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

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

Abstract—Testing is an invaluable tool in software development: estimates show up to 50% of resources of software projects are spent on testing. However, measuring the quality of tests is difficult. Determining what tests to write is often left to the discretion and experience of the individual developer. Model-Based Testing (MBT) takes a systematic approach to test generation that avoids those biases, allowing for an objective measure of test coverage.

MBT introduces the issue of updating models and code together, a practice agile development approaches often avoid. This is made worse as MBT often requires models not usually developed in normal development processes.

This paper introduces a a minimal set of structural and behavioral models to automatically generate acceptance tests. In choosing these models the primary concern was keeping models intuitive and accessible for practitioners. The detail description of the models and the relevant transformations are also provided. The feasibility of the proposed approach is demonstrated through a non-trivial case study. Further evaluation is given by analysing the outcome of the case study (e.g., discovered faults and errors and test coverage) with respect to different coverage criteria, namely line coverage, input coverage, and use case coverage. The effectiveness of the approach is also assessed by mutation testing.

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

1 Introduction

In the modern software era, testing plays an invaluable role in software development as a quality assurance measure. Estimates have been made that up to 50% of effort and resources in software projects are allocated to testing. In particular, testing is a cornerstone to quality assurance in lightweight [1]. However, they are plagued with several deficiencies in this regard, such as complex and difficult-to-maintain test scripts [2].

A promising solution to testing issues lies in automated Model-Based Testing (MBT) [3], which provides a structured approach to testing from high-level behavioural models. The benefit of MBT is primarily in automated test case generation and automated analysis of the test results, and thus they have received significant attention in testing complex software systems. However, many of MBT methods are not directly suitable for application to lightweight methodologies [2]. There are several studies on the opportunity of benefiting from MBT in lightweight approaches and vice versa (e.g., [4], [5], [6], [7]). A comprehensive study on the application of MBT in lightweight processes has been provided in our earlier work [8].

Model based testing (MBT) involves creating test cases from software models. When test cases are created from models, they reside at a higher level of abstraction, therefore creating and maintaining them will cost less, and will utilize more automation possibilities. Also, MBT takes a systematic route to test generation; this allows for a more goal-oriented and direct approach in achieving desirable test coverage criteria, especially when compared to ad-hoc testing.

Keeping models updated has always been a dire problem in development approaches that are based on models, and MBT is no exception. Also, models that MBT requires, usually involve details that are not considered in normal (lightweight) development processes, or are completely different models from the models developed during general software analysis and design, which only makes this problem worse.

The goal of this research is to introduce a minimal set of structural and behavioural models that can be used as a basis for MBT. The models that are normally produced during the development process, and simple models, developed solely for testing, were given priority. The primary concern in choosing these models was keeping models intuitive and accessible for practitioners, so that the proposed method would require no particular modelling or model-based testing knowledge.

We have focused on acceptance testing as lightweight methodologies typically consider unit testing and ignore acceptance testing due to associated technical difficulties [9], [10].

In general, automatic acceptance testing techniques are categorised in two types: 1) Script-based techniques, and 2) Techniques based on system behavioural models. In the *script-based techniques*, developers write tests based on informal requirements and case by case, in a scripting language specified by a testing framework. The FIT [1] and Cucumber [2] frameworks, used in Acceptance Test Driven Development (ATDD) [11] are examples of such techniques,

- *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

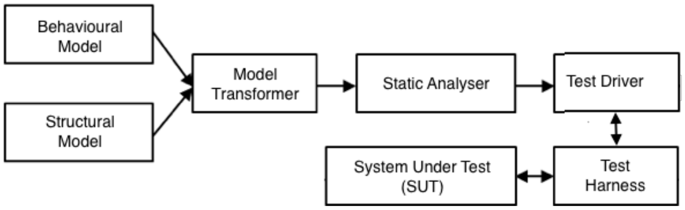


Fig. 1. Overall structure of the proposed framework.

which however suffer from low abstraction level that results in brittle, high-maintenance tests which must be developed and maintained individually. In the *techniques based on system behavioural models*, tests are generated based on system behaviour (e.g., use cases) that is formalised as behavioural models (e.g., [12], [13], [14]). These models implicitly define the test execution paths for the system under test (SUT). Deriving tests based on behavioural models addresses the main problem with script-based testing–by providing higher level of abstraction. It, however, struggles with complex and inappropriate models for lightweight processes.

Accordingly, this paper presents an acceptance testing technique based on system behavioural models, and introduces a set of simple models, so it becomes practical for application in agile processes. Our method uses simple models, in particular class diagrams and use cases, that are considered as the most useful models in software development [15], [16]. The proposed approach is in fact an implementation of the "Directing testing by modelling the permissible order of operations" and "Using static analysis" patterns that were introduced for applying MBT in agile/lightweight processes in our earlier work [8].

Fig. 1 shows the overall structure of the proposed testing framework that has been tailored to data-oriented systems. Tests are generated usign 1) the domain model (or the class diagram), as the structural model, 2) use cases, as the behavioural model, and 3) a static analyser (i.e., Alloy [17]). The models are automatically translated into Alloy specifications. The solutions to the Alloy models are translated into executable test cases which are executed on the SUT via the test driver and the test harness.

The effectiveness and applicability of the proposed framework was evaluated by applying the framework in testing a medium-sized business application. The outcome of the case study, including discovered errors and test coverage, was analysed. Various testing coverage, namely line coverage, input coverage, and use case coverage, and mutation testing were used for the analysis.

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 tooling support requires to create test models (DSLs), automatically transform models into Alloy (model transformations), and execute test cases on the SUT (test driver and test harness).
- *Experiments.* We present results from a case study, and measuring the results using coverage criteria and mutation testing.

Outline. The remainder of this paper is organized as follows. Section 2 explains a 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.

2 Running Example

To illustrate the framework and the testing activities, e.g., creating and preparing test models, we use a simple library system throughout the paper.

In this system, a 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. We identify (and use) six use cases for the system:

1) Add Book
    - Input: book name
    - Pre-condition(s): the name should not be empty and duplicate.
2) Add Member
3) Borrow Book
    - Pre-condition(s): the book should be free to borrow and the member has not exceeded the limitation.
4) Return Book
5) Delete Book
    - Pre-condition(s): the book should be free (not borrowed).
6) Delete Member
    - Pre-condition(s): the member should not have any borrowed book.

3 Framework Overview

This section overviews our lightweight method for automatic model-based acceptance testing. Fig. 2 shows the overview of the proposed testing framework, which is built on Alloy. All the models are translated into Alloy models which are then solved in order to find test cases; each solution is an execution trace of the SUT representing a test case. The required models are partitioned into *Primary models* and *Subsidiary models*. The *primary models* are created from the description of use cases (in natural language) and serve as the main test models for test generation. The *subsidiary models* are required in aspects of test execution such as referring to domain objects and resetting the SUT at the beginning of each test run. Additionally, the framework requires a *test harness* that is developed by the tester and is specific to the SUT. The test harness consists of three parts: *data generator*, *command executor*, and *Inspector* and is responsible for the communications between the test driver and the SUT.

From testers' perspective, the following key steps are involved in test generation and execution by the presented framework.

1) The use cases and the domain model are defined throughout the general software development process.
2) The domain (structural) model is specified as EMF [18] models.
3) Input data types (and their possible partitioning) are identified and defined using the introduced DSL by the framework.
4) Use cases are described with the provided DSL.
5) The rules, applying on labels, and the initial state of the system are described with the given DSL.
6) The tester specifies the test goal.
7) The test harness is developed by the tester.
8) All of the above models are then automatically translated into Alloy.
9) Alloy finds a set of execution traces (i.e., test paths or test cases), with their required input values, and provides them to the test driver.
10) The test driver runs the system under test (SUT) based on the provided execution traces, by Alloy; finds the associated use cases to each test case and executes them in order.

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

3.1 Structural Model

The tester creates a structural model of the domain model. The model includes those domain classes that are essential in analysis and typically represent persistent data. Our approach uses the class diagram as defined by an EMF model conforming to Ecore metamodel.

In case the internal state of a domain object is relevant to the use cases, objects can be optionally associated with an "*object label*". A *label* is determined based on an object internal data or its relations with other objects. The labels are used to determine whether an object has a certain behaviour or not, and hence, they are especially used in the conditional expressions in modelling use cases (more details in Section 4.2). For example, an object may have "Active" and "Inactive" labels, and only *active* objects can participate in a use case. This concept is similar to "State" design pattern [19].

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 requirements (or use cases) describing effects of the use case on domain objects. Generally, use cases do not have required formalism which makes them insufficient for automatic test generation []. Our approach utilizes an internal domain specific language (DSL) within Java for describing use cases, which provides required formalism and remains familiar for the developer; the behavioural model is constructed via calling specific methods in a language that the developer is already familiar with (i.e., Java). We decided to introduce a new DSL for modelling the behaviour of a system as

- developers are already familiar with programming languages and modelling environments and thus, existing IDEs can be used for creating and maintaining our models as well,
- 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 [20], jMock[3], and Selenium [4], have been very successful in using DSLs for specifying tests.

The introduced DSL supports modelling of

1) input parameters (of the problem domain or other inputs) 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).

Note that use cases are inherently not object-oriented, and we assume that relating use cases and the domain model (e.g., mapping the input parameters and describing the effects of executing use cases) is not too complex (i.e., it can be done in a straightforward way). Additionally, as described in the previous section, "object labels" are used the conditional expressions of pre-/post-conditions.

3.3 Test Data

Generating test data is one the main issues in automatic test generation []. A prevalent approach for generating test data, among MBT methods, is input domain partitioning [21]. In this approach, the input domain is partitioned such that it is enough to test the system only for an arbitrary representative of every and each partition. This approach is particularly suited well to the kinds of system not involving complicated data manipulation and data 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 prescribed architecture by the framework.

3.4 Subsidiary Models

In addition to the aforementioned models, the testing approach relies on three other models that are classified as subsidiary models, as they are less important and are smaller than other models.

[3] jMock http://jmock.org

[4] Selenium - https://www.seleniumhq.org/

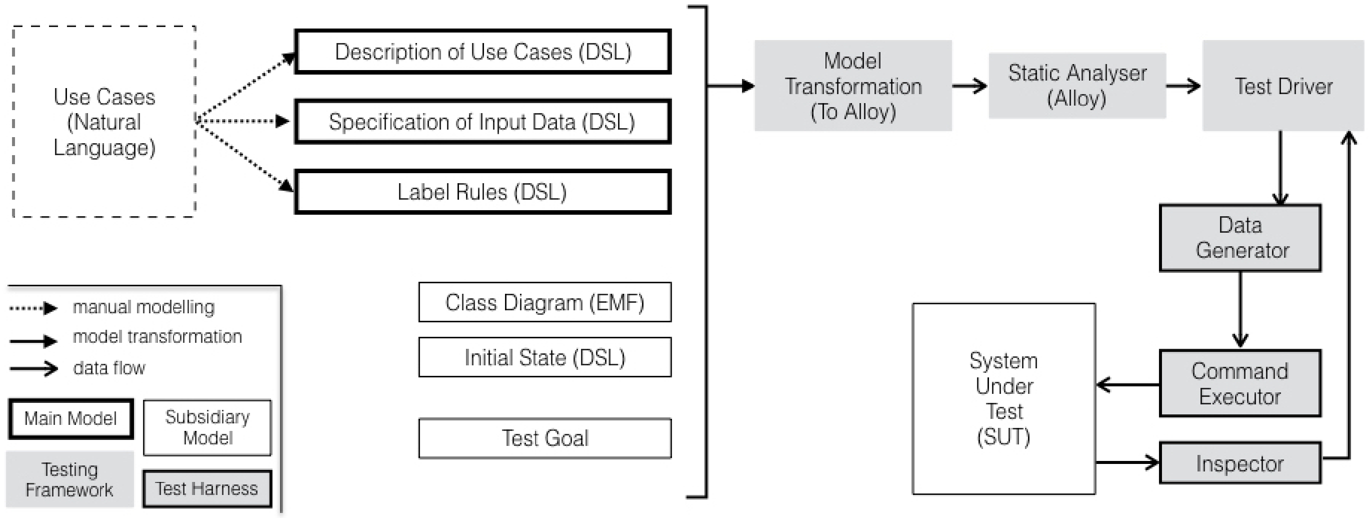


Fig. 2. The architecture of the proposed testing framework.

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) can not be specified in the behavioural model DSL. The framework includes a DSL for defining such dependencies in a separate model.

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. One goal is successful execution of use cases, and the other goal is running the system aiming at falsifying the pre-conditions of a use case. The search scope determines the maximum length of execution traces, and and is needed as we utilize a finite model generator (i.e., Alloy). We have included the ability for the tester to force the inclusion or exclusion of a specific use case in 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.

3.5 Model Transformer: to Alloy

As mentioned earlier, the framework has been built on Alloy and thus, all models are translated into Alloy models. This component translates the structural and behavioural models into Alloy models, which have the required formalism and constitute the main test models used for test generation.

3.6 Static Analyser: Alloy

The static analyser finds a set of execution traces based on the formal specification of the system (the generated Alloy models); each trace is principally a solution for the formal specification, and also represents a test case with its associated use cases. Our framework uses Alloy as the static analyser. Alloy has a high-level specification language in comparison to other model analysis tools based on CTL[5], LTL[6], and first-order logic. 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 the other approaches that are based on the set theory (e.g., Z [22]) have weak execution support. One of the advantages of Alloy is its ability to enumerate all possible solutions for a given set of constraints for which it is also called a Model Finder. In Alloy, the specification is converted into a Boolean SATisfiability problem (SAT) 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. This also limits the solver to finding small instances of the solution space. But, considering the "Small Scope Hypothesis" by Jackson [23], "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". This hypothesis has not been proved and also it is obvious that some errors only happen in large processing. However, the success of Alloy in modelling and testing indicates that the hypothesis is intuitive and is nearly valid.

3.7 Test Driver

The test driver executes the traces from the model finder on the SUT. A test case may be associated to a number of use cases with their input parameters. Each test case also specifies the expected state of the system after executing that test. A state of the system is represented by a snapshot of the objects of the domain model at a particular moment.

better term than cover/re

[5] CTL Computation Tree Logic
[6] LTL Linear Temporal Logic

The test driver finds the associated use cases for each test case and, after generating required input parameters (as prescribed by the test case), executes them on the SUT, via the test harness. After the execution of each use case, the current state of the SUT is compared to the expected state of the system (specified by the test case) Inconsistencies indicate a failed test, and test execution stops. Otherwise, the next use case is executed. The workflow of the test driver is illustrated in Fig. 3.

3.8 Test Harness

The test harness is an SUT-specific piece of code developed by testers to utilise the testing framework. All communications between the test driver and the SUT, including executing a use case and observing the behaviour of the SUT, are performed by this component. The test harness has two main parts:

1) the *command executor* that is responsible for executing use cases on the SUT. Similar to the "Command" design pattern, it provides the required interfaces to the test driver for executing use cases.
2) the *inspector* that allows the test driver to observe the internal state of the SUT, i.e., the domain objects and their labels.

It also includes the data generator that generates appropriate test data for use cases, using the test data model. Although the data generator is shown as a separate component in Fig. 1, it is not an independent components and is developed as part of "Command Executor".

In the following two sections, we describe the details of creating primary and subsidiary models, which are ultimately translated into Alloy specifications, and the test generation and execution workflow.

4 Creating Test Models

The framework relies on a set of primary and subsidiary models that have to be created or generated by testers and provided to the framework. These models are automatically translated into Alloy models which are the actual models used for the test generation and execution. This section describes how the required models are created or generated in the context of the example library system.

4.1 Structural Model

The structural model is created based on the EMF model of the SUT that only shows the structure of the class diagram; it does not include the internal data of the classes. The class diagram and the EMF model of the example library system is depicted in Fig. 4.

Using the Eclipse[7] IDE, the Java code of the EMF model is generated. This code is required and used in defining the behavioural model. The EMF model is also transformed into an Alloy specification (.als file), which is the actual structural model used in the testing process. The detail of model transformations into Alloy are provided in Section 5.3.

4.2 Behavioural Model

The first step in defining the behavioural model is to identify object labels, as labels are used in creating the behavioural model (defining use cases). The labels can be identified by considering the state-based behaviour of the system, 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.

After defining the object labels, the behavioural model is created which consists of description of use cases with the introduced DSL. Fig. 6 shows the description of "Borrow Book" use case using the DSL.

Each use case is defined with a Java class which is annotated as `@SystemOperation`, which is used by the framework to identify the system use cases. A use case description specifies one or more execution paths that are defined as a method using the signature shown in line 3 in Fig. 6. These methods are annotated as `@Description` and return an object of type `ModelExpectations`.

The core of the behavioural modelling is performed in the configuration of this object (line 5 to 12). Firstly (lines 5-7), the parameters and the pre-conditions of the use case are defined and introduced to the framework. Line 5 shows that there is a parameter of type `BOOK` called `thisBook`, in the use case (the words in capital letters refer to the domain model Java code generated by EMF). The *inState* method specifies the permissible labels for an object. Line 5 of Fig. 6 indicates that the book to borrow has not been already borrowed; its label should be `Not Borrowed`. Similarly, line 6 and 7 describe the parameter of type `Member` and its permissible labels. In the second part (line 9 to 12), the post-conditions of the execution path of the used case are described, such as changing the state to a specific state (line 9) or an arbitrary state (line 10), and adding a relation to the specified relation type (line 11).

4.3 Test Data

One of the main, and challenging, tasks in testing is test data generation. This sections describes how data is generated for use cases that needs input data. As mentioned before 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 introduced to the framework. For example, the "Add Book" use case requires the input parameter *bookName* that should be automatically generated. A book name is a string and is partitioned into "empty", "duplicate", and "unique".

For defining the partitioning of a data type, the data type has to be defined and introduced to the framework. A data type is an implementation of the `DataFactory` interface in which each partition is represented by a method annotated as `@Partition` and returns an object of type `PartitionDescription`. Each `PartitionDescription` is responsible for data generation for its own partition, in its `generate` method. Fig. 7 shows the definition of `BookName` data type and its partitioning namely "empty", "unique", and "nonUnique". In lines 31-41, the partitioning is defined. Each partition

[7] Eclipse - http://eclipse.org

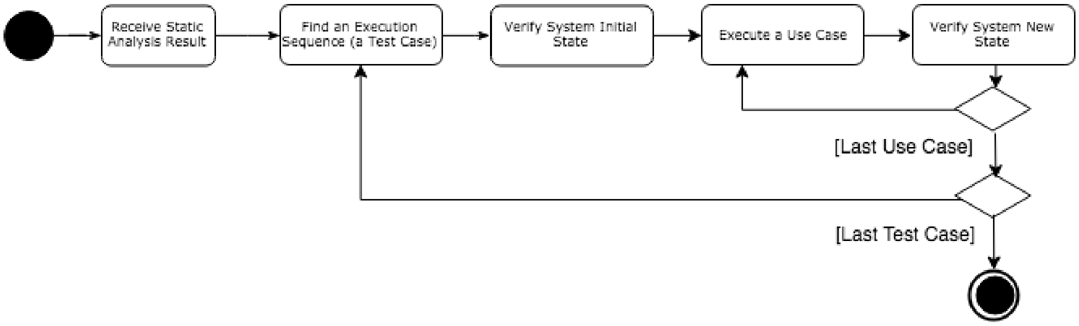
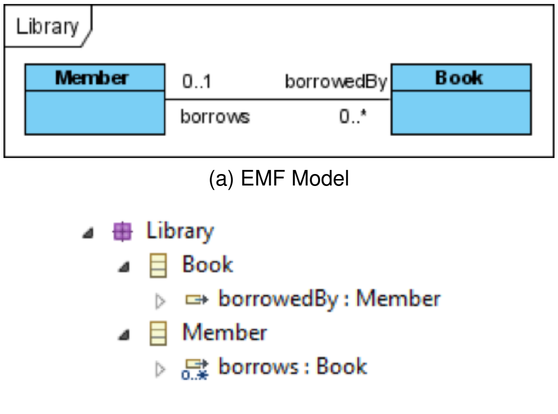

Receive Static Analysis Result
Find an Execution Sequence (a Test Case)
Verify System Initial State
Execute a Use Case
Verify System New State
[Last Use Case]
[Last Test Case]

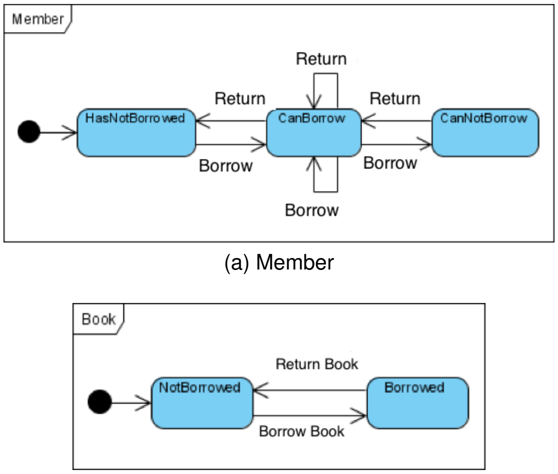
Fig. 3. The workflow of the test driver.



(a) EMF Model

(b) Class Diagram

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



(a) Member

(b) Book

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

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.

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

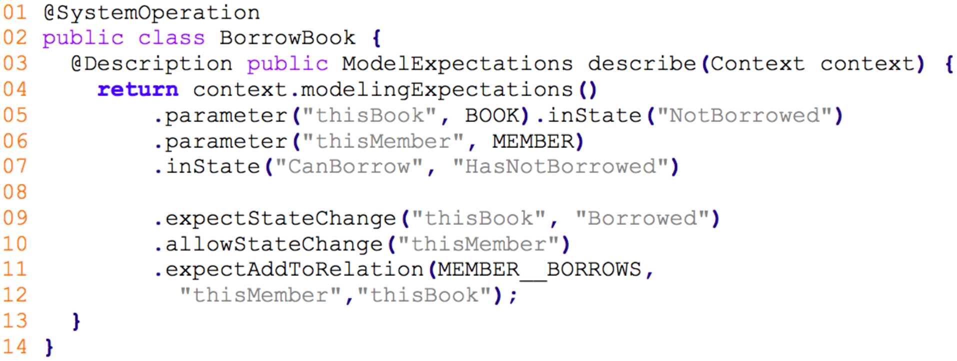
The test harness, an SUT-specific code developed by the tester, is responsible for the communications between the test driver and the SUT, including executing use cases, observing the internal state of the SUT, which are performed by "Command Executor" and "Inspector" components respectively. The test harness also includes the data generator that generates appropriate test data for use cases. The generator is developed in the context of the "Command Executor".

4.4.1 Introducing the SUT

In order to introduce an SUT to the framework, the `SoftwareSystem` interface is defined and provided to testers. The test harness introduces an object that implements this interface to the testing framework as the system under test. Later on, this object is used for executing commands and inspection; it is passed to these components. The `SoftwareSystem` interface has a `reset` method which is for bringing the system into its initial state. This interface and its implementation for the example library system is shown in Fig. 9.

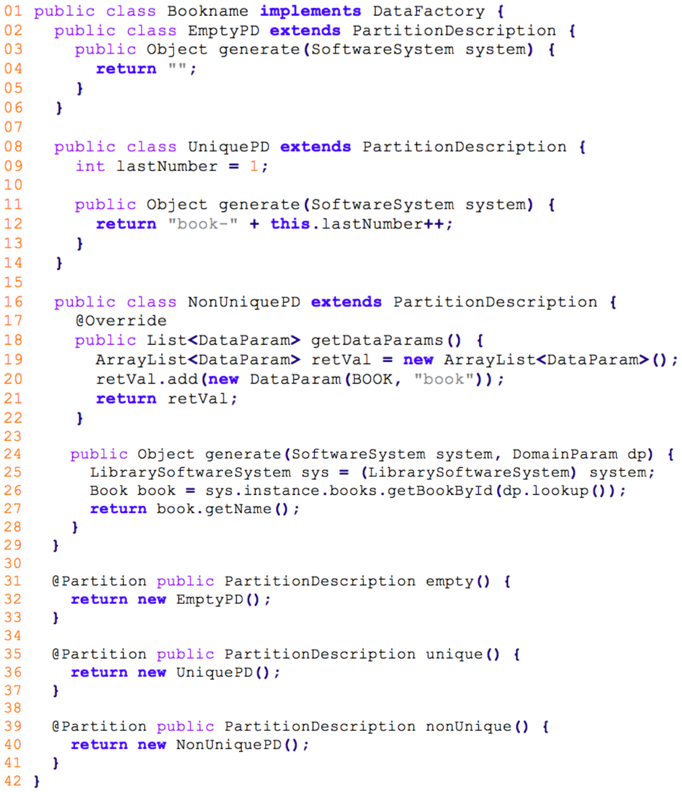
4.4.2 Inspection

The test harness has to be able to return a list of the objects, with their associated labels, for each class in the domain model. Each object is introduced with a unique identifier (ID) to the testing framework. The identifiers are managed by the test harness and the SUT. Accordingly, an abstract super class (`AbstractDomainObjInspector`) is provided that is inherited by the test harness. Fig. 10 shows an example implementation of this super class and the inspector for the `Member` class.



```
@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.



```
public class Bookname implements DataFactory {
  public class EmptyPD extends PartitionDescription {
    public Object generate(SoftwareSystem system) {
      return "";
    }
  }

  public class UniquePD extends PartitionDescription {
    int lastNumber = 1;

    public Object generate(SoftwareSystem system) {
      return "book-" + this.lastNumber++;
    }
  }

  public class NonUniquePD extends PartitionDescription {
    @Override
    public List<DataParam> getDataParams() {
      ArrayList<DataParam> retVal = new ArrayList<DataParam>();
      retVal.add(new DataParam(BOOK, "book"));
      return retVal;
    }

   public Object generate(SoftwareSystem system, DomainParam dp) {
     LibrarySoftwareSystem sys = (LibrarySoftwareSystem) system;
     Book book = sys.instance.books.getBookById(dp.lookup());
     return book.getName();
   }
  }

  @Partition public PartitionDescription empty() {
    return new EmptyPD();
  }

  @Partition public PartitionDescription unique() {
    return new UniquePD();
  }

  @Partition public PartitionDescription nonUnique() {
    return new NonUniquePD();
  }
}
```

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

The @Inspector annotation with the name of the target type, in line 1, allows the framework to identify the inspector for the target domain class (e.g., “Member” in Fig. 10) . The constructor, line 5-7, takes an object implementing SoftwareSystem, which is the object representing the SUT (mentioned in Section 4.4.1). 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.

4.4.3 *Command Execution*

The test harness implements the Command interface (Fig. 11) for each use case so that the use case is related to the SUT and it is possible to run the test cases on the SUT. Each use case description class should have a 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.

An implementation of the command method, for the “Borrow Book” use case, has been shown in Fig. 12, 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.

Additionally, 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

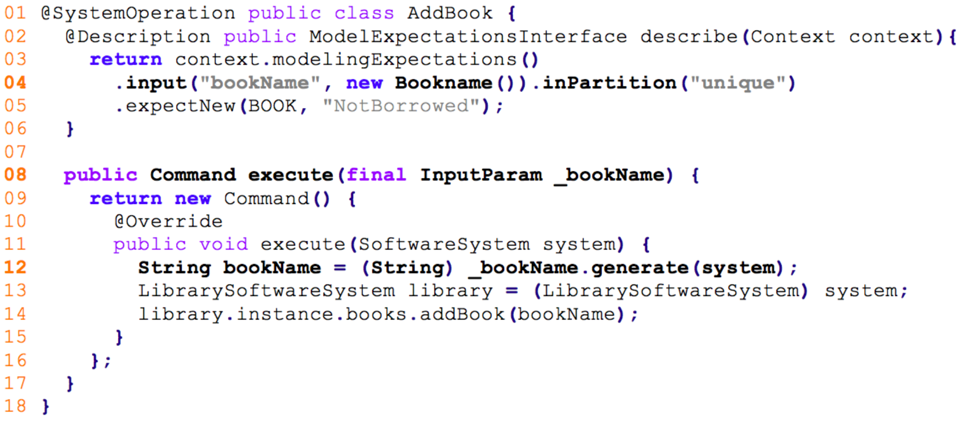
The testing framework also relies on three subsidiary models that are smaller and less important than other models, namely the initial state, object labels rules, and test goals. There are two built-in test goals including 1) successful execution of use cases, and 2) running the system aiming at falsifying the pre-conditions of a use case. The (user defined) test goals are specified in the Alloy models and thus the test goals are not discussed (covered) in this section.

4.5.1 *Initial State*

This model specifies the objects in the initial state, with their labels and the relations between them. Fig. 13 specifies an initial state for the example library system. The initial state model consists of one book and one member that has borrowed the book.

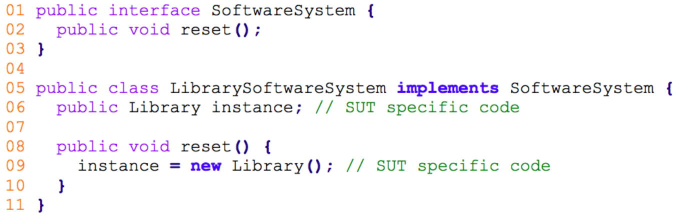
4.5.2 *Object Labels Rules*

As mentioned before, certain constraints on relations between objects (e.g., number of involved objects in a relation) can not be specified in the behavioural model. For example, in the modelling of the post-conditions of the“Borrow Book”



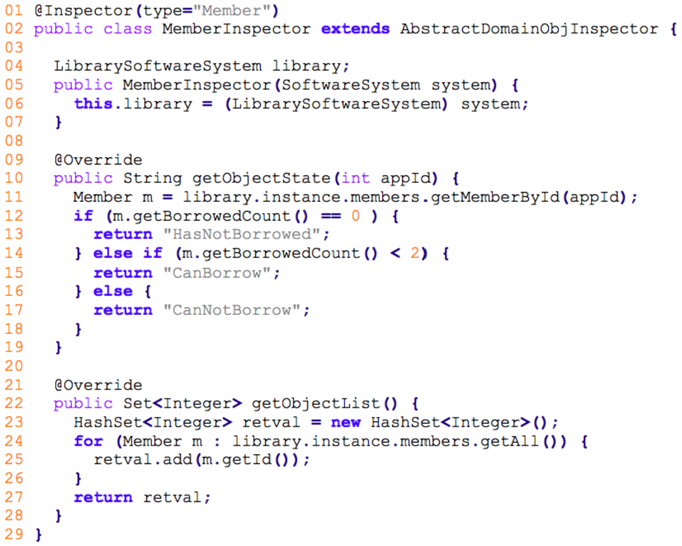
```java
@SystemOperation public class AddBook {
  @Description public ModelExpectationsInterface describe(Context context){
    return context.modelingExpectations()
      .input("bookName", new Bookname()).inPartition("unique")
      .expectNew(BOOK, "NotBorrowed");
  }

  public Command execute(final InputParam _bookName) {
    return new Command() {
      @Override
      public void execute(SoftwareSystem system) {
        String bookName = (String) _bookName.generate(system);
        LibrarySoftwareSystem library = (LibrarySoftwareSystem) system;
        library.instance.books.addBook(bookName);
      }
    };
  }
}
```

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



```java
public interface SoftwareSystem {
  public void reset();
}

public class LibrarySoftwareSystem implements SoftwareSystem {
  public Library instance; // SUT specific code

  public void reset() {
    instance = new Library(); // SUT specific code
  }
}
```

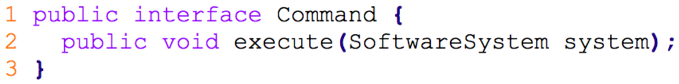
Fig. 9. The `SoftwareSystem` interface and its implementation for example library system.



```java
@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. 10. An example inspector, for the “Member” class.

use case (Fig. 6), the label of the object `thisMember` can not be determined as it is dependant on its label before the execution of the use case. Such dependencies are defined in the models describing the rules applied on the objects to determine their labels. Fig. 14 illustrates the rules applied on the “Member” class and its labels. In line 3, the `setFor` method specifies the rules are applied for which class. Line 4 indicates that if the given relation (MEMBER_BORROW) is empty then the label is `HasNotBorrowed`. Similarly, line 5 says if the size 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 are applicable.



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

Fig. 11. `Command` interface.

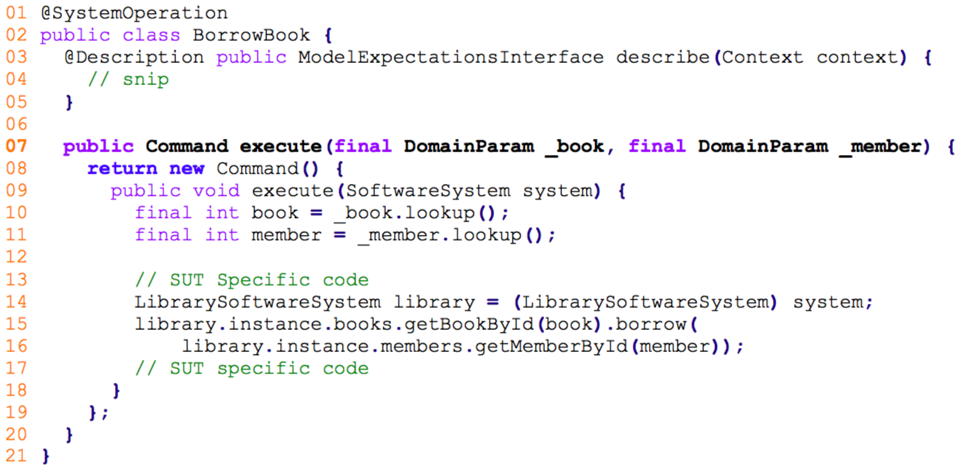
4.6 Introducing the Models to the Framework

All of the aforementioned models have to be introduced to the testing framework. To do so, each type of models are bundled into a package which are introduced to the framework by implementing the `SUT` interface. The implementation of the SUT is also introduced to the framework.Doing so, the testing process can be carried out. The testing framework uses ‘reflection’ to find the required classes (and methods) to execute the test cases (i.e., corresponding use cases) on the system. The `SUT` interface is depicted in Fig. 15.

5 Test Generation and Execution

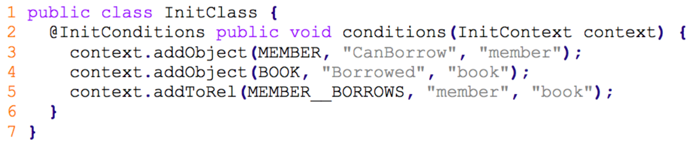
The previous section introduces the test models, required by the testing framework, and explaines how they are created or generated by a tester. This section explains the test generation and execution workflow. The testing framework has been built upon Alloy [17] and the Alloy Analyser (AA) [24] with 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 [25] instances for that specification. Next, the testing framework uses reflection to translate these instances



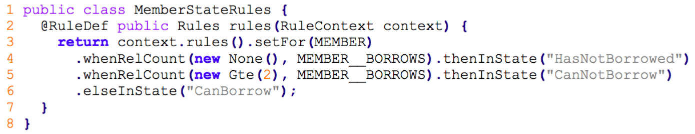
```
@SystemOperation
public class BorrowBook {
  @Description public ModelExpectationsInterface describe(Context context) {
    // snip
  }

  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. 12. 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. 13. 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. 14. Modelling of rules applied in “Member Class”.

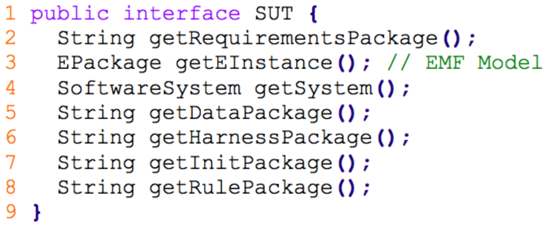
to real execution paths and concrete inputs, which form the executable test cases for the SUT.

In the following, we first describe the basics of the Alloy specification language and the Alloy Analyser; details can be found in [17], [23], [24]. 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 is associated with a basic type. An Alloy specification is a sequence of paragraphs that can be of two kinds: signatures, used for construction of new *types*, and a variety of formula paragraphs, used to record *constraints*. Each specification starts with a `module` declaration that names the specification.

A *signature* paragraph represents a basic (or uninterpreted) type and introduces an independent top-level sets. A signature declaration may include a collection of relations (that are called *fields*) in it 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 the constraints on 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.



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

Fig. 15. `SUT` interface.

Fig. 16 shows a simple alloy model for Linked List. The model defines two signature: `List` and `Node`. The `List` signature has a field called *header* showing the first element on the list. Line 9-12 define the facts on linked lists, indicating that all *Nodes* are either the first element in a list or accessible by traversing the `next` field on a list. In this declaration, “+” , “.”, and “^” are, respectively, operators for set union, relational composition, and transitive closure. Line 11 says that Linked Lists do not have a cycle. Lines 14-16 indicate that there is at least one Linked List in the solutions. Finally, in line 18, the main predicate used for finding the solutions, along with the search scope, is specified.

The Alloy Analyser [24] 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*. For the example alloy model in Fig. 16 with the given scope, there are seven solutions shown in Fig. 17. Each atom is in a signature and the relations between atoms are instances of their fields.

```
sig List {
  header : Node
}

sig Node {
  next: lone Node
}

fact {
  Node in List.header + List.header.^next
  no n : Node | n in n.^next
}

pred showLists {
  some List
}

run showLists for 2
```

Fig. 16. Example Alloy model of Linked List.

5.2 Generated Alloy Model

The test models, created/defined by testers, are automatically translated into Alloy specifications consisting of six `modules`, that are defined in separate Alloy files (.als). The modules and their dependencies are shown in Fig. 18, 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).

In the rest of this section, the details of the model transformation for each module is described.

5.3 Structure Model

The EMF model of the system is automatically translated into an Alloy specification, using the Xpand[8] model-to-text transformation language. Fig. 19 shows the automatically generated Alloy specification for the structure of the example library system. Each class is translated into a signature within which each association is represented by a field (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, and for modelling inheritance, the same concept in Alloy is used.

Alloy does not have a built-in notion of state mutation; it does not allow to change the relationships, whereas, in the context of testing, the domain model would be changed because of executing the use cases. Consequently, the concept of changing state has to be explicitly defined in the Alloy specification. To this end, the `state` signature is defined and all the relations that would change during test executions, are in a relation with an atom of this signature. This is shown by putting the extension `-> State` at the end of the changing relations.

The `System` signature represents the system under test and keeps track of the objects that are in the domain model at any time. It defines a field for each concrete class in the domain model (e.g., Fig. 19, lines 1-4).

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

In order to confine the state space and prevent useless solutions, for testing, the objects that are not in the domain model can not be participated 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. 20.

5.4 Operations: Use Cases

The next step is to translate the use case descriptions into Alloy. Basically, each use case is translated into a corresponding signature and a predicate. Firstly, the signature is used for tracing the execution flow: a solution does not show which predicate in the specification has resulted in the solution. Whereas, the test driver needs to know which predicate has been selected at any state. Thus, by using *conjunction*, an atom of the signature is attached to each predicate, so that it is possible to trace the execution flow. Secondly, the use case signature consists of the fields corresponding to its parameters. The signature for the “Return Book” use case is depicted in Fig. 21. Lines 5-8 are for confining the state space.

The corresponding predicate for each use case, takes the atoms of the previous and subsequent states, and an atom of the signature of the use case, and defines the constraints for creating the subsequent state. The predicate for the “Return Book” use case is shown in Fig. 22. This predicate consists of the following parts: defining the domain of the input parameters (lines 2-5), defining pre-conditions (lines 6-10), specifying the frame conditions (lines 11-20), and defining the post-conditions (lines 21-24).

The predicate of a use case may also contains 1) a part for defining variables (by 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. 23 showing the predicate for the “Add Member” use case. New objects are chosen such that choosing a new object does not result in different solutions by Alloy, which is done by defining a total order on the signatures for all classes. At any time, the smallest atom that does not exist in the domain is returned as the new object (line 2).

In addition to the corresponding predicate, a negative predicate with the prefix **fail_** is generated in order to test the exceptions in the system. The predicate is similar to the main predicate except that it does not define the post-conditions and the new objects parts. It does not make

[8] Xpand http://eclipse.org/modeling/m2t/?project=xpand

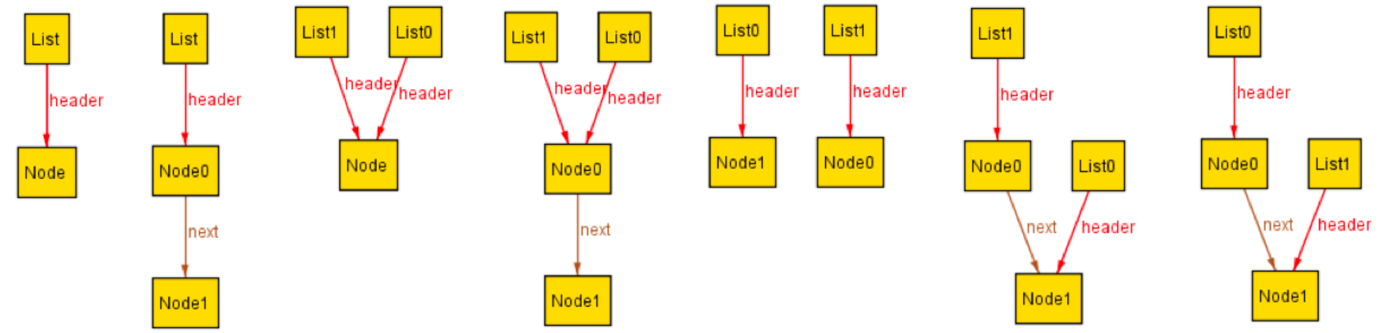


Fig. 17. Solutions for the example Alloy model of Linked List.

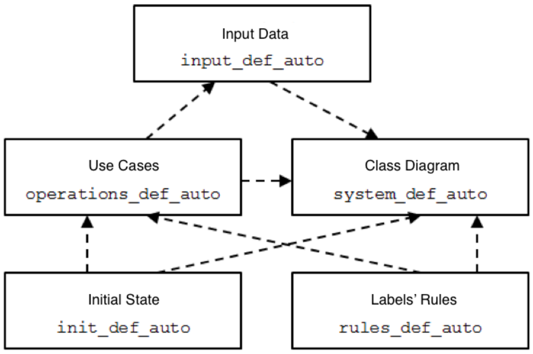


Fig. 18. Generated Alloy specifications and their dependencies.

```
one sig System {
  book: Book -> State,
  member: Member -> State,
}
abstract sig DomainObjState{}
abstract sig DomainObj {
  state : DomainObjState -> State,
}
sig Book extends DomainObj {
  rel_borrowedBy : Member lone-> State,
}
fact {
  all s: State | rel_borrowedBy.s = ~(rel_borrows.s)
}
abstract sig BookState extends DomainObjState {}
fact {Book.state.State in BookState}
sig Member extends DomainObj {
  rel_borrows : Book -> State,
}
fact {
  all s: State | rel_borrows.s = ~(rel_borrowedBy.s)
}
abstract sig MemberState extends DomainObjState {}
fact {Member.state.State in MemberState}
```

Fig. 19. Alloy specification of the structure of the example library system.

any change in the domain model and only works on its pre-conditions, that are the negative of the original pre-conditions.

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.

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

```
sig ReturnBookOp extends Op {
  returnBookOp_thisMember : one Member,
  returnBookOp_thisBook : one Book,
}
fact { no disj op1, op2 : ReturnBookOp | {
   op1.returnBookOp_thisMember = op2.returnBookOp_thisMember
   op1.returnBookOp_thisBook = op2.returnBookOp_thisBook
}
```

Fig. 21. The signature for the "Return Book" use case.

5.5 Initial State

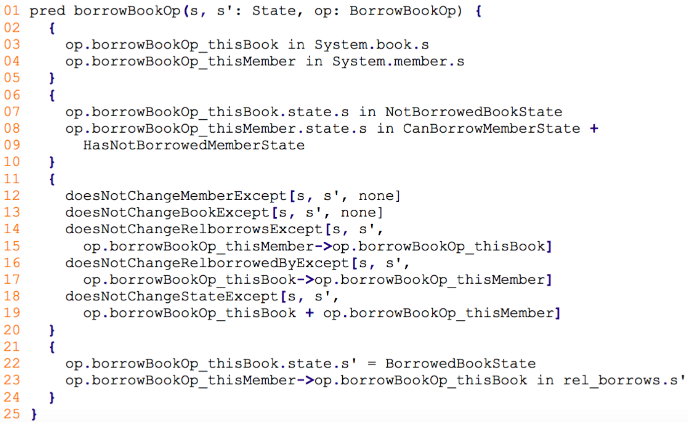
For transforming the initial state model into an Alloy specification, the size of the fields are specified. Also, for simplifying the solution space, the smallest possible object in the initial state is defined. Fig. 24 shows the translation of the initial model in Fig. 13. Lines 2-4 define a book that has been borrowed, lines 5-7 defines a member that can borrow books, and lines 8-10 define the association between these two objects.

5.6 Labels Rules

The Alloy specification for the object labels rules are generated based on our internal Java-based DSL introduced in Section 4.5.2. Fig. 25 shows the translation of the model in Fig. 5.

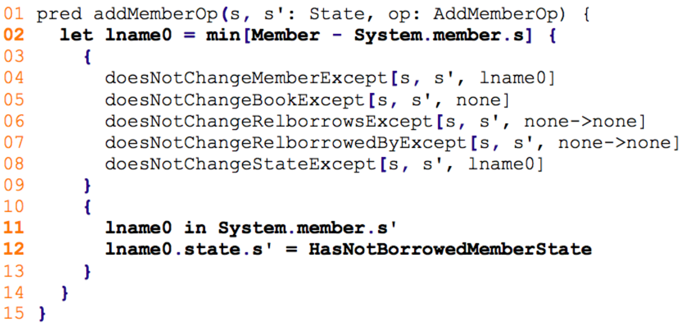
5.7 Input Partitioning

The partitionings of the input domain is also specified in Alloy. The specification for the input partitioning in the example library system, is depicted in Fig. 26. 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).



```
pred borrowBookOp(s, s': State, op: BorrowBookOp) {
  {
    op.borrowBookOp_thisBook in System.book.s
    op.borrowBookOp_thisMember in System.member.s
  }
  {
    op.borrowBookOp_thisBook.state.s in NotBorrowedBookState
    op.borrowBookOp_thisMember.state.s in CanBorrowMemberState +
      HasNotBorrowedMemberState
  }
  {
    doesNotChangeMemberExcept[s, s', none]
    doesNotChangeBookExcept[s, s', none]
    doesNotChangeRelborrowsExcept[s, s',
      op.borrowBookOp_thisMember->op.borrowBookOp_thisBook]
    doesNotChangeRelborrowedByExcept[s, s',
      op.borrowBookOp_thisBook->op.borrowBookOp_thisMember]
    doesNotChangeStateExcept[s, s',
      op.borrowBookOp_thisBook + op.borrowBookOp_thisMember]
  }
  {
    op.borrowBookOp_thisBook.state.s' = BorrowedBookState
    op.borrowBookOp_thisMember->op.borrowBookOp_thisBook in rel_borrows.s'
  }
}
```

Fig. 22. The predicate for the “Return Book” use case.



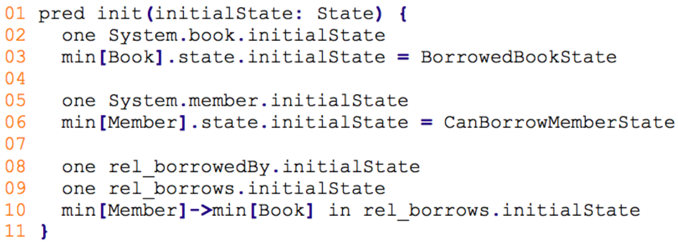
```
pred addMemberOp(s, s': State, op: AddMemberOp) {
  let lname0 = min[Member - System.member.s] {
    {
      doesNotChangeMemberExcept[s, s', lname0]
      doesNotChangeBookExcept[s, s', none]
      doesNotChangeRelborrowsExcept[s, s', none->none]
      doesNotChangeRelborrowedByExcept[s, s', none->none]
      doesNotChangeStateExcept[s, s', lname0]
    }
    {
      lname0 in System.member.s'
      lname0.state.s' = HasNotBorrowedMemberState
    }
  }
}
```

Fig. 23. The predicate for the “Add Member” use case.

5.8 Main Module and Test Goals

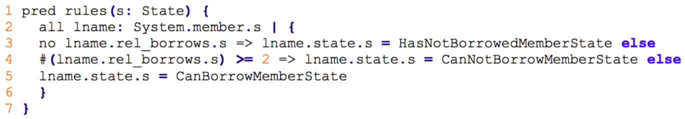
The main module is static and is independent of the SUT and its models. The module is depicted in Fig. 27. In the beginning, the module is defined and other modules are opened (lines 1-7). Then, the signatures for the system states and the abstract signature Op, used for tracing use cases, are defined (lines 9-12). Next, in line 15-17, it is indicated that in all the states, excluding the final state, a use case has to be executed. The predicate apply (given in Fig. **??**) decides which predicate, corresponding to the use cases, has to be applied on the model. Line 19 applies the rules on the labels and line 20 confines the state space and prevents from creating useless atoms Op. In lines 25-27, the initial state of the system is described, and the search scope is defined in lines 29-30. The additional, user-defined constraints in the Alloy model are defined in the predicate testSystem, if required. In this example, the search scope is 5 atoms for all signature, excluding the atoms for the State signature (line 30). The scope for the State signature is four that results in execution traces of length three (one state is four the initial state).

If the test goal is to generate exceptional states, the line 21 is commented out and line 22 is uncommented, and thus, the predicate fail_apply is applied on the final state. The predicate is similar to predicate apply except that it includes the negative predicates for the use cases. In this case, the true paths of length three are tested and then an exceptional state happens (final state). This is because that exceptions can not alter the domain model and hence, examining them in the middle of a path would be useless.



```
pred init(initialState: State) {
  one System.book.initialState
  min[Book].state.initialState = BorrowedBookState

  one System.member.initialState
  min[Member].state.initialState = CanBorrowMemberState

  one rel_borrowedBy.initialState
  one rel_borrows.initialState
  min[Member]->min[Book] in rel_borrows.initialState
}
```

Fig. 24. The initial state in Alloy.



```
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. 25. The object labels rules in Alloy.

5.9 Generated Test Cases

For the example library system, the Alloy generates 73 test cases with the given search scope in Fig. 27. One of the test cases is depicted in Appendix A, Fig. 32. The active use case in each state is shown by diamond. The atom System is shown by gray rectangle and the atoms of the domain model are shown by bold rectangle. The labels and input data are respectively shown by eclipse and dashed lines.

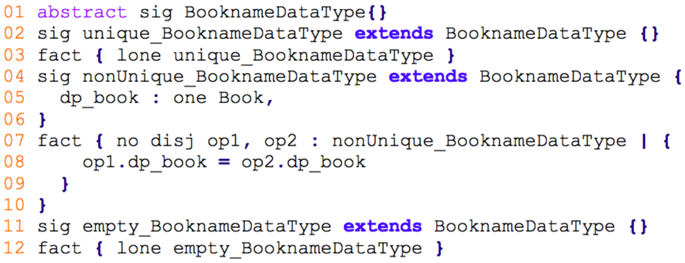
If the test goal is to generate exceptions (i.e. using the fail_apply predicate) for the search scope of size one, five test cases are generated which are shown in Appendix A. Fig. 33 shows the test case that removes a member that has a book in loan. Similarly, in Fig. 34, a book that has been borrowed is deleted. Fig. 35 illustrates an invalid borrow operation. Fig. 36 and Fig. 37 tends to add a book with empty name and a book with a duplicate name (Book0), respectively.

6 Evaluation

To assess the effectiveness and applicability of our approach, we have first conducted a case study on a non-trivial medium-sized business application. Then, we evaluate the outcome based on different coverage criteria and mutation testing technique.

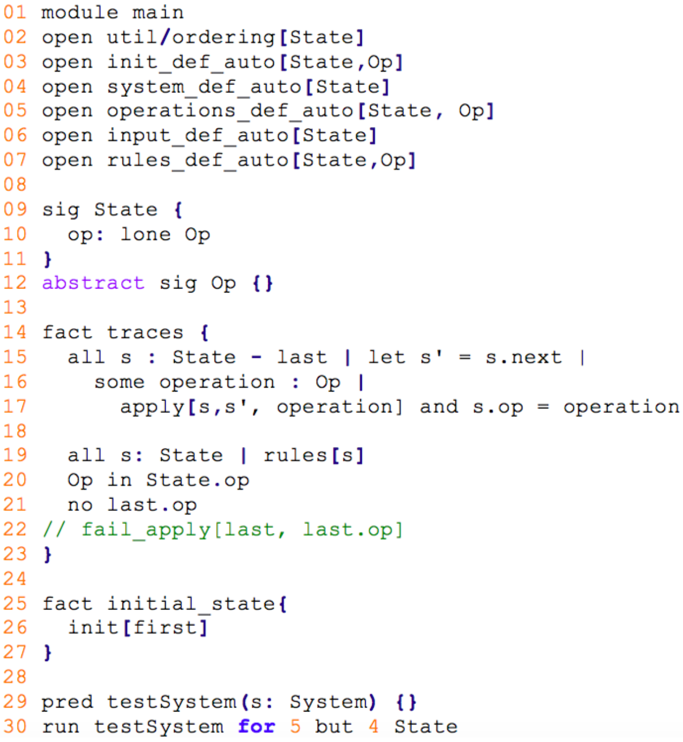
6.1 Case Study

The subject is an organisation internal mailing system which is a windows-based desktop application. The system was developed throughout an iterative-incremental process using Java language. The code for the business logic consists of about 1000 LOC and the UI code has about 6000 LOC. The requirements and use cases were documented in the analysis phase, and most of the use cases are about recording and manipulating the internal mails. Consequently, the system was appropriate for using our testing framework. Moreover, as the system was developed incrementally, the system was approved (and verified) by the customer at the end of each iteration, and hence, the system has acceptable quality (i.e.,



```
abstract sig BooknameDataType{}
sig unique_BooknameDataType extends BooknameDataType {}
fact { lone unique_BooknameDataType }
sig nonUnique_BooknameDataType extends BooknameDataType {
  dp_book : one Book,
}
fact { no disj op1, op2 : nonUnique_BooknameDataType | {
    op1.dp_book = op2.dp_book
  }
}
sig empty_BooknameDataType extends BooknameDataType {}
fact { lone empty_BooknameDataType }
```

Fig. 26. The input partitioning in Alloy.



```
module main
open util/ordering[State]
open init_def_auto[State,Op]
open system_def_auto[State]
open operations_def_auto[State, Op]
open input_def_auto[State]
open rules_def_auto[State,Op]

sig State {
  op: lone Op
}
abstract sig Op {}

fact traces {
  all s : State - last | let s' = s.next |
    some operation : Op |
      apply[s,s', operation] and s.op = operation

  all s: State | rules[s]
  Op in State.op
  no last.op
// fail_apply[last, last.op]
}

fact initial_state{
  init[first]
}

pred testSystem(s: System) {}
run testSystem for 5 but 4 State
```

Fig. 27. The Alloy main module.

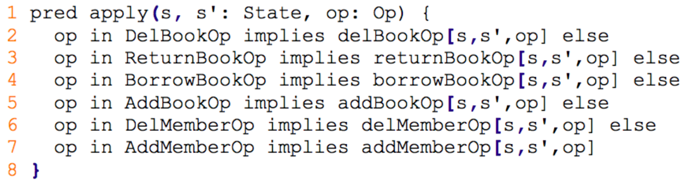
accepted by the customer). The detail description of the case study is available in the master thesis of the first author [26].

In the case study, we have focused on 15 use cases, depicted in Fig. 29. The system has other use cases such as “View Roles” and “Aggregated Report”, which do not make any change on the domain model and thus they were excluded in our study.

Structural Model. The structural model is shown in Fig. 30 which was created based on the analysis structural model created for the system in the analysis phase.

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

- Logical two-values; it is used for accepting or rejecting a process, and also defining the state of a mail, namely “waiting” and “for information”. It has two partitions of “true” and “false”.
- String; it is used for string inputs such as mail’s content 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.
- User Name: 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”.



```
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. 28. The `apply` predicate for the example library system.

Object Labels. Seven labels were identified and defined 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 do not have state-based behaviour and thus no labels were defined for them.

Subsidiary Models. Due to limited data dependencies, there is no rules for determining the labels, and thus there is no model of the rules. The only subsidiary model is the model describing the initial state. We used the initial state depicted in Fig. 31 in the testing experiment.

Test Harness. The test harness was developed for our case study and it consists of about 300 LOC. For this case, we used the uispec4j[9] in developing the components responsible for automatic communication with the UI. Additionally, most of the existing code were reused for developing the “Inspector” component.

Behavioural Model. Throughout modelling the use cases, we found out that there were sever analysis problems in three use cases, namely “Remove User”, “Delete Role”, and “Remove Role”, which stop the execution of the SUT and subsequently, the testing process. These use cases were not dealt with correctly; the related letters to a user or role, were not kept in the system in case of removing the user, deleting the role, or removing the role from the user, while it was explicitly specified in the requirements to keep the letter for other users and roles. Consequently, we excluded these use cases from our study. The behavioural model included about 170 LOC in the internal DSL.

6.1.1 Test Execution and Result

After generating or 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 and 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, which could be improved by increasing the number of threads.

Although the system was verified and approved by the customer throughout several iterations, we found out

[9] uispec4 http://uispec4j.org

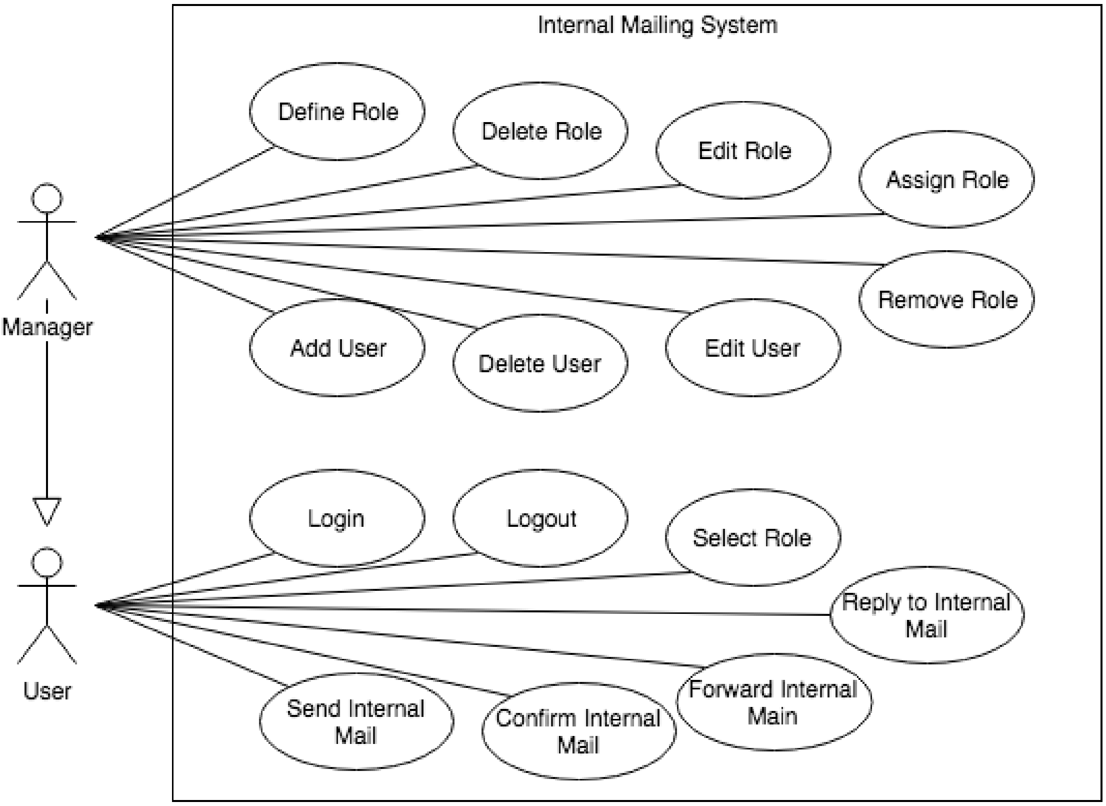


Fig. 29. Use case diagram for the case study.

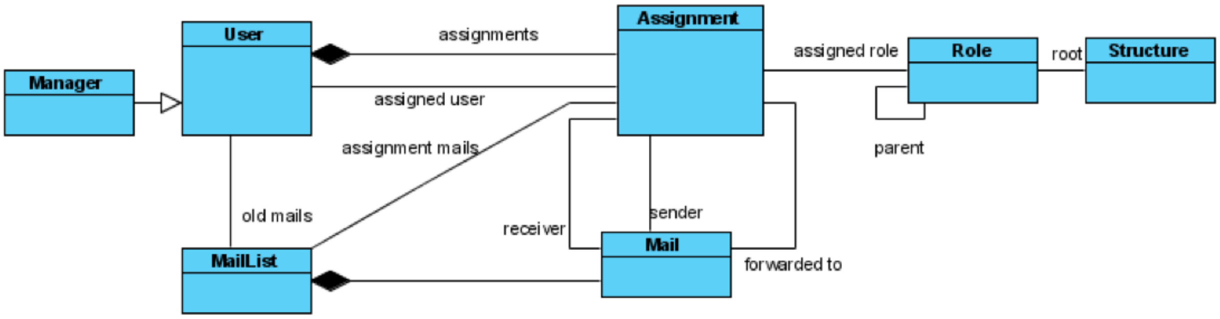


Fig. 30. Structural model of the case study.

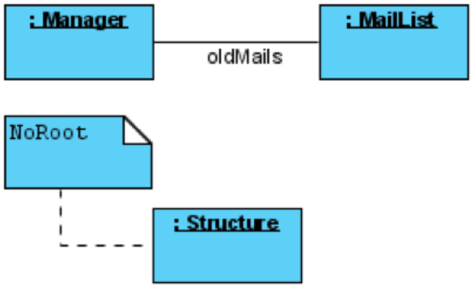


Fig. 31. Initial State for the case study.

quite significant problems in 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, in some cases, is explicitly forbidden (e.g., user name, password, and role name) and, in some cases, is unclear if it is permissible or not, due ambiguity in requirements (e.g., content and title of a mail). We also found out that the system allows a role sends an internal mail to herself, while the system does not create any internal mail[10] .

The problems in the analysis were explained in Sec 6.1. In requirements specification, it was not clear if an internal mail to the direct upper manager needs to be approved or not[11].

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

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

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)

TABLE 2
Line coverage for the case study.

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

6.2 Line Coverage

To evaluate the line coverage of our testing framework, we used EMMA[12], which manipulates the executable bytecode to find out which line is executed or not. The line coverage of the aforementioned test cases are shown in Table 2. The "Unrelated" refers to the classes for the use cases excluded in the study, which has not been specified for the business logic, as it was hard and impossible to clearly determine *unrelated* code.

Excluding the "Unrelated" code, the coverage for UI and business logic were %90 and %73 which results in total coverage of %90, achieved without writing any test script, and is a high coverage for a non-trivial system.

6.3 Mutation Testing

In order to assess the effectiveness of the proposed testing framework, mutation testing was used, in addition to line coverage. To do so, the PIT[13] mutation generator tool was used. The main reason for choosing this tool was its applicability on large projects. The tool analyses the execution flow of use cases and accordingly, optimises the mutation generation processes, by determining proper mutation locations, prioritising the executions of use cases, and finding use cases for mutations. Another important feature of this tool is that the mutations are done on Java bytecode.

The default 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 test cases used for line coverage (Table 1) were used for mutation testing, as well. The summary of the result is provided in Table 3 and Table **??**, showing the results respectively based on the mutation operators and the modules. Overally, %56 of the mutants were killed while the mutations in the lines that were not covered by the test cases, have been considered. Accordingly, having these mutants excluded, the mutation score will be %68 and %79 respectively, for the UI and business logic modules. The low mutation score for the UI module is due to difficulties in identifying 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.

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

6.4 Use Case Coverage

A very related coverage criteria to the testing framework is the *use case coverage* which is classified into two types: 1) each use case has to be executed, at least, one time regardless of the execution paths inside the use case, and 2) all execution paths inside each use case have to be covered.

The first type can be easily achieved by the proposed framework via (i) putting `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, and accordingly defines related signature for them. Then, Alloy Analyser is forced to cover specific paths by putting texttttsome [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,

[12] EMMA http://emma.sourceforge.net/
[13] PIT http://pitest.org/

an enough large search scope would result in complete coverage which is typically impossible in practice.

6.5 Coverage based on Input Partitioning

Another type of 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 extra partitioning for the input domain and considering them as input partitioning would result in the challenging problem of determining the search scope.

7 Discussion and Limitations

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. In this context, the framework can not be used 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.

Nevertheless, considering the target of this research that is testing data-oriented systems developed using lightweight development processes, these assumptions and limitations do not hamper (or limit) the application and benefits of the framework in practice. Our case study 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 would identify problems in analysis and ambiguities in requirements specifications, even throughout the development of a system that is continuously delivered and approved by the customer.

Additionally, considering the previous and similar studies in the literature, our framework advantages over them 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.

The work presented here is inspired by TestEra, presented by Khurshid et al. [44], in which tests are generated using Alloy. Our work however, substantially differs from TestEra: 1) TestEra focuses on unit testing (tests are defined at method level) whereas in our work, use cases and their execution sequence are used for test generation (i.e., acceptance testing), 2) in TestEra, the Alloy specifications are created manually and thus, the testers have to learn to model in Alloy, in contrast to the proposed framework, and 3) TestEra uses the heap memory and thus, is limited to executing time objects, while in our framework, the tester can define any approach for inspection.

Another similar work to ours is [14] in that tests are automatically generated based on 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 differences between this work and our framework.

In [27], a first-order logic language is used to define the 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, [28], 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. But, they are defined for unit testing. Also, testers (developers) would be more familiar with our internal Java-based DSL in comparison to virtual contracts.

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. [27] 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. [28] 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 [27] except in that the behaviour is modelled by a first order logic language.

Bouquet et al. [29] defines a subset of UML 2.1 [30] that allows formal behaviour models of the SUT. The subset uses class, instance and state diagrams, in addition to OCL [31] 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" [33] 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[14] 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' preconditions. 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 [34] 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 [35]. 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, 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 [35]).

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 [36].

Different specifications have been considered and used for automatic test generation, such as Z specifications [22], [37], [38], [39], UML statecharts [40], [41], ADL specifications [42], [43], Alloy specifications [17], [44], [45], JML specifications [46].

Horcher [37] presents a technique for software testing based on Z specifications [22]. 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 [43] introduce a technique for deriving test conditions–a set of boolean conditions on values of parameters–from ADL specifications [42]. These test conditions are used to guide test selection and to measure comprehensiveness of existing test suites.

Boyapati et al. [46], 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. [44] 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.

[14] 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

[14] JML Java Modeling Language http://jmlspecs.org

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 [14].

In a later work, Rosner et at. [47] 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 [46] 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 [44].

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 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 testing framework. 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] K. Beck, *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[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] 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

[5] 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

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

[7] 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.

[8] 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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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

[16] ——, "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.

[17] 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

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

[19] 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.

[20] 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

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

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

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

[24] 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

[25] 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

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

[27] 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.

[28] 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.

[29] 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

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

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

[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] 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

[34] 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.

[35] 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

[36] 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

[37] 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.

[38] 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

[39] 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

[40] J. Rumbaugh, I. Jacobson, and G. Booch, Eds., *The Unified Modeling Language Reference Manual*. Essex, UK, UK: Addison-Wesley Longman Ltd., 1999.

[41] 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

[42] 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

[43] 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

[44] 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.

[45] 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.

[46] 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

[47] 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.

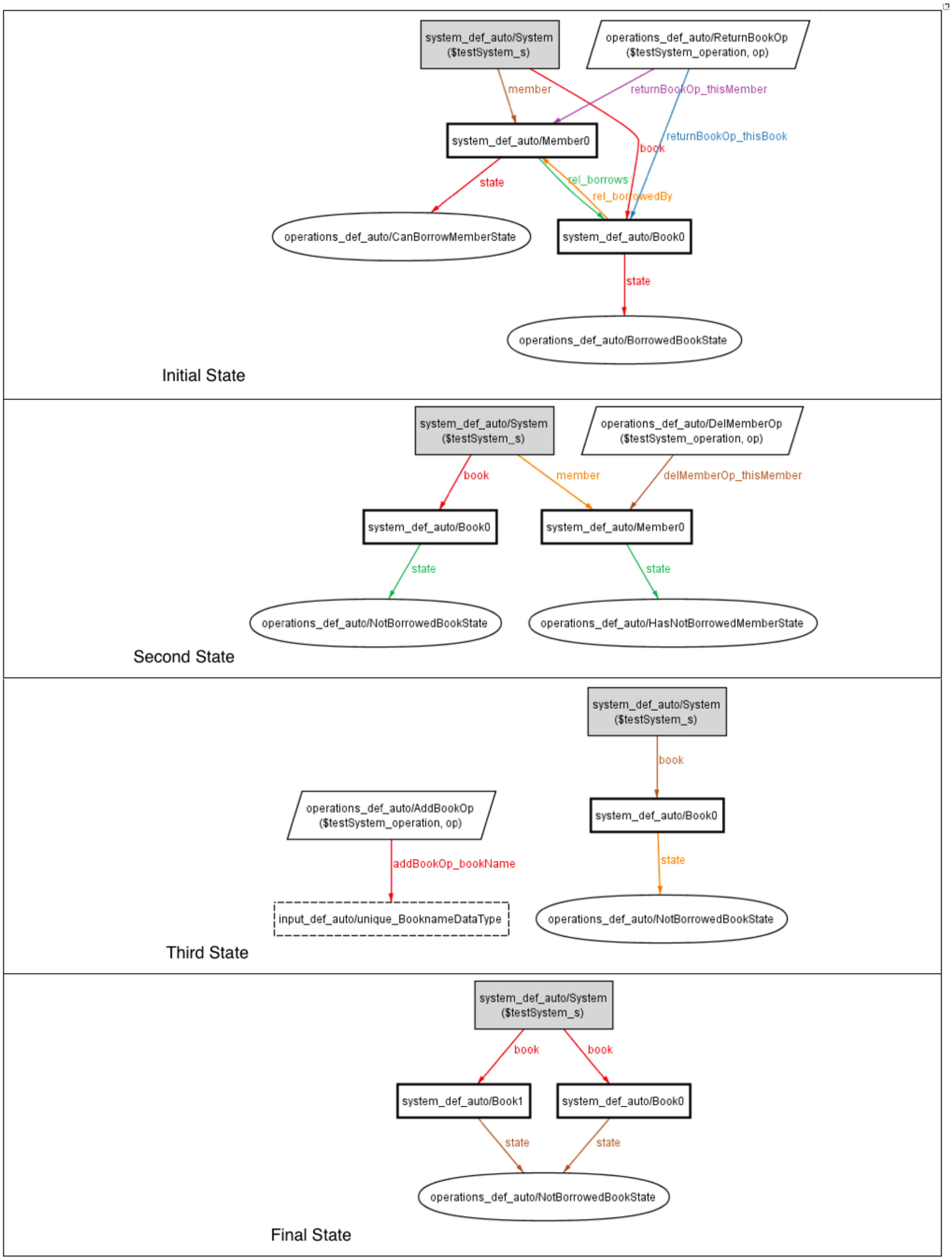

system_def_auto/System ($testSystem_s)
operations_def_auto/ReturnBookOp ($testSystem_operation, op)
member
returnBookOp_thisMember
system_def_auto/Member0
returnBookOp_thisBook
book
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState
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. 32. An execution path (test case) generated by Alloy.

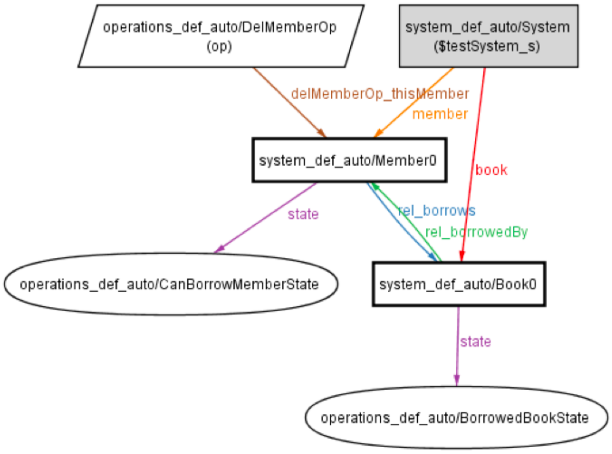

operations_def_auto/DelMemberOp (op)
system_def_auto/System ($testSystem_s)
delMemberOp_thisMember
member
book
system_def_auto/Member0
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState

Fig. 33. Exception state - remove member.

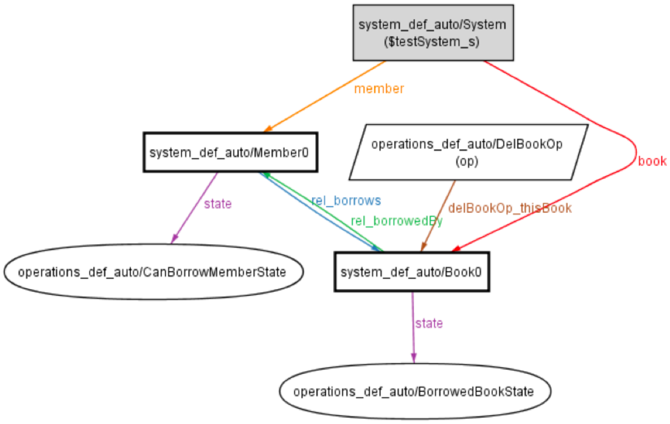

system_def_auto/System ($testSystem_s)
member
system_def_auto/Member0
operations_def_auto/DelBookOp (op)
book
state
rel_borrows
rel_borrowedBy
delBookOp_thisBook
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState

Fig. 34. Exception state - remove book.

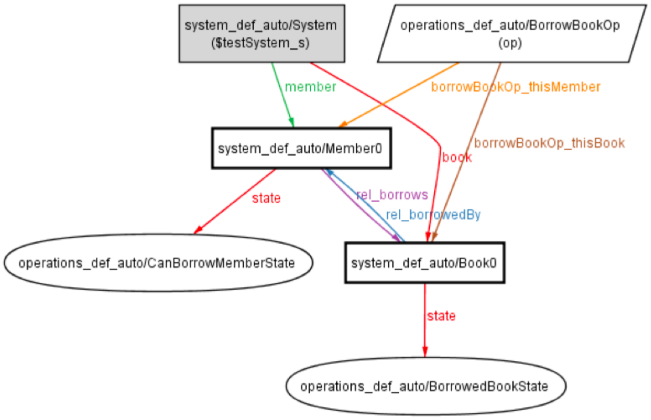

system_def_auto/System ($testSystem_s)
operations_def_auto/BorrowBookOp (op)
member
borrowBookOp_thisMember
system_def_auto/Member0
book
borrowBookOp_thisBook
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
system_def_auto/Book0
state
operations_def_auto/BorrowedBookState

Fig. 35. Exception state - borrow book.

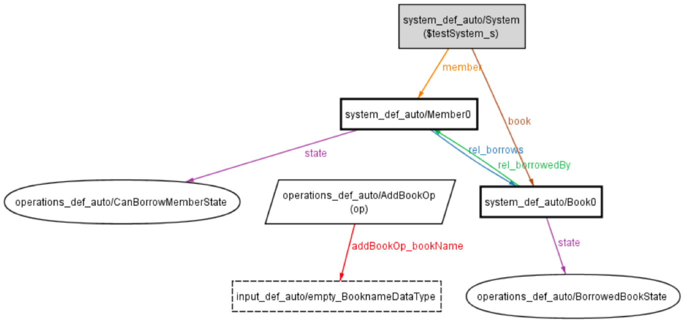

system_def_auto/System ($testSystem_s)
member
system_def_auto/Member0
book
state
rel_borrows
rel_borrowedBy
operations_def_auto/CanBorrowMemberState
operations_def_auto/AddBookOp (op)
system_def_auto/Book0
addBookOp_bookName
state
input_def_auto/empty_BooknameDataType
operations_def_auto/BorrowedBookState

Fig. 36. Exception state - add book: empty name.

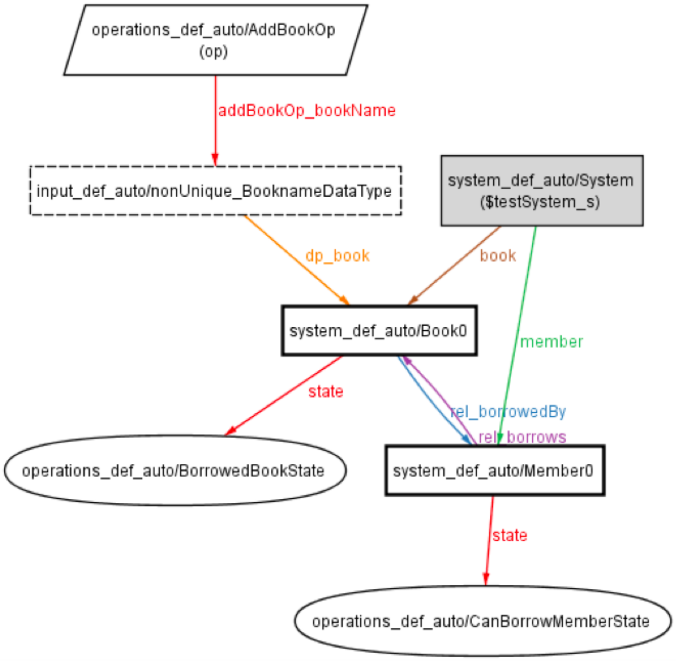

operations_def_auto/AddBookOp (op)
addBookOp_bookName
input_def_auto/nonUnique_BooknameDataType
system_def_auto/System ($testSystem_s)
dp_book
book
system_def_auto/Book0
member
state
rel_borrowedBy
rel_borrows
operations_def_auto/BorrowedBookState
system_def_auto/Member0
state
operations_def_auto/CanBorrowMemberState

Fig. 37. Exception state - add book: redundant name.