University of Amsterdam

Master Thesis Project

Forgery: Synthesizing Database Transactions

Guy - Sean Rombaut

Supervised by: Dr. Tijs van der Storm Jouke Stoel

Organizations: Internationale Nederlanden Groep (ING)

Centrum Wiskunde & Informatica (CWI)

February 5, 2016

Contents

1

Chapter 1

Abstract

The popularity of modeling languages is increasing. This is mostly due to the fact that using modeling languages is an efficient way to end up with a high quality system [?, p. 6]. By providing immediate feedback to users they allow early detection of design errors [?]. Nonetheless these models are far from being a real implemented product. Manual implementation is required which makes it more prone to mistakes. Our motivation for this research was to find a solution in this matter.

Considering the human factor, it is not possible to develop fault-free software in practice[?]. Software issues can be solved but when the data itself is effected this becomes more difficult. Data errors can harm the reputation of an orginazation, diminish financial gains and create uncertainty in an organization. Therefore, we decided to focus on the generation of database systems rather than whole programs.

Forgery is a tool for generating database schemes and synthesize transactions based on a predefined model. Forgery uses Alloy as a specification language for describing models and validating them. An Alloy model is a collection of constraints and relations that describes a set of structures. Using pre- and postconditions it defines the operations that are allowed in the system. Forgery converts them into database tables, procedures and structural constraints.

2

Chapter 2

Preface

This research was done for the Dutch bank ING. The original project aimed to find a solution regarding the communication issues between technical and non-technical teams inside the organization. For example, specifications ambiguity and misunderstanding. The project was initially called Fors and later on was renamed Rebel. It was basically a domain specific language (DSL) that parses itself into Alloy. With Alloy it is possible to validate the model.

Forgery actually aims to find a solution regarding faults in the process of the realization of the model. Forgery can use the output of Fors to support the realization. Together with Fors we may achieve two things: better specifications and better realization.

In order to better understand this paper, it is recommended to read some manuals of Alloy.

3

Chapter 3

Background

In this chapter we will discuss: the background of developing Forgery, the previous work that has been done in this field and the motivation for doing it. We will also present the research question along with a description of the remaining chapters of this thesis.

3.1 Motivation

The success of implementing software projects is directly affected by the quality of its specifications [?, p. 12]. The specifications are typically defined based on two conceptual views: business and technical and are usually defined by different teams or people with various backgrounds. The gap between those two different perspectives may lead to costly misunderstandings [?, p. 1]. Changing specifications after implementation of software often takes much more time and is more expensive [?].

Today several tools exist for modeling and verifying software specifications. Examples are Alloy and Z-notation. These tools allow software engineers to create prototypes of their ideas and identify errors, before realization.

However, sometimes such modeling tools seems to be too complicated. Even though the mathematical notations of these tools are unambiguous, the use of set theories, logic and algebra requires special expertise [?, p. 10]. In addition, these tools are useful especially for prototyping general models and less effective when it comes to specific domains. We focus on such a specific domain (financial systems).

Because of the above-mentioned, we aimed at creating a new tool that would be better suited for prototyping financial systems and could be used by both the business and the development teams. We have developed our own Domain Specific Language (DSL) and called it “Fors” which derived from: Separating Configuration From Formal Specification [?]. The concept of a DSL is very simple: Instead of aiming to solve any kind of computing problem, DSLs aim to solve specific class of problems. [?]ln our case the DSL aims to solve problems of financial systems.

Fors expresses the operations of a system in a language whose vocabulary, syntax and semantics are formally defined in an easy and natural way. This way, Fors is comprehensible for both business and development teams. In addition, Fors is able to check the correctness of a software model. Fors parse formal specifications into Alloy syntax: algebraic logic formulas based on the notion of relations (We will elaborate on this more later in this thesis). Using an Alloy based engine we are able to solve such formulas and find ambiguities in the model.

Fors minimizes the gap between the business and the technical views by creating a common language and the ability to identify contradictions or faults in a specific model. However, there is still a main issue that remains: Programmers will have to implement the real product by hand (according to the specifications). Hence, it is not guaranteed that the final results would be exactly the same as defined in the specifications. When considering human factor also this system is prone to error.

Therefore, our motivation was to find out whether we would be able to create a tool for automatic system generation. We decided to scope our research on the data-side as we found it highly interesting. As mentioned previously, data errors can harm the reputation of an orginazation, diminish financial gains and create uncertainty in an organization. Synthesizing the data may avoid such events.

Data is usually stored in a record-keeping system called a database. A database therefore is a repository for a collection of data files on which users may perform a variety of operations (e.g. adding, modifying or reading files [?, p. 11]).

The scope of a database is often described as having the following three aspects:

* Data Structure - the structure is a representation of the arrangement, relationships, and contents of  data [?]. The structure is described diagrammatically by the data schema.
* Data Manipulation - the available operations that can be applied on the data. Mainly CRUD  (Create, Read, Update and Delete) operations.
* Data Integrity - refers to the accuracy and consistency (validity) of data over its lifecycle.

Using a database has numerous benefits that lay mostly in the fact that data control is centalized. First, redundancy can be reduced. In contrast to private files, by using a relational database it is possible to merge related or overlapping information. Second, by linking multiple rows and use of transactions we can avoid inconsistency (corollary of the previous point). Also, security (per- missions) and standards (e.g. representation of the data) can be enforced. And last, sharing data between multiple workstations is easy [?, p. 16].  Since Fors uses Alloy; and Alloy is based on relations, we researched if it is possible to make a link between Alloy and relational databases; and then we may automatically generate a matching database.  A relational database consists of three main principles:

* Tables are the logical structure (but physically they can be stored in multiple way - binary trees, hashing etc).
* The information principle - the entire content of the database is represented in one and only one way.
* The operators available to the user derive from old state to a new one.  The reason that such a system called "relational" is because basically a relation is a mathematical term

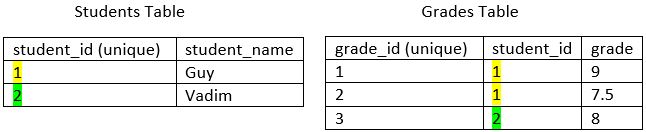
for a table and it doesn’t only refer to entities and relationships [?, p. 26]. A relational system is based on the relational model of data.

Example - Simple student grades system:

Students table is storing students data (name), and grades table is used for storing the student grades. In order to make a link between a student and a grade we use a unique id. The user can use multiple operations to manipulate the data. Each operation (e.g. delete) will generate a new table; that means changing from "old" state to a "new" one. For example, by deleting a grade row, the relation will be replaced by a new one (excluding the deleted row). Similarly, inserting or updating operations.

Now we return back to relations. We have already mentioned the term "relation" multiple times. Before we can fully define a relation we have to introduce another term called tuple due to the fact that tuples are a necessary stepping-stone to understand what a relation is. Given a collection of types (e.g. number, date, address etc) Ti(i = 1,2,3...,n), a tuple value on those types - t, say - is a set of ordered triples of the form < Ai,Ti,vi >, where Ai is an attribute name, Ti is a type name and vi is a value of type Ti. The value n is the arity of t (unary, binary etc) [?, p. 142].

t(n) =<< A1,T1,v1 >,< A2,T2,v2 > ...,< An,Tn,n >> Few notes regarding tuples:



CHAPTER 3. BACKGROUND 6

• The heading of t is the set of attributes. • The type of t is determined by the types of its heading. • Each tuple contains exactly one value for each of its attributes. • Every subset of a tuple is a tuple. • There are no duplicate tuples.

Example - Tuple: Students = < Student, Guy >, < Student, V adim > And now, here is the precise definition of a relation [?, p. 146]. A relation r consists of heading and a body, where:

• The heading of r is a tuple heading as defined before.

• The body of r is a set of tuples with identical headings.

Example - Relation: Grades = {<< Student, Guy >, < Grade, 9 >>, << Student, Guy >, < Grade, 7.5 >>, << Student, V adim >, < Grade, 8 >>}

And lastly, the relations theory provides a set of operations for relations. For example, subset of ⊂, superset of ⊃, equals = and others. Those will also be discussed in the next chapters.

As said, Alloy is also based on the notion of relations and it use them as its main structure. Therefore, the data model of Alloy’s can be translated directly into persistent database schema. Obviously, Alloy is much more than a data structure, it provides functions and more. We will discuss more about Alloy in the next chapter.

We decided to use SQL (Structured Query Language) for the automation of database generating. SQL is a declarative language designed for constructing relational databases and managing the data that is held in it.

3.2 Problem Analysis

Generating a database system automatically using only Alloy specifications seems to be hard. We will cover the reasons in this section.

3.2.1 Data Structure

Although Alloy relations and database tables are both based on the theory of relations, their structural representation is different.

* Database tables are two-dimensional, while relations can have multiple dimensions.
* Database tables are ordered (top-bottom and left-right), while relations not.
* Database tables can contain empty data (null) and duplicates, while relations/tuples not.
* Each relation usually involve a type name. When it comes to database tables, types are typically omitted.  In addition, database systems make use of Keys which are a vital part of the table structure. For example, Primary, Foreign and Unique Keys. Alloy deals with it partially or not at all.  In order to be able to generate tables that can represent the corresponding Alloy relations, we need to agree on certain rules of interpreting these relations (e.g. row or columns orderings are irrelevant) [?, p. 151].

CHAPTER 3. BACKGROUND 7 3.2.2 Data Operations

Alloy uses predicates as operations system. It allows users to create customized actions for modifying the data in the system using preconditions, postconditons and algebraic formulas (We will discuss about it in the next chapter). In database systems the core operations involve insert, update or delete. We need to bridge between algebraic notation and a SQL query notation.

Moreover, as mentioned before, in database systems the operators available to the user derive from old state to a new one. In Alloy this is not necessarily the case. We need to enforce the specifications to follow that way.

And lastly, we have to assure that the new state is valid (according to the specified conditions of the formula). For example, Alloy allows setting multiple operations inside single predicate. In database systems, each operation is discrete and it may occur that one operation succeeded and the following one didn’t. In this case, the new state is invalid. To solve this for example, we can use transactions.

We will discuss about the solutions and the related work in the next few chapters.

3.3 Research Question

This paper researches the implementation problem and tries to answer the following question:

How to bridge between Alloy-based specifications and realization?

To answer this question, the following sub-questions have been formulated: • How can we meet Alloy specifications? • What are the limitations on Alloy specifications? • How to validate that Forgery works?

Chapter 4

Alloy

Alloy is a declarative specification language for describing models with structural constraints and behavior used for modeling software systems. In addition, it includes a tool called Alloy Analyzer for visualizing models, exploring and checking the properties of them.

In this chapter we concentrate on the language part of Alloy, and we will provide few examples as demonstration. We will not cover the whole language but mostly what is required in order to understand this mass.

4.1 Data Structure

Alloy data model is based on atoms, signatures and fields. Atom is the most basic element in Alloy which belong to a specific type. signature is a set of atoms, which also define the data type. A field describes relation between different types of signatures. It is a set of tuples which consists different types of data. Such a relation can be unary or binary (using →).

The following example describes a price list (tariff) of products in shop.

sig Price {} s i g Product {} sig Shop {

tariff : Price −> Product

}

The signatures are price, product and shop. The tariff describes the relation between each product and price within the shop. Running the Alloy Analyzer will generate all possible examples that follows the model. For example we may have two shops, with two different products that has the same price. In that case, we have two atoms of type shop, two atoms of type product, and single atom of type price.

4.1.1 Cardinalities

Cardinality refers to the amount of elements in a set. Alloy allows us to set cardinality constraints on the data model. Alloy supports lone (size is at most 1), one (size is only 1), some (size is at least 1), no (size is 0). For example, let’s assume that we want only one shop in our model:

sig Price {} s i g Product {} one sig Shop {

tariff : Price −> Product Alloy Analyzer will now only generate examples with exactly one shop.

}

8

CHAPTER 4. ALLOY 9 4.2 Data Operations

Alloy allows to define predicates as an operations that act on that system. A predicate may convert a certain state of the data system into a new one, based on the rules it defines. The predicates accept signa- tures as an input, and the operations themselves defined using algebraic formulas. We may use semantics such as + (union), & (intersection), in (subset), → (tupling), and . (join).

The following example describes a predicate that allows adding a new tariff (of product pro with a price of pri to the shop). The shop s describes the old state while s′ describes the new state of the system.

sig Price {} s i g Product {} one sig Shop {

tariff : Price −> Product pred AddTariff(s, s’ : Shop, pro: Product, pri: Price) {

}

}

4.3 Data Invariants

Alloy allows to create system invariants using facts. Those properties are meant to hold of all models constructed by Alloy. Any configuration that is an instance of the specification has to satisfy all the facts. In the previous example we could have products that doesn’t belong to any shop (Product0).

Using facts, we can assure, for example, that each product must belong to a shop.

sig Price {} s i g Product {} one sig Shop {

tariff : Price −> Product pred AddTariff(s, s’ : Shop, pro: Product, pri: Price) {

s’.tariff = s.tariff + (pri −> pro) f a c t ProductMustHaveShop {

all pro: Product | pro in Shop.tariff[Price]

}

4.4 Assertions

Assertions are constraints that were intended to follow from facts of the model. Assertions are used for checking that the desirable invariants exist using Alloy Analyzer. Alloy Analyzer tries to find counter examples that does not follow those constraints.

Let’s assume that we want that each shop will have maximum one tariff, and such invariant was forgotten.

}

}

s’.tariff = s.tariff + (pri −> pro)

The shop had the product ’0’ with price ’0’ and after we added a product ’1’ with a price of ’1’.

CHAPTER 4. ALLOY 10

sig Price {} s i g Product {} one sig Shop {

tariff : Price −> Product pred AddTariff(s, s’ : Shop, pro: Product, pri: Price) {

s’.tariff = s.tariff + (pri −> pro) f a c t ProductMustHaveShop {

all pro: Product | pro in Shop.tariff[Price]

}

assert LoneTariff { all s: Shop | lone s.tariff

}

By running a check for the assertion LoneTariff, Alloy Analyzer will alert that it found a counter example:

By clicking on the counterexample, Alloy Analyzer will present the all counter-models. The follow- ing counterexample shows that there are more than one tariff, which is in conflict with the mentioned constrained.

This way, we able to check our model, and make it more robust by minimizing mistakes.

}

}

Chapter 5

Forgery Overview

In this chapter we will briefly give an overview on the solution that we offer. It might give a general idea for those who are not interested in investing time reading it into details.

5.1 Key Ingredients

As introduced in the background Alloy specification consists of four main parts: Signatures (and their Cardinalities), Predicates, Facts and Assertions. Together they hint about how the model should be implemented. Using few examples we will demonstrate the conversion from Alloy specifications to imple- mented database.

The following example specifies a homework submission and grading system.

sig Submission { } sig Grade { } sig Student { } sig Course {

roster : Student , work: roster −> Submission , gradebook : work −> Grade

} For those specifications Alloy will generate multiple models. Arbitrarily we chose the following one:

The system contains two courses, two student which are enrolled to each of those courses. Also, Stu- dent0 which is enrolled to Course1 submitted his work and got a grade for it.

First We need to generate a database schema for storing the data. As can be seen, Alloy Signatures can be translated directly into persistent database schemas. In Forgery, for each Signature we create a table, and for each field we create a junction tables that points to the relevant signature tables. Each Signature table stores its atoms. The relations set by ids which are the primary keys.

For example, the student table is created by the following SQL code:

CREATE TABLE ‘ student ‘ ( ‘ i d ‘ INT ( 6 ) UNSIGNED NOT NULL AUTO\_INCREMENT PRIMARY KEY, ‘value‘ VARCHAR(100) NULL

) ENGINE=InnoDB DEFAULT CHARSET=UTF8; A junction table, for example for the relation roster is created by the following SQL code:

CREATE TABLE ‘ roster ‘( ‘ i d ‘ INT ( 6 ) UNSIGNED NOT NULL AUTO\_INCREMENT PRIMARY KEY, ‘course\_id ‘ INT(6) UNSIGNED NOT NULL, ‘student\_id ‘ INT(6) UNSIGNED NOT NULL,

11

CHAPTER 5. FORGERY OVERVIEW 12

UNIQUE INDEX ui(‘course\_id‘,‘student\_id‘), FOREIGN KEY (‘course\_id ‘) REFERENCES ‘course ‘( ‘id ‘) , FOREIGN KEY ( ‘ student\_id ‘ ) REFERENCES ‘ student ‘ ( ‘ id ‘ )

) ENGINE=InnoDB DEFAULT CHARSET=UTF8;

We use foreign keys to enforce integrity. These constraints guarantee that, for example, a row in the table roster with a field studentid referencing the student table will never have an studentid value that does not exist in the students table. In addition, since Alloy refers to sets, according to the set theory, every element of a set must be unique; no two members may be identical. Hence, we create a a SQL unique index.

Now when we have a database which we can store data to, we need a way to insert, update or delete data. For example, creating Atoms and implementing Alloy’s predicates. For that purpose we decided to use SQL stored procedures. Stored procedures are similar to procedures in other programming languages in that they can accept inputs, return output and support programming statements for performing oper- ations on the database.

In this chapter we will not go deeply into the algorithm, as that will be described in the next chapter. Alloy is based on the notation of algebraic mathematics. Operators over sets and relations have their usual semantics: + (union), & (intersection), in (subset), → (tupling), and . (join). SQL supports simple operands such as +, -, x, ÷ and set operations such as union, intersection, difference etc. That gives us enough flexibility to transform the Alloy predicates into SQL procedures.[?]

We use the tag symbol (’) for describing a state transition. For example, student s is the pre-condition, and s’ is the post-condition.

The following predicate enroll a student to a course. PUT HERE PREDICATE CODE

In database terms, we create a procedure perform an Insert query as the following. PUT HERE SQL CODE

Similarly to Predicates, Facts are also written in algebraic form.

The following fact checks that all grades at at least 5.5. PUT HERE PREDICATE CODE

Also in this case, we create a procedure that verifies this condition. PUT HERE SQL CODE

Those procedures are automatically called when there are modifications in the database. In case of a failure to match the condition, the changed that made will not be committed and the database will be reverted to its valid state (all actions are transactional).

5.2 Architecture Overview

Forgery consists of 7 main modules:

1. Syntax Validator: Forgery checks the input using Alloy API. A failure throws an exception, and  stops the execution.
2. Parser and Mappers: Forgery parses the Alloy syntax and generates ASTs (Abstract Syntax Trees). Those ASTs are flattened to a simpler map structures for later processing.
3. Tables Generator: Creating SQL tables including keys, relations and uniqueness constraints.
4. Procedures Generator for Facts and Cardinalities: Creating SQL procedures for verifying the specified invariants and cardinalities.
5. Procedures Generator for Predicates: Creating SQL procedures for systemic operations according to the specifications.

CHAPTER 5. FORGERY OVERVIEW 13 6. Traces Generator: Reverse engineering for Alloy traces, used for the Validator.

7. Validator: replicating the operations that made in Alloy for finding models and comparing data to validate behavioural similarities.

CHAPTER 5. FORGERY OVERVIEW 14

Figure 5.1: Forgery architecture

We implemented Forgery using Rascal. We chose Rascal due to its powerful DSL and AST (Abstract Syntax Trees) tools. We used a standard SQL language. Therefore, any arbitrary database system can be used. We used MySQL as it has large community. In this overview we will introduce the main points of how Forgery transform Alloy into SQL, and vital explanation for SQL.

5.3 Alchemy Comparison

Although related papers will be presented in a different chapter, there is a unique paper called Alchemy [?] that we have studied and due to overlapping research we will present it here. Similarly to Forgery, Alchemy compiles Alloy specifications into database implementation. We will discuss about the main differences here.

First, Alchemy generates a synthesizing API as a filtering layer that communicates with the database. In other words, the data validity can be guaranteed only when using this API. In contrast, Forgery gener- ates a SQL system, the constraints are in the tables and the database level, hence it is more robust - the validity occurs on the data storing level. Also, SQL language is usually more familiar for technical people. And therefore, it may give a communication advantage.

Secondly, as introduced before, Alloy supports assertions for checking the model. Using the powerful Analyzer we able to detect mistakes in our specifications. Forgery supports assertions and even use their traces for evaluation. Alchemy introduced a shortcut to create predicates. However, those predicates specifications might be in contradiction to invariants. They added an auto-repair functionality that fix the data in case it is in conflict. However, the since the specifications are invalid by the nature of Alloy. The assertions and the analyzer cannot be used. Alchemy does not have evaluation method.

Thirdly, Alchemy does not generate command options to insert or delete atoms. By default, Forgery creates such procedures (With an option to create them manually).

CHAPTER 5. FORGERY OVERVIEW 15 5.4 Scheme Generation

In this section we will discuss about how forgery generates the database scheme; including tables, fields etc. We will use the similar example as described in the overview with additional functionality and we will dive deeper into details.

The example describes a homework submission and grading system. Student’s work may be submitted in pairs or individually. The gradebook stores the grade for each student on each submission. The system has some constraints and actions like enrolling students and so on, which we will not be discussed in this overview.

sig Submission {} sig Grade {} sig Student {} sig Course {

roster : set Student , work: roster −> lone Submission , gradebook : work −> lone Grade

}

Alloy uses signatures (e.g. Submission) to describe a data model [?, p. 30]. Every signature defines the data type, and consists set of atoms drawn from that type. Atoms are elements that are created based on the system user’s inputs. Forgery allows atoms to be created only within those signatures.

In addition, a signature can define fields (e.g. in Course). A field describes a relation between different types of Signatures. Basically, it is a set of tuples which consists different types of data. Such a relation can be unary or binary. E.g. roster and work respectively.

CHAPTER 5. FORGERY OVERVIEW 16 5.4.1 Atoms Tables

The signatures Student, Submission, Grade and Course are sets of atoms: Student = {Guy,Tijs,Jouke..} Submission = {homework, project..} Grade = {5.5, 8..}

Course = {Construction..}

For each signature, atoms table is created. Every table has two fields: id and value. The id field is a primary key for identifying the atom, and value is the input data. Chosen types: Int for id as it is always an integer, and Varchar for value so it can contain any type of input (strings, numbers etc).

Figure 5.2: Generated tables for Signatures

CHAPTER 5. FORGERY OVERVIEW 17 5.4.2 Fields Relations

The signature Course defines the following relations: roster (enrolled students), work and gradebook. roster = {< Construction, Guy >, < Construction, T ijs > ..} work = {< Construction, Guy, hwk1 > ..}

gradebook = {< Construction, Guy, homework, 8 > ..}

For every relation, a junction table is created. It describes the relationship between the different multiple relations and the atoms. Every table contains the ids of the atoms that the relation describes. The table name is based on the relation name (right side before the colon sign :). Relation can be unary or binary. Binary relation is expressed by the tuple sign →.

Moreover, Forgery adds SQL Foreign Keys that links between the tables. They are naturally extracted from the Alloy semantics. It guarantee that every junction table will contain valid data that points to existing atoms.

When a relation points to a another relation, Forgery extracts the atomic type. Which means, in other words, Forgery flattens all the relations so the tables will contain pointers to atomic tables only, and this way, the relationships between the tables would be simpler. It adds data redundancy but it makes the algorithm much more simpler.

Figure 5.3: Generated tables for Relations

CHAPTER 5. FORGERY OVERVIEW 18 5.4.3 Cardinalities

Alloy syntax supports multiple quantifiers to describe constraints on the data model: lone (at most one atom), some (at least one atom) and one (single atom). Forgery uses two techniques to implement those constraints:

1. Adding Unique Indexes for tables: the shared condition of lone, one and set is that the table must not contain more than one similar element. Adding Unique Index guarantee this condition (although only partially/weaker condition for one).

2. Creating Stored Procedures: one and some cardinalities cannot be fully implemented by applying structural constraints into the SQL table itself. Therefore, we create a SQL procedure instead for each cardinality. The procedure contains a query that counts the rows and group it based on the cardinality criteria and then compare it to the specified cardinality. Those procedures are automatically called when the data is changed. If any violation occurs, the data will be reverted and error will be shown (transactional action). The cardinality procedures are stored in the database and can be identified with the prefix c\_. More about procedures will be explained in the next chapter.

CHAPTER 5. FORGERY OVERVIEW 19 5.4.4 The Algorithm

Let S be a list of signatures, s is a single signature, R is its contained relations, r is a single relation. Algorithm 1 Database scheme mapper

1: 2: 3: 4: 5: 6: 7: 8: 9:

10: 11: 12: 13: 14: 15: 16:

function Create Scheme(S) tables = ()

for each signature s in S do add (s.name : id, value) to tables for each relation r in s.R do

fields = {} add s.name.‘\_id‘ to fields add Atomic fields (r.op1, tables, S) to fields if r.type is Binary then

add Atomic fields (r.op2, tables, S) to fields end if

add (r.name : f ields) to tables end for

end for

return tables end function

◃ map (table name: list of fields)

◃ set of table fields with cardinalities



CHAPTER 5. FORGERY OVERVIEW 20 Algorithm 2 Returns the most basic fields (Relations to their atoms)

* 1:  functionAtomicFields(r,tables,S)
* 2:  if r.name in S.names then
* 3:  return r.name.‘\_id‘
* 4:  end if
* 5:  return tables[r.name]
* 6:  end function

Algorithm 3 Returns the query

* 1:  function Generate Query(tables)
* 2:  query = ””
* 3:  for table in tables do
* 4:  add SQL Statement ("Create Table", table.name) to query
* 5:  for field in tables[table] do
* 6:  add SQL Statement ("Create Column", field.name) to query
* 7:  add SQL Statement ("Foreign Key", field.name) to query
* 8:  if field.cardinality is lone or one or set then
* 9:  add SQL Statement ("Create Unique Index", fields) to query
* 10:  end if
* 11:  if field.cardinality is some or one then
* 12:  add SQL Statement ("Create Cardinality Procedure", fields) to query
* 13:  end if
* 14:  end for
* 15:  add SQL Statement ("Create Column", ‘id‘) to query
* 16:  add SQL Statement ("Create Primary Key", ‘id‘) to query
* 17:  end for
* 18:  return query
* 19:  end function

CHAPTER 5. FORGERY OVERVIEW 21 5.5 Predicates Generation

This overview will discuss about the logic behind interpretation of Alloy predicates and creation of atoms. The same example from the previous chapter will be used with few additional statements.

The new statements introduces new features such as adding or deleting students, as they enroll in or drop the course, and assigning grades for each of their submitted work in pairs.

sig Submission {} sig Grade {} sig Student {} sig Course {

roster : set Student , work: roster −> lone Submission , gradebook : work −> lone Grade

} pred Enroll (c , c ’ : Course , \_sNew : Student) {

c’.roster = c.roster + \_sNew and no c’.work [\_sNew]

} pred Drop (c, c’ : Course, s: Student) {

} pred

s not in c’.roster SubmitForPair ( c , c ’ : Course , s1 :

Student ,

s2 :

Student ,

\_bNew :

Submission ) { // pre−condition s1 in c.roster and s2 in c.roster and // update c’.work = c.work + (s1 −> \_bNew) + (s2 −> \_bNew)

} pred g : Grade) {

}

AssignGrade (c , c ’ : Course , s : Student , b : Submission , c’.gradebook = c.gradebook + (s −> b −> g)

CHAPTER 5. FORGERY OVERVIEW 22 5.5.1 Using Stored Procedures

Stored Procedure is a SQL feature that encapsulates a query for re-usability purposes. It is used as a layer that communicates with the database internally. Stored Procedures allow faster execution time and they may be useful as a safer synthesizing mechanism (E.g. privileges) [?].

Basically, Stored Procedures are similar to other programming languages in that they can accept input parameters and return multiple values. Also, they may contain programming statements for performing operations in the database and indicate status of failure or success.

Forgery uses stored procedures for multiple purposes. One of them was already introduced in the previous overview in reference to cardinalities. Some others will be covered as well in this chapter.

Alloy Predicates

Alloy uses predicates (e.g. Enroll) to capture the actions that are supported in the system. Each predicate describes the required state that the system should be in when applying it. Predicates has a header and a body.

Predicates Header: Predicates accept inputs from the user that are used for the states transformation. Each input contains a variable and a mapping to his belonged table. Similarly to Alchemy [?], Forgery uses the prime symbol ′ as a variable suffix to distinguish between pre- and post-states of the operation (e.g. c and c′).

The inputs may be one of the two different types: an integer or a string. Inputs accept integers as default and each of them represents an Atom id. However, when it comes to a new Atom the id does not exist yet. Therefore, Forgery uses the underline symbol \_ in the variable prefix to refer to a new atom. In this case, the input type is a string, which is the data that the Atom carries. It may be a name of a course or a student and it may be just empty, depends on our model.

For each request for new atom, an Insert query will be added to the procedure, and the created id will be placed in the variable value. E.g. in our example \_sNew : Student the generated query will be:

INSERTINTO student(‘value‘) VALUES (\_sNew); SET \_sNew = LAST\_INSERT\_ID();

Predicates Body: The predicates body may contain a formula in which the defined variables in the header are used. The formula semantics of Alloy is based on the class of relational algebras. A predicate may define multiple formulas. Formulas are joined together using the and word. Forgery handles them as a list of commands which performed serially one after one.

Forgery assumes that the formulas are correct by testing them in Alloy Checker1. In addition, Forgery first checks whether the command refers to a pre- or post-condition and generates the query accordingly. For example, the Alloy membership test operator in will be converted into one of the following:

1. Precondition is converted to "rollback if not exists" query (and the opposite for not in). Example: s1 in c.roster

IF NOT EXISTS (SELECT ‘id ‘ FROM ‘roster ‘ WHERE ‘student\_id‘=s1 AND ‘course\_id‘=c)

THEN SELECT ‘An error has occurred , operation rollbacked & the stored procedure was terminated ‘ ; ROLLBACK;

END IF; 2. Postcondition is converted to "insert if not exists" query (and delete for not in): s1 in c′.roster

1Not implemented yet.

CHAPTER 5. FORGERY OVERVIEW 23

IF NOT EXISTS (SELECT ‘id ‘ FROM ‘roster ‘ WHERE ‘student\_id‘=s1 AND ‘course\_id‘=c)

THEN INSERTINTO ‘roster‘ (‘student\_id‘, ‘course\_id‘) VALUES(s1, c);

END IF; More operators:

1. Union - the operator ′+′ is used to compute union of sets. Both sets must be of the same relation type and it is verified by Alloy Checker. Forgery makes the union using two different methods:
   * (a)  When the formula can be used equivalently with in operator (insert), similar method to in will be used (without existence testing). For example: c′.roster = c.roster+s1 is equivalent to s1 in c′.roster (As all operations are transactional). Another example: c′.roster = c.roster + s1 + s2 is equivalent to s1 in c′.roster and s2 in c′.roster.
   * (b)  In all other cases, the data in the table will be first flushed and then new data will be inserted2.
2. Alchemy uses framing conditions (e.g. c′.gradebook = c.gradebook). However, since Forgery execute  all queries transactionally they are not needed (as mentioned before).
3. Difference - the operator ′−′ works similarly to union, but with delete statements instead3.
4. Join - operation is represented by square braces [] (although in other languages it usually means array access). In our example the formula no c′.work[\_sNew] meant to verify that there is no work for this course that was submitted by the enrolling student. The generated query is:  IF EXISTS (SELECT id FROM ‘work‘ WHERE ‘student\_id‘=\_sNew AND ‘course\_id‘=c)  THEN  SELECT ‘An error has occurred , operation rollbacked and the stored procedure was terminated ‘ ;  ROLLBACK; END IF;

4. Tupling - represented by → symbol. Example: c′.work = c.work + (s1 → \_b) + (s2 → \_b). The generated query is:

INSERT INTO ‘work‘ (course\_id , student\_id , submission\_id) VALUES (c, s1, \_bNew);

INSERT INTO ‘work‘ (course\_id , student\_id , submission\_id) VALUES (c, s2, \_bNew);

5. Intersection - represented by & symbol. Supported only in facts (will not be discussed here). Atoms Creation:

Forgery supports two ways for creating new Atoms. The first way, which was introduced already is by using underline \_ variable prefix. The other way is by using the Forgery "create" procedures. Forgery automatically generates atom creation procedure for each signature. For example:

DELIMITER // CREATE PROCEDURE ‘create\_submission ‘(IN atomVal VARCHAR(100))

BEGIN DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN

2Not implemented yet. 3Not implemented yet.

CHAPTER 5. FORGERY OVERVIEW 24 ROLLBACK;

END;

START TRANSACTION; INSERT INTO ‘submission ‘ (‘value‘) VALUES (atomVal); −− facts and cardinalities are included here.

COMMIT; END //

As mentioned before, Forgery verifies all the invariants (cardinalities and facts) in each procedure. This rule applies here. This means that if there is any violation, the data will be reverted.

CHAPTER 5. FORGERY OVERVIEW 25

Figure 5.4: Generated Relational Database

CHAPTER 5. FORGERY OVERVIEW 26 5.6 Validation

5.6.1 Reverse Engineering

While Alloy search for models, it generates traces and saves them as a temporary XML file. Those traces are the operations that were performed in the system and they create the possible models or counter examples.

The idea was to use those traces for duplicating the operations in Forgery’s database and then com- paring the results to those in Alloy for tests purposes. In other words, applying reverse engineering on the results of Alloy. However, the traces miss some important information.

By default, the traces are not ordered and this has to be configured manually. Otherwise, they are useless because we cannot know the data flow. To order them, the ordering library has to be included and initialized using init predicate (ignored in Forgery). And also, the logic behind the steps progress from one to another has to be defined using traces fact (also ignored in Forgery).

Besides, the operation name that leaded to a new state is not saved so we able to know only the data of the new state (Atoms and the Relations) but not who caused i. Therefore, to deal with it we made a small trick. We added an arbitrary atom called operationid and his value is connected to each operation and identifies it uniquely. For example:

open util/ordering[Course] as CourseOrder sig Student {} sig Course {

roster : set Student , operation\_id : Int

}

pred Enroll (c, c’ : Course, \_st : Student) { c’.roster = c.roster + \_st

and c’.operation\_id = 1

}

pred init(c: Course) { no c.roster

and c . operation\_id = 0

}

fact traces{ init[first]

all c: Course − last | let c’ = next[c] | some st : Student | Enroll [ c , c ’ , st ]

} Alloy’s output:

The data flow:

1. Initializing (0): The precondition for the initializing state is that there are no students enrolled and therefore no students enrolled to Course0.

2. Enrolling (1): The student is enrolled to Course1. 3. Enrolling (1): The student is enrolled to Course2.

CHAPTER 5. FORGERY OVERVIEW 27 5.6.2 Validation Scenarios

In order to validate Forgery’s output we have to consider the following scenarios:

True Positive - False Negative

• True Positive: Correct models in Alloy and in Forgery.

• False Negative: Correct models in Alloy but incorrect in Forgery.

We have to irritate over all the correct Alloy models and test them in Forgery using the traces as described before.

True Negative - False Positive

• True Negative: Incorrect models in Alloy and in Forgery.

• False Positive: Incorrect models in Alloy but correct in Forgery.

Alloy supports policy constraints. One of them which has been already introduced is a fact. Another type but similar is assert. While a fact is used to force something to be true of the model, an assert is a claim that something must already be true due to the rest of the model [?]. Assertions are used mostly to test the predicates. They verify that the specified model is correct and that we used the desirable conditions (e.g. not too weak).

Similarly to finding models, Alloy uses the Assertions to find counterexamples for the model. Traces are also generated in this case, and we able to use them to test Forgery too.

Chapter 6

Related Work

Alloy is increasingly becoming popular declarative modelling language thanks to providing early error de- tection, supported by automated model analysis tools for simulating and debugging models [?]. Therefore, there were few related papers that were written.

6.1 From UML to Alloy and Back Again

This paper presents a study involving UML2Alloy, a tool for transforming UML models in form of UML class diagrams which are augmented with OCL constraints, to Alloy. The conversion allows analysis of UML models via Alloy, to identify consistencies in those UML models.

6.2 Alchemy: Transmuting Base Alloy Specifications into Imple- mentations

We present Alchemy, which compiles Alloy specifications into implementations that execute against per- sistent databases. Alchemy translates a subset of Alloy predicates into imperative update operations, and it converts facts into database integrity constraints that it maintains automatically in the face of these imperative actions.

6.3 Mapping between Alloy specifications and database imple- mentations

An abstract Alloy specification is far from an actual implementation, and manually refining the former into the latter is unfortunately a non-trivial task. This paper identifies a subset of the Alloy language that is equivalent to a relational database schema with the most conventional integrity constraints, namely functional and inclusion dependencies.

6.4 Towards an Operational Semantics for Alloy

In this paper we demonstrate the subtlety of representing state in Alloy speci- fications. We formalize a natural notion of transition semantics for state-based specifications and show examples of specifications in this class for which analy- sis based on relational algebra can induce false confidence in designs.

28

Chapter 7

Conclusions

In this work we presented Forgery as a tool that supports the realization of Alloy based specifications. We able to generate a full-scale database including structural tables, constraints and functions for data updating operations. The output is a pure SQL that doesn’t require any external API or additional de- pendencies. In addition, it is easily expandable by using different SQL features. For example it is possible to manage privileges of users and that way increasing security.

As mentioned before, it is not possible to develop fault-free software in practical scenario considering human nature [?]. That also applies for Forgery. Hence, we developed a validation mechanism with reverse engineering that allows to compare the results to those in Alloy. Moreover, enabling Assertions helps to find issues in the model; A feature that was not possible in Alchemy.

Together with Fors we created a toolchain that supports the main software development processes: modelling and implementation. Those two important processes have direct affect on the software quality. By improving them organizations might save lot of money and other resources.

There are still challenges to be solved in Forgery,for instance, expanding the set of supported Alloy syntax and resolving potential exceptions. However, to the best of our knowledge, considering multiple scenarios, Forgery has achieved his goals.

29

Bibliography

* [1]  Pete Sawyer and Gerald Kotonya, Software Requirements (IEEE, 2001).
* [2]  Walid Taha, Domain-Specific Languages (Rice University, 2008).
* [3]  Chiel Peters, Fors: Separating Configuration From Formal Specification (University of Amsterdam, 2014).
* [4]  Daniel J. Dougherty et al, Alchemy: Transmuting Base Alloy Specifications into Implementations (Worcester, 2008).
* [5]  Daniel Jackson, Software Abstractions: Logic, Language and Analysis (MIT, 2012).
* [6]  Sabrina De Capitani et al, Database Security (Wiley, 2002).
* [7]  Loic Gammaitoni et al, Verifying Modelling Languages using Lightning (University of Luxembourg, 2014).
* [8]  Ajeet Kumar et al, Early Software Reliability Prediction: A Fuzzy Logic Approach (Springer, 2013).
* [9]  Rob Seater and Greg Dennis, Tutorial for Alloy Analyzer 4.0 (MIT, 2014).

[10] Barry Boehm and Victor R. Basili, Software Defect Reduction Top 10 List (IEEE, 2001). [11] Tony Patton, Determine when to use stored procedures vs. SQL in the code (TechRepublic, 2005). [12] T. M. Murali, SQL and Relational Algebra (VirginiaTech Engineering College, 2010).

[13] Khaleel Ahmad and Nitasha Varshney, On Minimizing Software Defects during New Product Devel- opment Using Enhanced Preventive Approach (IJSCE, 2012).

[14] Daniel Jackson, Micromodels of software: lightweight modelling and analysis with Alloy (MIT, 2002). [15] Michael Brackett, Data Architecture and Data Structures (DATAVERSITY, 2013). [16] Christopher J. Date, An Introduction to Database Systems (Pearson, 2003).

30

Chapter 8

Appendix

8.1 Alloy Quick Reference

Full Reference: http://www.ics.uci.edu/ alspaugh/cls/shr/alloy.html

8.1.1 Logic

The Alloy logic is a first-order logic in which the domain is the set of all relations, and terms include relational expressions such as joins.

Everything in Alloy is a relation!

* A relation is a set of tuples of the same (positive) arity. Each tuple lists entities that are related to each other. The size of the relation is the number of tuples; the arity of the relation is the arity of the tuples.
* Sets are represented by unary relations. Each 1-tuple in the unary relation contains an element of the set.
* Scalars are represented by singleton sets. Since a set is a unary relation, an scalar is thus represented as a singleton (size 1) unary relation.

As a result, the operators apply to relations, sets, and scalars, and there are very few cases that produce no result.

Page numbers refer to Daniel Jackson, Software Abstractions, MIT Press 2006.

31

CHAPTER 8. APPENDIX 32 8.1.2 Syntax

|  |  |
| --- | --- |
| Set constants 50 | |
| univ | The universal set |
| none | The empty set |
| Relation constants 50 | |
| iden | The identity relation |

|  |  |  |
| --- | --- | --- |
| Relation operators 55 | | |
| Symbol | Name | Syntax |
| -> | (Arrow) product | R1 -> R2 |
| . | Join | R1 . R2 |
| [] | Join (a second notation for it) | R2 [R1] |
| ̃ | Transpose | ̃R |
| ˆ | Transitive closure | ˆR |
| \* | Reflexive transitive closure | \*R |
| <: | Domain restriction | Set <: R |
| :> | Range restriction | R :> Set |
| ++ | Override | R1++R2 |
| Logical operators 69 | | |
| Symbol | Keyword | Name or result |
| ! | not | negation |
| && | and | conjunction |
| || | or | disjunction |
| => | implies | implication |
| <=> | iff | logical equivalence |
|  | else | A=>B else C ≡ (A&&B)||(!A&&C) |
| Quantifiers/predicates 70 | | |
|  | Quantification Q var:set | formula | Predicate on relations Qe |
| all | universal | — |
| some | existential | size is 1 or greater |
| no | ¬∃ | size is 0 |
| lone | zero or one exists | size is 0 or 1 |

one

exactly one exists singleton

Set operators 52

Symbol

Name

Result

+

Union

A set

&

Intersection

-

Difference

in

Subset

T or F

=

Equality

CHAPTER 8. APPENDIX 33

|  |  |
| --- | --- |
| let 73 | |
| let x = e | A | A with every occurrence of x replaced by expression e |

Signatures and relations

(Parts of this subsection describe the Alloy language.) Each set of atoms is defined by a signature, with keyword sig. A signature can contain zero or more relation declarations, separated by commas. Each declaration

names a (binary) relation between the set defined by the signature and a set or relation.

// Simple example

abstract sig Person {

father: lone Man,

mother: lone Woman

}

sig Man extends Person {

wife: lone Woman

}

// Signature

// A declaration

// Another declaration

sig Woman extends Person {

husband: lone Man

}

The extended signature must be either a top-level signature or a subsignature.

Constraining a declaration

There are two ways:

1. with set or relation multiplicity constraints in the signature. These are a quick shorthand. The ex- ample above has several of these (all are lone).
2. with a fact 117 that states a constraint on the set or relation. The constraint is expressed in the Alloy logic.  (The fact keyword may be omitted if the fact is only about the relations of a single signature, and it immediately follows that signature — then it is a signature fact, and is implicitly universally quanti- fied over the signature’s set, and may use this as if it were the variable of this implied quantification.)

Multiplicity constraints in declarations

Relationships among signatures

Every S is a T, and every U is a T

S in T U in T

S extends T U extends T

subset

An S can also be a U



extension

An S cannot also be a U

|  |  |
| --- | --- |
| Set declarations with multiplicities 76 | |
| e is a expression producing a set (arity 1) | |
| x: set e | x a subset of e |
| x: lone e | x empty or a singleton subset of e |
| x: some e | x a nonempty subset of e |

x: one e

x a singleton subset of e (i.e. a scalar)

CHAPTER 8. APPENDIX 34

|  |  |
| --- | --- |
| x: e | x a singleton subset of e (equivalent to one) |
| Relation declarations with -> multiplicities 77 | |
| A and B are expressions producing a relation m and n are some, lone, one, or not present (which is equivalent to set) | |
| r: A m -> n B | m elements of A map to each element of B |



Facts

each element of A maps to n elements of B

117 A fact contains a formula in the Alloy logic that is assumed to always be true. See the Alloy language for more details.

Disjointness

71 disj before a list of variables restricts their bindings to be disjoint.

Cardinality constraints

80 The prefix operator # (cardinality) on a relation produces the relation’s size. The result can be operated on with + - = < > =< >=. Positive integer literals can appear in cardinality expressions.

sum x: e | ie sums the value of ie for each x in set e.

8.1.3 Modelling

The Alloy language uses the Alloy logic plus some other constructs to make models. "a description of a software abstraction" 4.

(Recall that in FOL a model means something different.)

Language constructs

The Alloy language adds these constructs to the Alloy logic:

In Alloy, a model is

1. A module line gives the relative pathname of the model’s file (minus the ".als" suffix). The pathname is relative to the directory that imported module pathnames are going to be relative to. (Obviously, the module line is mostly redundant with the file’s full pathname.)
2. A sig (signature) declares one or more sets of atoms, and their relations to other sets.
3. A fun (function) defines a way of getting a relation (or set, or atom). It can take parameters that are used in getting its result. It can define a relation (usually using ->) and make use of it to produce its result. It is a FOL function for the Alloy logic, in which expressions are relations.
4. A pred (predicate) defines a formula (true or false). It can take parameters that are used in getting its result. It is a FOL predicate for the Alloy logic.
5. A fact defines a formula that you assume is valid (always true, for any world). The Alloy analyzer uses facts as axioms in constructing its examples and counterexamples.
6. You run a predicate in order to see the examples (if any) the Alloy analyzer finds for which the predicate is true.  You define the scope that the analyzer checks by saying things like "run for 3" or "run for 3 but 4

Dog". The analyzer will then check only possible examples that contain no more than that many of atoms from each set.

If it finds an example, then the predicate is satisfiable.

If it finds no examples, the predicate may be either invalid (false for all possible examples); or it may be satisfiable but not within the scope you used.

CHAPTER 8. APPENDIX 35

1. An assert (assertion) defines a formula that you claim will always be true. An assertion differs from a fact in that the Alloy analyzer will check an assertion to see if it is true for all the examples in a scope, whereas the analyzer assumes each fact is true and uses them to constrain which examples it looks at.
2. You check an assertion in order to see whether the Alloy analyzer finds any counterexamples.  You define the scope as for a run command.  If it finds a counterexample, then the predicate is unsatisfiable.  If it finds no counterexamples, the predicate may be either valid (true for all possible examples); or it may be unsatisfiable but not within the scope you used.

Which construct to use where?

1. Writing a model (Alloy file) that might need to import other models? Use module.
2. Need a set of atoms? Use a sig.
3. Need an expression, whose value is a function (or set, or scalar)? Use a fun (function).
4. Need a formula, whose value is true or false? Use a pred (predicate).
5. Need to state an axiom that you want to be true always? Use a fact (function).
6. Need an example for which a pred is true? run the predicate to see if one exists. It’s like using an existential quantifier over all the predicate’s parameters.
7. Want to claim something is always true? Use an assert (assertion).
8. Want to see if an assert is unsatisfiable? check the assertion to see if any counterexample can be found.

8.1.4 Signatures

|  |  |
| --- | --- |
| Signatures 91 | |
| sig A {fields} | Declares a set A of atoms |
| sig A extends B {fields} | Declares a subset A of set B, disjoint from all other extends subsets of B |
| sig A in B {fields} | Declares a subset A of B |
| sig A in B + C {fields} | Declares a subset A of the union (+) of sets B and C |
| abstract sig A {fields} | Declares a set A that contains no atoms other than the ones in its subsets (if any) |
| one sig A {fields} | Declares a singleton set A |
| lone sig A {fields} | Declares a set A of 0 or 1 atom |
| some sig A {fields} | Declares a nonempty set A |
| sig A, B {fields} | Declares two sets A and B of atoms Wherever A appeared above, a list of names can appear |
| Fields (in a signature for set A) 95 | |
| f: e | Declares a relation f that’s a subset of A->e. e can be any expression that produces a set — union, intersection, ... , any combination. |
| f: lone e | Each A is related to no e or one e. |

f: one e Each A is related to exactly one e.

CHAPTER 8. APPENDIX 36

|  |  |
| --- | --- |
| f: some e | Each A is related to at least one e. |
| f: g->h | Each A is related to a relation from g to h. |
| f: one g lone -> some h | The multiplicities have their usual meanings. Here, each A is related to exactly one relation relating each g to 1 or more h’s, and each h is related to 0 or 1 g. |

8.1.5 Functions

8.1.6 Predicates

8.1.7 Facts

8.1.8 Assertions

|  |  |
| --- | --- |
| Function 121s | |
| fun Name [parameters] : type {e} | Defines a function, with the given name and (possibly empty) parameters, and producing a relation (or set, or scalar) of the given type. The result is defined by the expression e, which may reference the param- eters. |
| Predicates 121 | |
| pred Name [parameters] {f} | Defines a predicate, with the given name and (possibly empty) parameters. A predicate always produces true or false, so no type is needed. The result is defined by the formula f, which may reference the parameters. |
| Facts 117 | |
| fact {e} | The expression e is a constraint that the analyzer will assume is always true. |
| fact Name {e} | You can name a fact if you wish; the analyzer will ignore the name. |
| Assertions 124 | |
| assert Name {f} | Defines a assertion, with the given name. Assertions take no parameters. An assertion always produces true or false, so no type is needed. The result is defined by the formula f. |