

inheritance is implemented using the inheritance concept in Alloy.

Alloy does not have a built-in notion of mutable state; it does not allow changes to the relations, whereas, in the context of testing, the domain model changes with the execution of use cases. Consequently, our approach models the concept of mutable state explicitly. A typical way to deal with this is defining a `state` signature and all the relations that would change during test executions are placed in a relation with an atom of `state`. The transformation adds the extension `-> State` at the end of the changing relations.

The `System` signature represents the system under test and tracks the objects that in the domain model at any time. The transformation defines a field for each concrete class in the domain model (e.g., Fig. 16, lines 1-4).

Object labels are modelled by the `DomainObjState` abstract signature, such that for each domain class, the transformation adds an abstract signature that extends `DomainObjState` and shows the labels for that class (e.g., lines 15-19). Additionally, all the signatures for domain classes inherit from the `DomainObj` abstract signature which contains the field defining the relation between atoms and their labels.

In order to constrain the state space objects not in the domain model cannot participate in relations and can not have label atoms. This is specified as a `fact` at the end of the structural Alloy specification. The fact for the example library system is depicted in Fig. 17.

5.4 Operations: Use Cases

The next step is to translate the use case descriptions into Alloy. Each use case is translated into a corresponding signature and a predicate. First, the signature is used for tracing the execution flow: a solution does not show which predicate in the specification has resulted in the solution. The test driver needs to know which predicate has been

9. Xpand http://eclipse.org/modeling/m2t/?project=xpand

```
fact {
  all local_name: Book, s: State |
   local_name not in System.book.s implies
   no local_name.state.s and
   no local_name.rel_borrowedBy.s
  all local_name: Member, s: State |
   local_name not in System.member.s implies
   no local_name.state.s and
   no local_name.rel_borrows.s
}
```

Fig. 17. The `fact` for confining the state space in the structure model of the example library system.

selected at any state. Using *conjunction*, an atom of the signature is attached to each predicate, so that it is possible to trace the execution flow. The use case signature contains fields corresponding to its parameters. Fig. 18 shows the signature for the “Return Book” use case. Lines 5-8 restrict the solution space.

```
1 sig ReturnBookOp extends Op {
2   returnBookOp_thisMember : one Member,
3   returnBookOp_thisBook : one Book,
4 }
5 fact { no disj op1, op2 : ReturnBookOp | {
6    op1.returnBookOp_thisMember = op2.returnBookOp_thisMember
7    op1.returnBookOp_thisBook = op2.returnBookOp_thisBook
8 }
```

Fig. 18. The signature for the “Return Book” use case.

Each use case predicate takes atoms representing the system state before and after the use case is executed. An atom of the signature of the use case provides access to the inputs. This predicate essentially defines the constraints for creating the subsequent state. Fig. 19 shows the predicate for “Return Book”. This predicate consists of the following parts: defining the domain of the input parameters, defining the pre-conditions, specifying the frame conditions, and defining the post-conditions.

```
01 pred borrowBookOp(s, s': State, op: BorrowBookOp) {
02   {
03     op.borrowBookOp_thisBook in System.book.s
04     op.borrowBookOp_thisMember in System.member.s
05   }
06   {
07     op.borrowBookOp_thisBook.state.s in NotBorrowedBookState
08     op.borrowBookOp_thisMember.state.s in CanBorrowMemberState +
09       HasNotBorrowedMemberState
10   }
11   {
12     doesNotChangeMemberExcept[s, s', none]
13     doesNotChangeBookExcept[s, s', none]
14     doesNotChangeRelborrowsExcept[s, s',
15       op.borrowBookOp_thisMember->op.borrowBookOp_thisBook]
16     doesNotChangeRelborrowedByExcept[s, s',
17       op.borrowBookOp_thisBook->op.borrowBookOp_thisMember]
18     doesNotChangeStateExcept[s, s',
19       op.borrowBookOp_thisBook + op.borrowBookOp_thisMember]
20   }
21   {
22     op.borrowBookOp_thisBook.state.s' = BorrowedBookState
23     op.borrowBookOp_thisMember->op.borrowBookOp_thisBook in rel_borrows.s'
24   }
25 }
```

Fig. 19. The predicate for the “Return Book” use case, describing the input parameters (lines 2-5), the pre-conditions (lines 6-10), the frame conditions (lines 11-20), and the post-conditions (lines 21-24).

The predicate of a use case may also contains 1) a part for defining variables (using the **let** structure) and 2) a part for creating new objects, when new objects are added to the domain model. An example of these parts are depicted in Fig. 20, which shows the predicate for “Add Member”. Any time a new objects is added to the domain model, the framework reduces the solution space by forcing a total order on the signatures for all classes and using the smallest one (line 2).

```
01 pred addMemberOp(s, s': State, op: AddMemberOp) {
02   let lname0 = min[Member - System.member.s] {
03     {
04       doesNotChangeMemberExcept[s, s', lname0]
05       doesNotChangeBookExcept[s, s', none]
06       doesNotChangeRelborrowsExcept[s, s', none->none]
07       doesNotChangeRelborrowedByExcept[s, s', none->none]
08       doesNotChangeStateExcept[s, s', lname0]
09     }
10     {
11       lname0 in System.member.s'
12       lname0.state.s' = HasNotBorrowedMemberState
13     }
14   }
15 }
```

Fig. 20. The predicate for the “Add Member” use case.

In addition to the corresponding predicate, the transformation produces a predicate with the prefix **fail_**. This predicate is similar to the main predicate except that it does not define the post-conditions and the new objects, as it is only used to generate a state where the pre-conditions are falsified. This predicate can be used to test the system is capable of detecting when use cases are not allowed to run.

Additionally, it is possible to define different execution paths for a use case. In this case, an abstract signature is defined for the use case and for each execution path a concrete signature is defined.

5.5 Initial State

For transforming the initial state model into an Alloy specification, the transformation specified the size of each relations. Also, for simplifying the solution space, the smallest possible object in the initial state is chosen using the total ordering. Fig. 21 shows the translation of the initial model in Fig. 12. Lines 2-4 define a book that has been borrowed, lines 5-7 defines a member that can borrow books, and lines 8-10 define the association between these two objects.

```
01 pred init(initialState: State) {
02   one System.book.initialState
03   min[Book].state.initialState = BorrowedBookState
04
05   one System.member.initialState
06   min[Member].state.initialState = CanBorrowMemberState
07
08   one rel_borrowedBy.initialState
09   one rel_borrows.initialState
10   min[Member]->min[Book] in rel_borrows.initialState
11 }
```

Fig. 21. The initial state in Alloy.

5.6 Labels Rules

The Alloy specification for the object labels rules are generated from our Java-based DSL introduced in Section 4.5.2. Fig. 22 shows the translation of the model in Fig. 5.

```
pred rules(s: State) {
  all lname: System.member.s | {
  no lname.rel_borrows.s => lname.state.s = HasNotBorrowedMemberState else
  #(lname.rel_borrows.s) >= 2 => lname.state.s = CanNotBorrowMemberState else
  lname.state.s = CanBorrowMemberState
  }
}
```

Fig. 22. The object labels rules in Alloy.

5.7 Input Partitioning

The partitions of the input domain are also specified in Alloy. The specification for the input partitioning in the example library system, is depicted in Fig. 23. Each data type is modelled with an abstract signature (line 1) that is extended with a concrete signature for each partition (e.g., line 2). If a partition has input parameters, they are also modelled (e.g., line 5).

```
01 abstract sig BooknameDataType{}
02 sig unique_BooknameDataType extends BooknameDataType {}
03 fact { lone unique_BooknameDataType }
04 sig nonUnique_BooknameDataType extends BooknameDataType {
05   dp_book : one Book,
06 }
07 fact { no disj op1, op2 : nonUnique_BooknameDataType | {
08     op1.dp_book = op2.dp_book
09   }
10 }
11 sig empty_BooknameDataType extends BooknameDataType {}
12 fact { lone empty_BooknameDataType }
```

Fig. 23. The input partitioning in Alloy.

5.8 Main Module and Test Goals

The main module is static and is independent of the SUT and its models. The module is depicted in Fig. 24. In the beginning, the module is defined and other modules are opened (lines 1-7). Then, the signatures for the system states and the abstract signature `Op`, used for tracing use cases, are defined (lines 9-12). Next, lines 15-17 require for all states, excluding the final state, a use case has to be executed. The predicate `apply` (given in Fig. 25) decides which predicate (corresponding to the use cases), has to be applied to the model. Line 19 enforces the label rules and line 20 constricts the state space and forbids the creation of useless `Op` atoms. Lines 25-27 describe the initial state of the system. The search scope is defined in lines 29-30. Optional, user-defined constraints in the Alloy model can be added in the `testSystem` predicate. In this example, the search scope is 5 atoms for all signature, excluding atoms of the `State` signature (line 30). The scope for the `State` signature is 4, which results in execution traces of length three (one state is needed for the initial state).

If the test goal is to generate exceptional states, line 21 is commented out and line 22 is uncommented. This applied the `fail_apply` predicate to the final state. The predicate is similar to predicate `apply` except that it includes the negative predicates for the use cases. In this case, valid paths of length three are tested and then an exceptional operation is attempted at the final state. Exceptions cannot alter the domain model and therefore including them in the middle of a path is useless.

```
01 module main
02 open util/ordering[State]
03 open init_def_auto[State,Op]
04 open system_def_auto[State]
05 open operations_def_auto[State, Op]
06 open input_def_auto[State]
07 open rules_def_auto[State,Op]
08
09 sig State {
10   op: lone Op
11 }
12 abstract sig Op {}
13
14 fact traces {
15   all s : State - last | let s' = s.next |
16     some operation : Op |
17       apply[s,s', operation] and s.op = operation
18
19   all s: State | rules[s]
20   Op in State.op
21   no last.op
22 // fail_apply[last, last.op]
23 }
24
25 fact initial_state{
26   init[first]
27 }
28
29 pred testSystem(s: System) {}
30 run testSystem for 5 but 4 State
```

Fig. 24. The Alloy main module.

```
pred apply(s, s': State, op: Op) {
  op in DelBookOp implies delBookOp[s,s',op] else
  op in ReturnBookOp implies returnBookOp[s,s',op] else
  op in BorrowBookOp implies borrowBookOp[s,s',op] else
  op in AddBookOp implies addBookOp[s,s',op] else
  op in DelMemberOp implies delMemberOp[s,s',op] else
  op in AddMemberOp implies addMemberOp[s,s',op]
}
```

Fig. 25. The `apply` predicate for the example library system.

5.9 Generated Test Cases

For the example library system, Alloy generates 73 test cases with the search scope given in Fig. 24, line 30 (i.e., `run testSystem for 5 but 4 State`). One of the test cases is depicted in Appendix A, Fig. 29.

If the test goal is to generate exceptions (i.e. using the `fail_apply` predicate), for the search scope of size one, five test cases are generated which are shown in Appendix A. Fig. 30 represents a test case that attempts to remove a member that has borowed a book. Fig. 31 shows a test case that attempts to deleted a book that has been borrowed. Fig. 32 illustrates an invalid borrow operation. Fig. 33 and Fig. 34 try to add a book with empty name and a book with a duplicate name (`Book0`), respectively.

move to table

6 Evaluation

To assess the effectiveness and applicability of our approach, we conducted a case study on a non-trivial business application. We present measures of test coverage including line coverage and mutation score to provide a comparison with manually developed tests.

@Masou which table??

6.1 Case Study

The subject program is an internal mailing system, developed as a desktop application. The system was developed throughout an iterative-incremental process using Java language by a team of undergraduate students as a final project. The authors of this study implemented the framework described in section 4 on the code developed by one of the student groups. The code consists of approximately 1000 LOC in business logic and 6000 LOC in UI logic.

Students documented the requirements and use cases were in an analysis phase, and most of the use cases are include sending and manipulating the internal mails. The system was developed incrementally, and a working system was delivered at the end of each iteration to the course graders. The detailed description of the case study is available in [28].

In the case study, we have focused on 15 use cases listed in Fig. 26. The system has other use cases such as "View Roles" and "Aggregated Report", which do not make changes to the domain model and thus were excluded from our study.

Structural Model. The structural model is shown in Fig. 27. Authors of this paper created it based on the analysis structural model the students created in the system analysis phase.

Data Types. We identified the following data types and partitioning for them:

- Boolean; used for accepting or rejecting a process and defining the state of a mail ("action required" and "information only"). It has two partitions of "true" and "false".
- String; used for string inputs including as mail's body and title. It has two partitions of "empty" and "non-empty".
- Role Name; for this data type three partitions, including "empty", "non-empty", and "duplicate", were defined.
- Username: it has three partitions: "empty", "non-empty", and "duplicate".
- Password Pairs: it is used for creating password in signing up. It has three partitions: "empty", "identical pairs", and "non-identical pairs".

Object Labels. Authors identified seven labels for the "Mail" object, namely "ForApproval", "ApprovedBefore", "Received", "ReceivedNeedsReply", "ReceivedBefore", "Forwarded", and "Sent". The "Structure" domain object has also two labels including "NotRoot" and "HasRoot". Other domain objects did not have state-based behaviour and thus we defined no labels for them.

Subsidiary Models. No label rules were used, due to limited dependencies. The only subsidiary model was the initial state(Fig. 28).

Test Harness. The test harness was developed for our case study and it consists of about 300 LOC. To connect the SUT to the test driver, we used the uispec4j[10] library. Additionally, most of the existing code of the system were reused for developing the "Inspector" component.

10. uispec4 http://uispec4j.org

TABLE 1
Test goals for the case study.

Test Goal	# Test Cases	Time (second)
Execution paths of length 4	129	28
Execution paths of length 6 including "Send Internal Mail"	72	26
Execution paths of length 7 including "Send Internal Mail" to non-lower staff	180	79
Execution paths of length 7 including "Reply to Internal Mail"	108	51
Execution paths of length 8 including "Confirm Internal Mail"	360	214
Execution paths of length 8 including "Forward Internal Mail"	288	174
Total	1137	572 (10 mins)

Behavioural Model. The process of modelling the use cases uncovered analysis problems in three use cases. These use cases ("Remove User", "Delete Role", and "Remove Role") were excluded from the rest of the case study. [11] The behavioural model included about 170 LOC in the internal DSL.

6.1.1 Test Execution and Result

After defining the required models, the system was tested using the test goals and constrains shown in Table 1. The test goals were defined incrementally starting with executing all short use cases, then longer paths covering complex use cases. We also tested the system in order to generate exceptions. All the computations were performed in one thread and each use case execution took about half a second, this could be improved by a concurrent implementation.

Although the system was approved and graded in several iterations, the case-study uncovered significant problems on different levels, including implementation, analysis, and requirement specification. For example, in the implementation level, we found out that the system generally accepts "empty" strings, which were explicitly forbidden by requirements (e.g., user name, password, and role name). In some cases, requirements were ambiguous (e.g., content and subject). We also found the system allows a role sends an internal mail to herself, while the system does not create any internal mail[12] .

Sec 6.1. discusses an analysis-level issue uncovered by the study. In requirements specification, it was not clear if an internal mail to the direct manager requires approval or not[13].

6.2 Line Coverage

We used EMMA[14], to measure the line coverage of tests generated by our framework (Table 2). "Unrelated" refers

11. Letters of deleted users and roles did not were deleted, even though requirements said the system had to retain them. Inclusion of these use cases would lead to early discovery of error in the code and would result in ending the testing process.

12. The output of Alloy finding this error is available at http://ce.sharif.edu/~jalalinasab/bug1.xml

13. The output of Alloy finding this error is available at http://ce.sharif.edu/~jalalinasab/bug4.xml

14. EMMA http://emma.sourceforge.net/

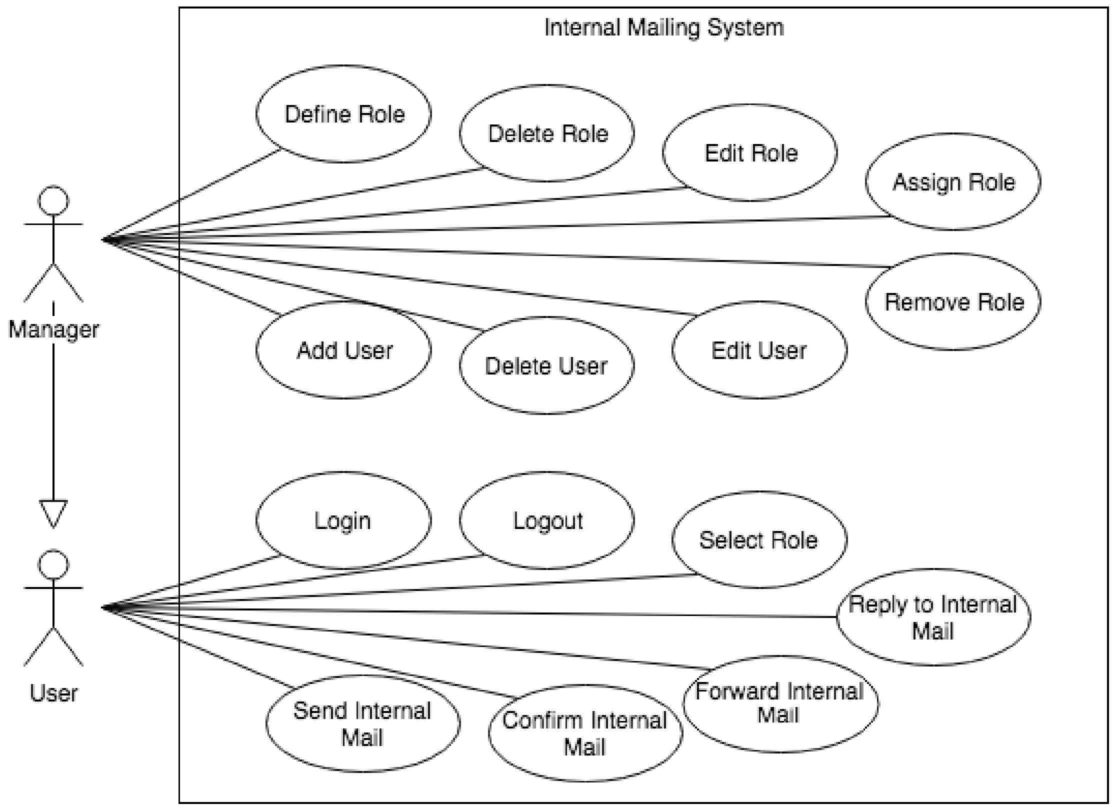


Fig. 26. Use case diagram for the case study.

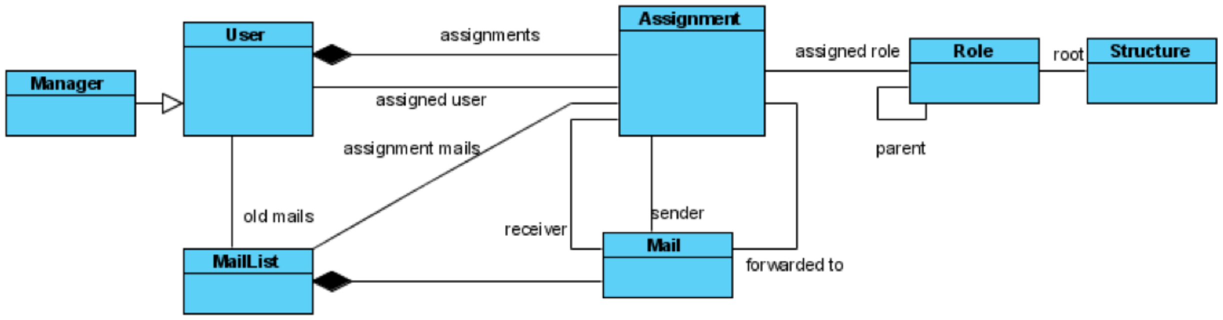


Fig. 27. Structural model of the case study.

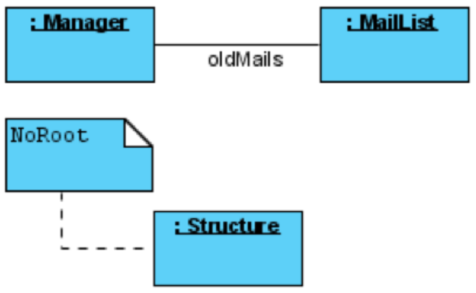


Fig. 28. Initial State for the case study.

to the classes for the use cases excluded from the study. We did not exclude “unrelated” use cases from business logic,

TABLE 2
Line coverage for the case study.

Module	Covered	Not Covered	Unrelated	Total
UI	3290	246	2310	5846
Business Logic	526	159	-	685

as it was hard to clearly determine *unrelated* code.

Excluding the “Unrelated” code, the coverage for UI and business logic were 90% and 73% achieved without writing any test script.

6.3 Mutation Testing

We also measured the mutation score of the generated tests using the PIT[15] mutation generator tool was used. The de-

15. PIT http://pitest.org/

TABLE 3
Mutation testing result with respect to the operators.

Operator	Survived	Killed	Not Covered	Total
1	2	21	6	29
2	21	90	19	130
3	3	4	0	7
4	5	17	6	28
5	14	75	28	117
6	79	121	70	270
Total	125	327	129	581

TABLE 4
Mutation testing result with respect to the modules.

Module	Survived	Killed	Not Covered	Total
UI	87	186	57	330
Business Logic	38	141	72	251

fault mutation operators supported in this tool were used in the case study, including 1) mutations in boundary values, 2) mutations by negation of conditions, 3) mutations in computational operators, 4) mutations in increment/decrement operators, 5) mutations in return values, and 6) mutations in methods with no return value.

The summary of the results are provided in Table 3 and Table 4, showing the results grouped by mutation operators and modules respectively. Overally, 56% of the mutants were killed. This calculation includes mutations in lines of code not covered by the tests. Excluding these mutants raises the mutation score to 68% for UI code and 79% for business logic code. The low mutation score for the UI module is basically because the test cases can not identify (and hence skip) the method calls that are for updating the representation of the UI. However, the achieved mutation scores indicate that the generated test cases are able to distinguish between the main program and the acceptable number of mutants particularly where the code is covered by test cases.

7 Discussion and Limitations

7.1 Coverage

Our testing framework can easily provide *use case coverage*, using user-defined test goals. The tester can enforce the inclusion of each use case, at least, once regardless of the execution paths inside the use case, or enforce that all execution paths inside each use case to be covered.

The first type can be easily achieved by the proposed framework by (i) adding `some [signature corresponding to the use case]`, for each use case in the predicate defining the test goals, and (ii) increasing the search scope so that Alloy can find a solution.

For the second type of use case coverage, the tester needs to define valid internal execution paths for each use case in the behavioural model. Alloy Analyser can be forced to cover specific paths by putting `some [signature corresponding to the use case]_[name of the execution path]` in the predicate defining the test goals and incrementally increase the search scope.

Nevertheless, an execution path may imply different concrete (or actual) paths. For example, it may have an input with a partitioning which requires to choose an input for each of the partitions. Also, there could be various paths resulting in a failed use case. In all of these cases, the tester has to specifically, model the internal execution paths to achieve the coverage based on the internal paths, as such the coverage is up to the paths specified by the tester which may however, not be included in the search space. In fact, an enough large search scope would result in complete coverage which is typically impossible in practice.

Another related coverage criteria is based on the partitioning of the input domain. The testing framework creates all the permutations of the states, and by choosing the two built-in test goals (complete execution of use cases and exceptions) the generated tests will cover all the state permutations. In fact, as the number of partitions are typically small, by choosing an enough large search scope all the permutations would be considered in the testing, i.e., complete coverage.

Note that the labels of the domain objects also imply partitioning for the input domain and considering them as input partitioning would result in the challenging problem of determining the search scope.

7.2 Relation to Similar Studies

Although the work presented here is inspired by TestEra [29], in which tests are generated using Alloy, our work substantially differs from TestEra: 1) TestEra focuses on unit testing (tests are defined at method level) whereas in this work, use cases and their execution sequence are used for test generation (i.e., acceptance testing), and 2) in TestEra, the Alloy specifications are created manually and therefore required testers to learn Alloy's modelling language.

Another similar work is [15], in which tests are automatically generated from the formal specification of requirements and the class diagram. The data types are limited to Integers and the verification is performed by a set of fault models to mutate an object diagram. Using a formal language, limiting to Integer data types, and verification based on fault models are the main limitations of that work compared to this.

In [30], a first-order logic language is used to define system constraints, and the test goals are specified by object diagrams, both of which are not appropriate for using in lightweight processes–our main target. Their approach is mainly appropriate for systems that has complex state-based behaviour, in contrast to our framework. Similarly, [31], which mainly uses state diagrams for describing requirements, requires complex behavioural modelling which makes it unsuitable for lightweight development.

Testing based on "virtual contracts" [32] is similar to our work as such they are analogous to pre-/post-conditions. However, that work focuses on unit testing. Also, testers would be more familiar with our internal Java-based DSL in comparison to virtual contracts.

Consequently, our framework has advantages over the previous and similar studies by 1) using lightweight and situational models, 2) automatically generating test execution paths, 3) providing flexible methods for generating test data and verification, and 4) providing traceability to requirements.

7.3 Limitations

Nevertheless, the presented testing framework has made few assumptions about the system under test and the methodology used for its development. Also, similar to any approach based on static analysis, our proposal is subject to constraints caused by using this technique. The assumptions and limitations are as follow.

- The requirements are available and the use cases have detailed specification and contains the details required by the framework.
- The structural model of the primary and persistent classes of the domain is available.
- The main target of the SUT is to manage the domain objects and hence, complex data processing systems are not suitable for using this framework.
- The state of the system is determined by the relationship between the domain objects and data dependencies are represented by object labels. This makes the framework unsuitable for the verification of protocols and reactive systems.
- The system under test is deterministic.
- Execution of use cases do not have any side effect. Otherwise, the appropriate mechanisms have to be provided by the test harness.
- The mapping between the input parameters and the conditional statements, to the domain objects can be done easily.

It is worth noting that this work applies well to testing data-oriented systems developed using lightweight development processes, as these assumptions and limitations do not hamper the application and benefits of the framework in practice. Our case study, in particular, demonstrates the applicability and effectiveness of the proposed framework in testing a typical, non-trivial software application. It shows that in addition to implementation errors, the framework can identify problems in analysis and ambiguities in requirements specifications, even throughout the development of a system that is iteratively delivered and approved by stakeholders.

8 Related Work

Here, we provide a discussion of the related efforts in the context of our research.

8.1 Pre-/Post-conditions-based Testing

Scheetz et al. [30] provides an approach for test generation based on class diagrams. The effects of each method on the state of the system (i.e., post-conditions) are specified by a first order logic language. Test goals are the descriptions of the expected objects' states in the system that are defined manually. Test goals and the system initial state are represented by object diagrams. These models, in addition to the post-conditions of the methods are given to a *planner* that generates a test suite that satisfies the test objectives. This approach is mainly appropriate for systems that has complex state-based behaviour, in contrast to our framework.

Cavarra et al. [31] present a test generation method based on class, object, and state diagrams. The class diagram identifies the entities in the system and the state diagrams–one for each class–explain how these entities may evolve. The object diagrams are used to describe the initial configuration of the system model, to specify a starting configuration in a test directive, and to flag configurations for inclusion or exclusion in a test model. A state diagram shows how an object will react to the arrival of an event. This method is similar to the method proposed in [30] except in that the behaviour is modelled by a first order logic language.

Bouquet et al. [33] defines a subset of UML 2.1 [34] that allows formal behaviour models of the SUT. The subset uses class, instance and state diagrams, in addition to OCL [35] expressions. The models are used as input for a model-based test generator, called LEIRIOS Test Designer, that automates–using theorem prover–the generation of test sequences, covering each behaviour in the model. Object diagrams are used to specify the system initial state and test data. The state diagrams are an optional part and used to model the dynamic behaviour of the SUT as a finite state transition system. The OCL language has been extended to support execution semantics. For each test target, an automated theorem prover is used to search for a path from the initial state to that target, and to find data values that satisfy all the constraints along that path.

8.2 Static Analysis-based Testing

The main idea in [32] to raise the abstraction level of the "design by contract" [36] concept to the level of design models. Each method's functionality is specified by a "visual contract" –consists of two UML composite structure diagram. One diagram designates the pre-conditions, while the other specifies the post-conditions. Visual contracts are then automatically translated into JML[16] assertions in the original code which are monitored, and an exception will be raised upon the violation of a contract. The static analysis techniques are used to automatically find methods' pre-conditions. The main shortcoming of this work is that it does not consider data dependencies. The work differs from the proposed framework in this paper as firstly, it focus on unit testing and also it uses models for describing pre-/post-conditions.

The technique presented in [37] allows for QA on static aspects of class diagrams using static analysis. It proposes an extension to object diagrams, namely the "modal object diagram" that allows defining positive/negative object configurations that the class diagram should allow/disallow. Doing so, each modal object diagram can be considered as a test case that the class diagram should satisfy. Ultimately, the verification is performed by a fully automated model checking-based technique.

The issue of testing UML models as independent from implementation is discussed in [38]. It proposes several testing adequacy criteria for different UML models. The criteria for class diagrams involve creating instances with all possible multiplicities, creating all possible subclasses,

16. JML Java Modeling Language http://jmlspecs.org

and creating objects with different field combinations. A set of coverage criteria are introduced for UML collaboration diagrams including predicate and term coverage of UML guards assigned to message edges, and the execution all the messages and paths. These coverage can automatically be applied on the created models and subsequently result in set of test cases (the latter aspect, however, is not discussed in [38]).

Di Nardo et al. [Nardo,2017;Nardo, 2015a;Nardo, 2015b] focus on testing data processing software that requires generating complex data files or databases, while ensuring compliance with multiple constraints. The structure and the constraints of input data are modelled by UML class diagrams and OCL constraints. The approach is built upon UML2Alloy [Anastasakis , 2007] to generate an Alloy model that corresponds to the class diagram and the OCL constraints of the data model. The Alloy Analyser is used to generate valid instances of the data model to cover predefined fault types in a fault model.

8.3 Specification-based Testing

Specification-based testing has been intensively considered in the testing literature. its importance was discussed in a very early paper by Goodenough and Gerhart [39].

Different specifications have been considered and used for automatic test generation, such as Z specifications [24], [40], [41], [42], UML statecharts [43], [44], ADL specifications [45], [46], Alloy specifications [18], [29], [47], JML specifications [48].

Horcher [40] presents a technique for software testing based on Z specifications [24]. The technique provides automated test execution and result evaluation. However, concrete input test data need to be selected manually from an automatically generated set of test classes. The UMLTest tool [Offutt, 1999] automatically generates tests from UML statecharts and enabled transitions, but requires all variables to be boolean. It also makes several limiting assumptions about the UML input file. Chang et al [46] introduce a technique for deriving test conditions–a set of boolean conditions on values of parameters–from ADL specifications [45]. These test conditions are used to guide test selection and to measure comprehensiveness of existing test suites.

Boyapati et al. [48], introduce Korat, a framework for automated testing of Java programs. Given a formal specification for a method in JML specifications, Korat uses the method pre-conditions and automatically generates all non-isomorphic test cases (within a given input size).The method post-conditions are used as a test oracle to check the correctness of each output.

Khurshid et al. [29] introduce a framework, called TestEra, for automated specification-based testing of Java programs. TestEra uses the method's pre-condition specification to generate test inputs and the post-condition to check correctness of outputs. TestEra supports specifications written in Alloy and uses the SAT-based back-end of the Alloy tool-set for systematic generation of test cases as JUnit test methods.

[15] presents an approach for automatic test generation using the information provided in the domain and behavioural models of a system. The domain model consists of UML class diagram with invariants, while the behavioural model consists of UML use cases. Each use case flow has an associated guard condition and a set of updates (to the domain object diagram and the output parameters). The approach uses a formal language for modelling and is limited to integer data types. The verification is performed by a set of fault models to mutate an object diagram and a novel algorithm which distinguishes between the original and the mutated object diagrams (kind of conformance testing). Whereas, in our framework, the verification is done by the Inspector component and based on labels which is more flexible than the approach presented in [15].

In a later work, Rosner et at. [49] introduces HyTeK, a technique for bounded exhaustive test input generation that automatically generates test suites from input specifications given in the form of hybrid invariants as such they may be provided imperatively, declaratively, or as a combination of declarative and imperative predicates. HyTeK benefits from optimization approaches of each side: (i) the information obtained while solving declarative portions of the invariant assists in pruning the search for partially valid structures from the imperative portion of the specification, and (ii) the *tight bounds* are computed from the declarative invariant and used during test generation both from the declarative and imperative parts of the specification, to reduce the search space. HyTeK combines a mechanism for processing imperative input specifications introduced in [48] through the Korat tool, with SAT solving for processing the declarative portions of the input specification, in the style put forward through the tool TestEra [29].

9 Conclusion

This paper presents a novel approach for model-based acceptance testing. It introduces a minimal set of structural and behavioural models that are normally produced during the development process, and also are intuitive and tangible for programmers and modellers. The domain model, represented as a class diagram or EMF model, is used as the structural model. Use cases, described by the provided DSLs, are used as the behavioural model. A use case description basically defines the pre-/post-conditions, and possibly the data for the use case. Our approach employs static analysis for automatic test generation and execution. All the models are automatically translated into Alloy specifications which are solved by the Alloy Analyser resulted in valid correct execution traces for the SUT.

The proposal was evaluated throughout a case study on a medium-sized business application and analysing the result based on different coverage criteria, namely line coverage, use case coverage, and data coverage. We also applied mutation testing to assess the quality and effectiveness of the generated test cases. The study demonstrates that the framework provides acceptable line coverage (90%) and mutation score of 72%. In addition to implementation errors, the framework would help in identifying errors in analysis and requirements specification.

As for future work, we plan to improve the introduced DSLs to be more concise and provide more modelling features. Moreover, we plan to enhance model transformations to have optimised Alloy models in order to reduce test execution times. Finally, we would like to work on techniques to efficiently deal with evolution in test models.

APPENDIX A
CASE STUDY - EXAMPLE TEST CASES

ACKNOWLEDGMENTS

This work was supported in part by Iran's National Elites Foundation. Raman Ramsin is the corresponding author.

REFERENCES

[1] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[2] D. Faragó, "Model-based Testing in Agile Software Development," *30. Treffen der GI-Fachgruppe Test, Analyse & Verifikation von Software (TAV), Testing Meets Agility*, 2010.

[3] J. Tretmans, "Formal Methods and Testing," R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Model Based Testing with Labelled Transition Systems, pp. 1–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=1806209.1806210

[4] L. Tan, O. Sokolsky, and I. Lee, "Specification-based testing with linear temporal logic," in *Proceedings of the IEEE International Conference on Information Reuse and Integration*, ser. IRI '04, Nov 2004, pp. 493–498.

[5] S. W. Ambler, "Test Driven Development (TDD) Survey Results," 2008. [Online]. Available: http://www.ambysoft.com/surveys/tdd2008.html

[6] ——, "Agile Testing and Quality Strategies: Discipline Over Rhetoric." [Online]. Available: http://www.ambysoft.com/essays/agileTesting.html

[7] M. Katara and A. Kervinen, "Making Model-based Testing More Agile: A Use Case Driven Approach," in *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing*, ser. HVC '06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 219–234. [Online]. Available: http://dl.acm.org/citation.cfm?id=1763218.1763238

[8] D. Faragó, "Improved Underspecification for Model-based Testing in Agile Development," in *Proceedign of the 2nd International Workshop on Formal Methods and Agile Methods*, ser. FM+AM '10, Pisa, Italy, 2010, pp. 63–78. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings179/article6224.html

[9] R. Löffler, B. Güldali, and S. Geisen, "Towards Model-based Acceptance Testing for Scrum," *Softwaretechnik-Trends*, vol. 30, no. 3, 2010.

[10] T. H. Ussami, E. Martins, and L. Montecchi, "D-MBTDD: An Approach for Reusing Test Artefacts in Evolving System," in *Proceedings of 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop*, ser. DSN-W '16, June 2016, pp. 39–46.

[11] D. Jalalinasab and R. Ramsin, "Towards Model-Based Testing Patterns for Enhancing Agile Methodologies," in *Proceedings of the 11th Conference on New Trends in Software Methodologies, Tools and Techniques*, ser. SoMeT '12, Genoa, Italy, 2012, pp. 57–72. [Online]. Available: https://doi.org/10.3233/978-1-61499-125-0-57

[12] ken Pugh, *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Addison-Wesley, 2011.

[13] C. Nebut, F. Fleurey, Y. L. Traon, and J. M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, March 2006.

[14] M. Sarma and R. Mall, "System Testing using UML Models," in *Proceedings of the 16th Asian Test Symposium*, ser. ATS '07, Oct 2007, pp. 155–158.

[15] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is More: A Minimalistic Approach to UML Model-Based Conformance Test Generation," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*, ser. ICST '08, April 2008, pp. 82–91.

[16] J. Erickson and K. Siau, "Theoretical and Practical Complexity of Modeling Methods," *Communications of the ACM*, vol. 50, no. 8, pp. 46–51, Aug. 2007. [Online]. Available: http://doi.acm.org/10.1145/1278201.1278205

[17] ——, "A Decade and More of UML: An Overview of UML Semantic and Structural Issues and UML Field Use," *Journal of Database Management*, vol. 19, pp. i–vii, Jul. 2008.

[18] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002. [Online]. Available: http://doi.acm.org/10.1145/505145.505149

[19] Eclipse Modeling Framework (EMF), "https://www.eclipse.org/modeling/emf/."

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] K. Beck and E. Gamma, "More java gems," D. Deugo, Ed. New York, NY, USA: Cambridge University Press, 2000, ch. Test-infected: Programmers Love Writing Tests, pp. 357–376. [Online]. Available: http://dl.acm.org/citation.cfm?id=335845.335908

[22] R. Ferguson and B. Korel, "The Chaining Approach for Software Test Data Generation," *ACM Transactions on Software Engineering Methodologies*, vol. 5, no. 1, pp. 63–86, Jan. 1996. [Online]. Available: http://doi.acm.org/10.1145/226155.226158

[23] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[24] J. M. Spivey, *The Z Notation: A Reference Manual, Second Ed.* Prentice-Hall, Inc., 1992.

[25] D. Jackson, *Software Abstractions: Logic, Language, and Analysis, Revised Ed.* The MIT Press, 2012.

[26] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: The Alloy Constraint Analyzer," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 730–733. [Online]. Available: http://doi.acm.org/10.1145/337180.337616

[27] I. Shlyakhter, "Generating Effective Symmetry-breaking Predicates for Search Problems," *Discrete Appl. Math.*, vol. 155, no. 12, pp. 1539–1548, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1016/j.dam.2005.10.018

[28] D. Jalalinasab, "A Lightweight Approach to Model Based Testing," Master's thesis, Sharif University of Technoligy, Iran, 2012.

[29] S. Khurshid and D. Marinov, "TestEra: Specification-Based Testing of Java Programs Using SAT," *Automated Software Engineering*, vol. 11, no. 4, pp. 403–434, Oct 2004.

[30] M. Scheetz, A. von Mayrhauser, and R. France, "Generating test cases from an OO model with an AI planning system," in *Proceedings of the 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*, 1999, pp. 250–259.

[31] A. Cavarra, C. Crichton, J. Davies, A. Hartman, and L. Mounier, "Using UML for automatic test generation," in *Proceedings of the International symposium of software testing and analysis*, ser. ISSTA '02. Springer-Verlag, 2002.

[32] G. Engels, B. Güldali, and M. Lohmann, "Towards model-driven unit testing," in *Proceedings of the 2006 International Conference on Models in Software Engineering*, ser. MoDELS'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 182–192. [Online]. Available: http://dl.acm.org/citation.cfm?id=1762828.1762859

[33] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A Subset of Precise UML for Model-based Testing," in *Proceedings of the 3rd International Workshop on Advances in Model-based Testing*, ser. A-MOST '07. New York, NY, USA: ACM, 2007, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1291535.1291545

[34] OMG, "Unified Modeling Language (UML) Version 2.0," 2005. [Online]. Available: http://www.omg.org/spec/UML/2.0/

[35] T. O. M. Group, "The Object Constraint Language (OCL)," 2015. [Online]. Available: \url{https://www.omg.org/spec/OCL/}

[36] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. [Online]. Available: http://dx.doi.org/10.1109/2.161279

[37] S. Maoz, J. O. Ringert, and B. Rumpe, "Modal Object Diagrams," in *ECOOP 2011 – Object-Oriented Programming*, M. Mezini, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 281–305.

[38] A. Andrews, R. France, S. Ghosh, and G. Craig, "Test adequacy criteria for UML design models," *Software Testing, Verification and Reliability*, vol. 13, no. 2, pp. 95–127. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.270

[39] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," *ACM SIGPLAN Notices - International Conference on Reliable Software*, vol. 10, no. 6, pp. 493–510, Apr. 1975. [Online]. Available: http://doi.acm.org/10.1145/390016.808473

[40] H.-M. Hörcher, "Improving software tests using Z Specifications," in *ZUM '95: The Z Formal Specification Notation*, J. P. Bowen and M. G. Hinchey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 152–166.
[41] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777–793, Nov. 1996. [Online]. Available: https://doi.org/10.1109/32.553698
[42] M. R. Donat, "Automating Formal Specification-Based Testing," in *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, ser. TAPSOFT '97. London, UK, UK: Springer-Verlag, 1997, pp. 833–847. [Online]. Available: http://dl.acm.org/citation.cfm?id=646620.697731
[43] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language Reference Manual*. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999.
[44] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications," in *Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard*, ser. UML'99. Berlin, Heidelberg: Springer-Verlag, 1999, pp. 416–429. [Online]. Available: http://dl.acm.org/citation.cfm?id=1767297.1767341
[45] S. Sankar and R. Hayes, "ADL&Mdash;an Interface Definition Language for Specifying and Testing Software," *ACM SIGPLAN Notices*, vol. 29, no. 8, pp. 13–21, Aug. 1994. [Online]. Available: http://doi.acm.org/10.1145/185087.185096
[46] J. Chang and D. J. Richardson, "Structural Specification-based Testing: Automated Support and Experimental Evaluation," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 285–302, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/318774.318948
[47] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan, "Software assurance by bounded exhaustive testing," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 328–339, April 2005.
[48] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 123–133, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566171.566191
[49] N. Rosner, V. Bengolea, P. Ponzio, S. A. Khalek, N. Aguirre, M. F. Frias, and S. Khurshid, "Bounded Exhaustive Test Input Generation from Hybrid Invariants," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 655–674. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660232

PLACE
PHOTO
HERE

Darioush Jalalinasab Biography text here.

PLACE
PHOTO
HERE

Masoumeh Taromirad Biography text here.

PLACE
PHOTO
HERE

Raman Ramsin Biography text here.

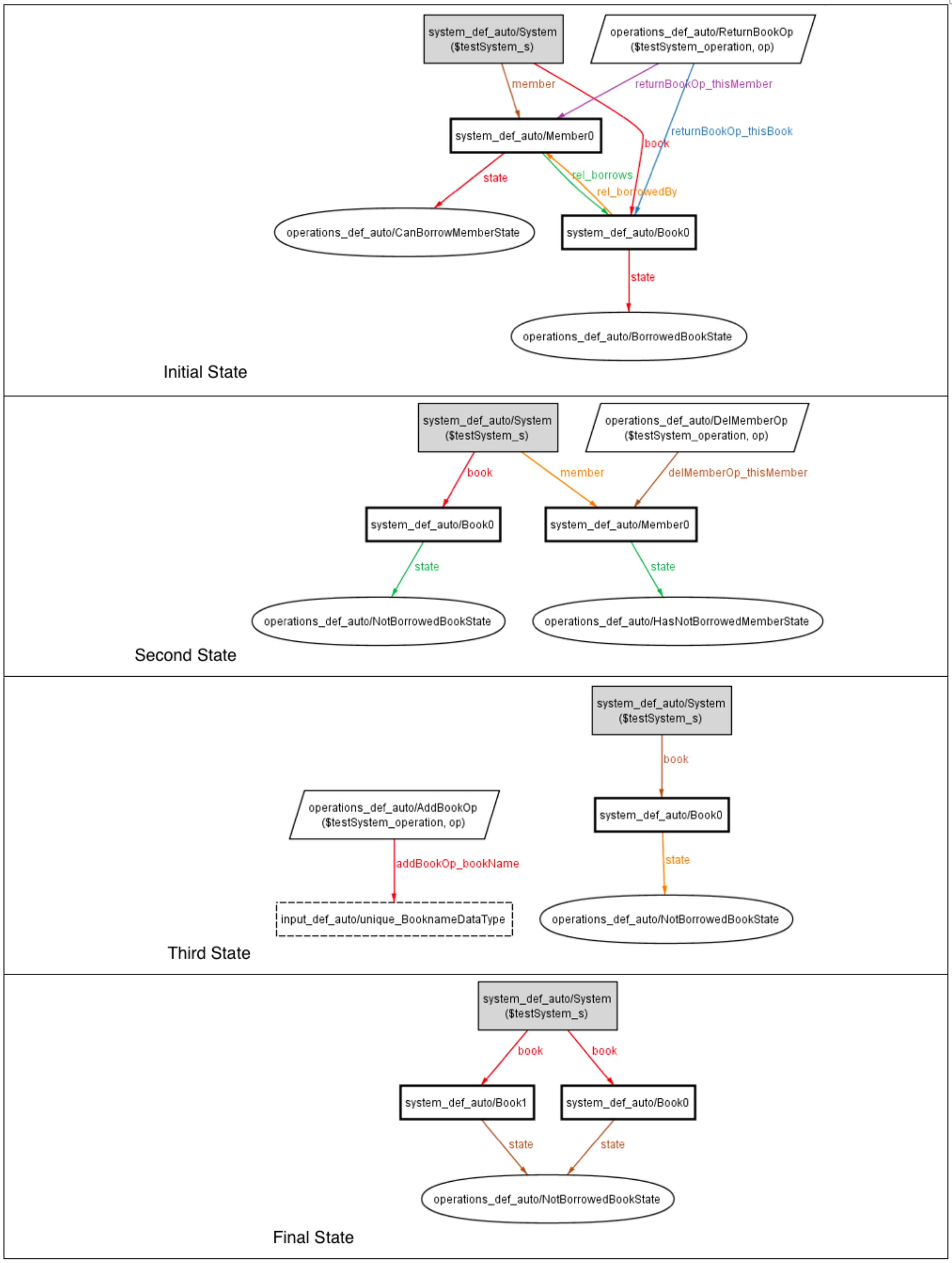

system_def_auto/System ($testSystem_s)
operations_def_auto/ReturnBookOp ($testSystem_operation, op)
member
returnBookOp_thisMember
system_def_auto/Member0
book
returnBookOp_thisBook
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState
Initial State
system_def_auto/System ($testSystem_s)
operations_def_auto/DelMemberOp ($testSystem_operation, op)
book
member
delMemberOp_thisMember
system_def_auto/Book0
system_def_auto/Member0
state
state
operations_def_auto/NotBorrowedBookState
operations_def_auto/HasNotBorrowedMemberState
Second State
system_def_auto/System ($testSystem_s)
book
operations_def_auto/AddBookOp ($testSystem_operation, op)
system_def_auto/Book0
addBookOp_bookName
state
input_def_auto/unique_BooknameDataType
operations_def_auto/NotBorrowedBookState
Third State
system_def_auto/System ($testSystem_s)
book
book
system_def_auto/Book1
system_def_auto/Book0
state
state
operations_def_auto/NotBorrowedBookState
Final State

Fig. 29. An execution path (test case) generated by Alloy - *Diamond*: active use case, *Gray Rectangle*: atom `System`, *Bold Rectangle*: atoms of the domain model, *Eclipse*: labels, and *dashed Line*: input data.

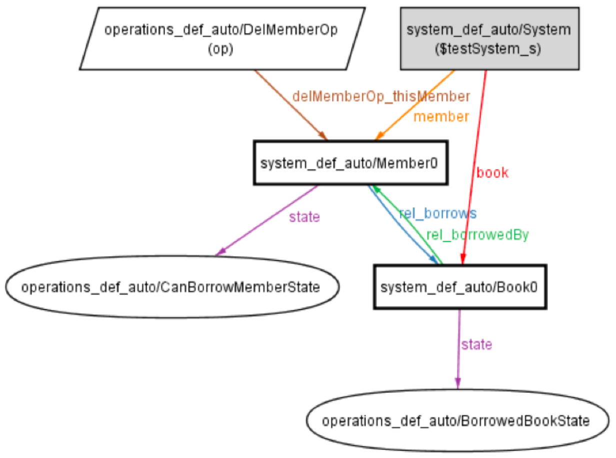

operations_def_auto/DelMemberOp (op)
system_def_auto/System ($testSystem_s)
delMemberOp_thisMember
member
book
system_def_auto/Member0
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState

Fig. 30. Exception state - remove member.

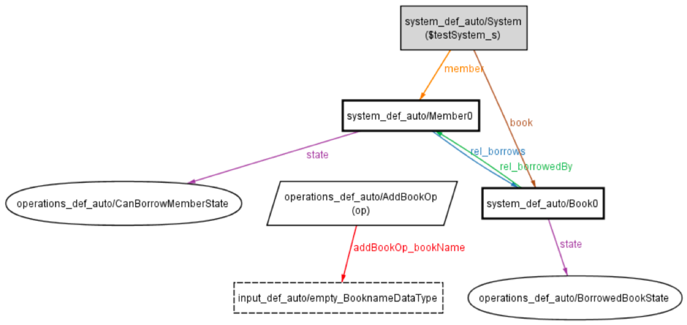

system_def_auto/System ($testSystem_s)
member
book
system_def_auto/Member0
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
operations_def_auto/AddBookOp (op)
system_def_auto/Book0
addBookOp_bookName
state
input_def_auto/empty_BooknameDataType
operations_def_auto/BorrowedBookState

Fig. 33. Exception state - add book: empty name.

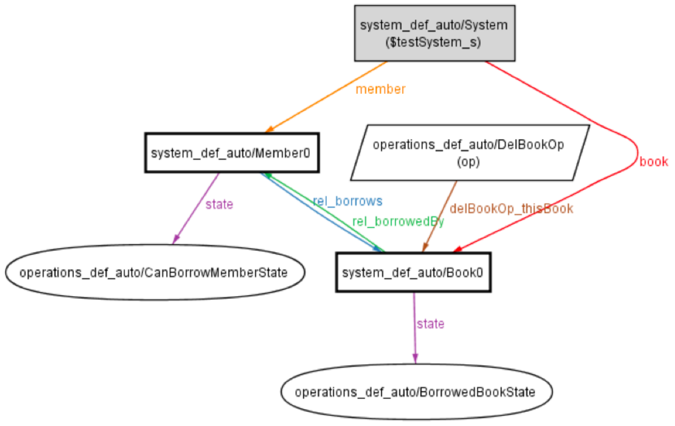

system_def_auto/System ($testSystem_s)
member
system_def_auto/Member0
operations_def_auto/DelBookOp (op)
book
state
rel_borrows
rel_borrowedBy
delBookOp_thisBook
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState

Fig. 31. Exception state - remove book.

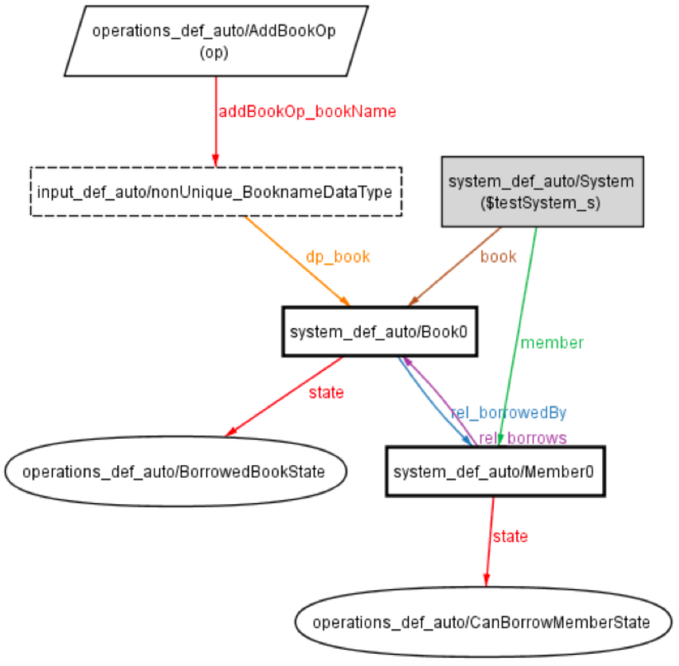

operations_def_auto/AddBookOp (op)
addBookOp_bookName
input_def_auto/nonUnique_BooknameDataType
system_def_auto/System ($testSystem_s)
dp_book
book
system_def_auto/Book0
member
state
rel_borrowedBy
rel_borrows
operations_def_auto/BorrowedBookState
system_def_auto/Member0
state
operations_def_auto/CanBorrowMemberState

Fig. 34. Exception state - add book: redundant name.

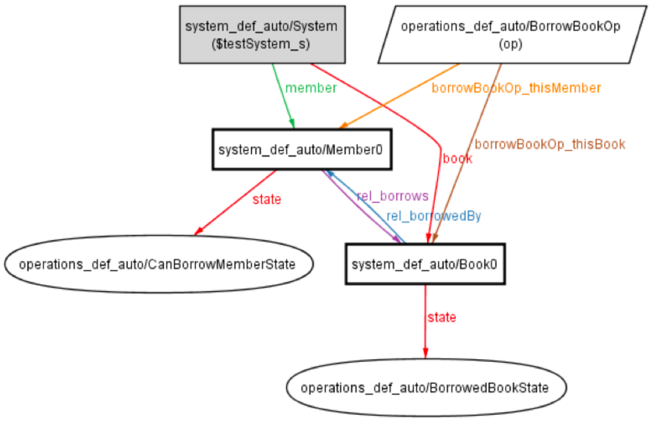

system_def_auto/System ($testSystem_s)
operations_def_auto/BorrowBookOp (op)
member
borrowBookOp_thisMember
system_def_auto/Member0
book
borrowBookOp_thisBook
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState

Fig. 32. Exception state - borrow book.