University of Amsterdam

Master Thesis Project

Forgery
Databases Generation and Transactions Synthesizing based on Alloy models

Guy - Sean Rombaut

Supervised by:
Dr. Tijs van der Storm
Dr. Jouke Stoel

Organizations:

Internationale Nederlanden Groep (ING)

Centrum Wiskunde & Informatica (CWI)

September 30, 2016

Contents

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 [6, p. 6]. By providing immediate feedback to users they allow early detection of design errors [7]. 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.

We decided to focus our research on the field of automatic database generation rather than whole programs as we found it very interesting. Considering the human factor, it is not possible to develop a fault-free software in practice [9]. When those issues occur the data itself may also be effected and this makes it more difficult to repair. Data errors can harm the reputation of an organization, diminish financial gains and create uncertainty in an organization.

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.

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 or misunderstandings between the teams.

The project was initially called *Fors* and later on was renamed to *Rebel*. *Rebel* is a domain specific language (DSL) that parses business software specifications into algebraic-based language which is called *Alloy*. With *Alloy* it is possible to validate models, and those can be implemented by the technical team.

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

Chapter 3

Background

This chapter discusses the background of developing *Forgery*, the previous work that has been done and the motivation for it. It also contains 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 [13, 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 [11, p. 1]. Changing specifications after implementation of software often takes much more time and is also more expensive [2].

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 [11, 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 [11]. 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. [16]. In 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 a 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 organization, diminish financial gains and create uncertainty in an organization. Synthesizing the data may reduce or even 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) [4, p. 11].

The scope of 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 [3]. 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 by use of transactions it is possible to avoid inconsistency of data (corollary of the previous point). Third, security (permissions) and standards (e.g. representation of the data) can be enforced. And last, using a database simplifies sharing data between multiple workstations [4, p. 16].

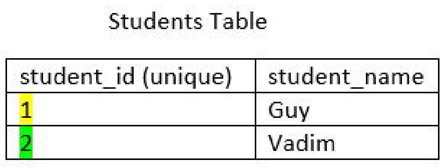
Fors uses *Alloy*; and *Alloy* is based on relations. We therefore investigated if it is possible to make a link between Alloy and relational databases. We consequently investigated the possibility to automatically generate a matching database.

A relational database consists of three main principles:

- Tables are the logical structure (although physically they can be stored in multiple ways like binary trees, hashing etc).
- The information principle - the entire content of the database is represented in one specific way and only that way.
- The operators available to the user derive from an old state to a new one.

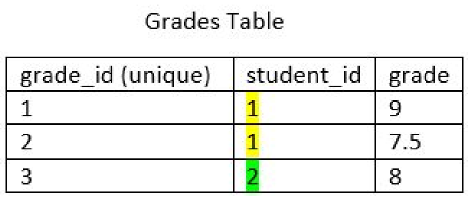
A relational database has the prefix *relational* not only because of entities and relationships but primarily because of the fact that relation is a mathematical term for a table [4, p. 26]. A relational system is based on the relational model of data.

Example - Simple student grades system:



Students Table

student_id (unique)	student_name
1	Guy
2	Vadim



Grades Table

grade_id (unique)	student_id	grade
1	1	9
2	1	7.5
3	2	8

In this example, we used the students table to store the names of the students. We used the grades table for storing the student's grades. In order to make a link between a student and a grade we used a unique numeric identifier.

The system user (e.g. a teacher) can now use multiple operations to manipulate the data. Each performed operation (e.g. deleting data) will generate a new table by changing the table from an "old" into a "new" state. For example, by deleting a grade row, this relation will be replaced by a new one (excluding the deleted row). In a similar fashion new tables will be generated when inserting or updating data.

We have already mentioned the term *relation* multiple times. Before we can fully define a relation we first have to introduce few more terms, which are crucial for understanding what a relation is.

Given a collection of data types (e.g. names, dates, addresses etc) $Ti(i = 1, 2, 3...)$, an attribute Ai is a pair of a data type Ti and a value of this type Vi (attribute value). So a tuple t, say - is a set of attributes.
$tn(n = 1, 2..) = < A1, V1 >, < A2, V2 >, ... < An, Vn >$
Where n is the arity of t (unary, binary etc) [4, p. 142].
For example, a binary tuple of type *contact*, we have an attributes of name and address so $t =<< Name, Guy >, < Address, Amsterdam >>$

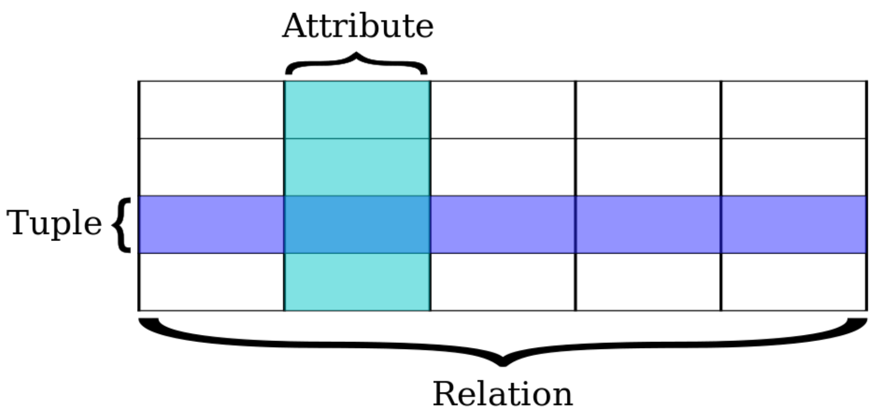
Few rules regarding tuples:

- Each tuple contains exactly one value for each of its attributes.
- Every subset of a tuple is a tuple.
- There are no duplicate tuples.

A relation r consists of heading and a body. The body of r is a set of tuples. And the heading of r is its tuples attribute types [4, p. 146].

Example - Relation: $Grades = \{<< Student, Guy >, < Grade, 9 >>,$
$<< Student, Guy >, < Grade, 7.5 >>,$
$<< Student, Vadim >, < Grade, 8 >>\}$

In summary, if a relation is a table, the rows are the tuples, and the columns are the attributes. The heading of the table defines the data types.



The relations theory provides a set of operations which we can apply on relations - such operations allow us to add, delete or modify data. For example, subset of $\subset$, superset of $\supset$, equals $=$ and others.

Practically, when it come to databases, many of them use SQL (Structured Query Language). SQL is a declarative language designed for constructing relational databases and managing the data that is held in it. For example, we can send a query to insert a new data.

In our thesis, we will try to convert Alloy syntax into SQL. *Alloy* is based on the notion of relations and it uses relations as its main structure. Therefore, the data model of *Alloy* can be translated directly into a relational database schema, using SQL. Obviously, *Alloy* is much more than a data structure. One of the other specifications of *Alloy* is that, it provides functions. We will discuss *Alloy* in depth in the next chapter.

3.2 Problem Analysis

Using only Alloy specifications to automatically generate a database system has been challenging. In this section we will discuss the reasons.

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 can contain empty data (null) and duplicates, while relations/tuples can not.
- In database tables, type names are usually omitted, relations usually involve a type name.

Another difficulty is that, database systems make use of *keys* (primary, foreign and unique keys) that are a vital part of the table structure (e.g. avoiding duplicates and even improve performance). *Alloy* only partially deals with keys or not at all.

In order to generate tables that represent the corresponding Alloy relations, we need to agree on certain rules of interpreting these relations (e.g. row orderings are irrelevant) [4, p. 151].

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 are insert, update or delete. We need to bridge between algebraic notation and a SQL query notation.

Furthermore, 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.

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 chapters.

3.3 Research Question

In this paper the following research question has been addressed:

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 of 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. Alloy used for modeling software systems. Alloy includes a tool called Alloy Analyzer for visualizing models, and for exploring and checking the properties of them.

In this chapter we concentrate on the language part of Alloy, and we will present various examples as demonstration. We will not cover the whole language but focus on the essentials required to understand this work.

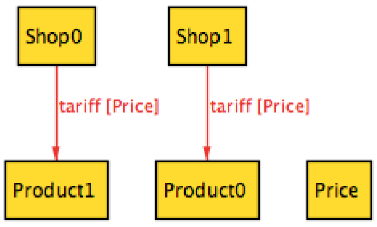
4.1 Data Structure

Alloy data model is based on *atoms*, *signatures* and *fields*. Atom is the most basic specific element in Alloy. **sig**nature subsequently, is a set of atoms that also defines the data type. A field then describes the relation between different types of signatures. It is therefore a set of tuples that consists of different types of data. Finally, in Alloy these relations can be unary or binary (using $\rightarrow$).

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

```
sig Price {}
sig Product {}
sig Shop {
        tariff: Price -> Product
}
```

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

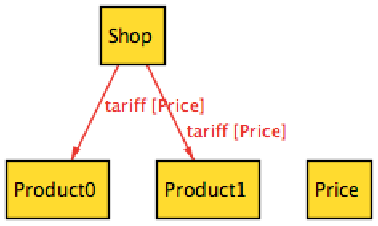


4.2 Quantifiers

Quantifier refers to the amount of elements in a set. Alloy allows us to define quantifier 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). and *all*/*set* (size is >= 0) which is used by default. For this example, we assume that we want only one shop in our model:

```
sig Price {}
sig Product {}
one sig Shop {
          tariff: Price -> Product
}
```

Alloy Analyzer will now only generate examples with exactly one shop.

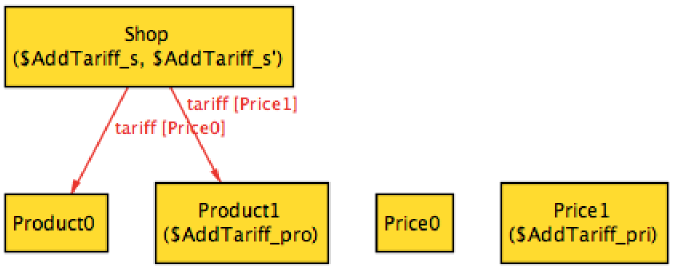


4.3 Data Operations

Alloy allows defining **pred**icates as operations that act on a specified model. A predicate may convert a certain state of the data system into a new one, based on the rules it defines. The predicates accept signatures as an input. The operations themselves are 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.

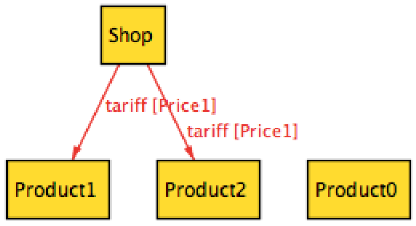
```
sig Price {}
sig Product {}
one sig Shop {
          tariff: Price -> Product
}

pred AddTariff(s, s' : Shop, pro: Product, pri: Price) {
          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'.

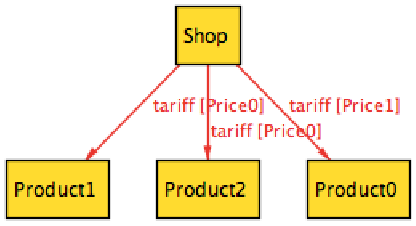
4.4 Data Invariants

Alloy allows to create system invariants using **fact**s. 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 (*Product*0).



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

```
sig Price {}
sig Product {}
one sig Shop {
          tariff: Price -> Product
}

pred AddTariff(s, s' : Shop, pro: Product, pri: Price) {
          s'.tariff = s.tariff + (pri -> pro)
}

fact ProductMustHaveShop {
        all pro: Product | pro in Shop.tariff[Price]
}
```



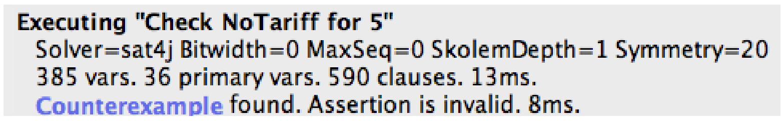
4.5 Assertions

Assertions are constraints that were intended to follow from facts of the model. **Assert**ions 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.

Our new requirement now is that we want that each shop will have maximum one tariff, but such invariant was not specified in our model yet.

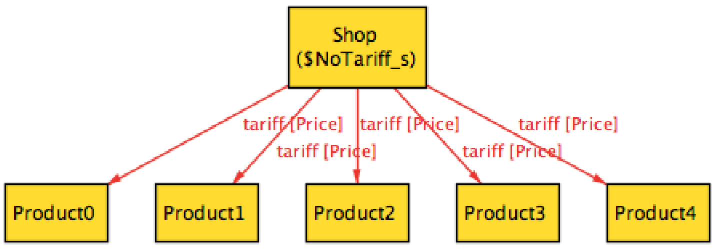
```
sig Price {}
sig Product {}
one sig Shop {
        tariff: Price -> Product
}

pred AddTariff(s, s' : Shop, pro: Product, pri: Price) {
        s'.tariff = s.tariff + (pri -> pro)
}

fact ProductMustHaveShop {
        all pro: Product | pro in Shop.tariff[Price]
}

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

By simulating the model, all the assertions are checked *LoneTariff*, Alloy Analyzer will alert that it found a counter example:



```
Executing "Check NoTariff for 5"
  Solver=sat4j Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
  385 vars. 36 primary vars. 590 clauses. 13ms.
  Counterexample found. Assertion is invalid. 8ms.
```

By clicking on the counterexample, Alloy Analyzer will present all found counter-models. The following 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 is more robust minimizing mistakes. Assertions are similar to unit testing which are popular among programming languages.

Chapter 5

Forgery Solution

In this chapter we will discuss on the solution that we offer. First we will introduce the key ingredients with few examples, and then we will go into deeper details.

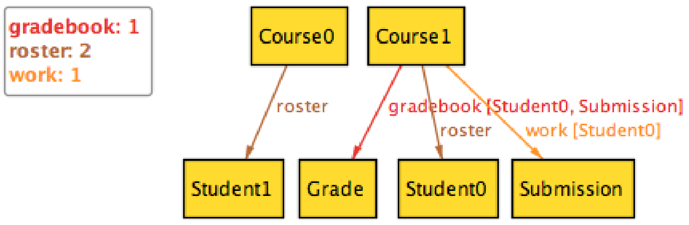
5.1 Key Ingredients

As introduced in the background Alloy specification consists of four main parts: Signatures (and their Quantifiers), 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 an implemented 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 and two students which are enrolled to each of those courses. E.g. Student0 is enrolled to Course1. Student0 also submitted his work and he was graded 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 table that points to the relevant signature tables. Each Signature table stores its atoms. The relations are linked by id's which are also the primary keys.

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

```
CREATE TABLE `student`(
        `id` INT(6) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
        `value` VARCHAR(100) NULL
);
```

The relation *roster* will result in a junction table that is created by the following SQL code:

```
CREATE TABLE `roster`(
        `id` INT(6) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
        `course_id` INT(6) UNSIGNED NOT NULL,
        `student_id` INT(6) UNSIGNED NOT NULL,
        UNIQUE INDEX ui(`course_id`,`student_id`),
        FOREIGN KEY (`course_id`) REFERENCES `course`(`id`),
        FOREIGN KEY (`student_id`) REFERENCES `student`(`id`)
);
```

We use foreign keys to enforce integrity. These constraints guarantee that, for example, a row in the table *roster* with a field *student_id* referencing the *student* table will never have an *student_id* 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 in, 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 operations on the database.

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 +, - and set operations such as union, intersection, difference etc. That gives us enough flexibility to transform Alloy predicates into SQL procedures.[8]

We use the tag symbol (') for describing a state transition. For example, student s is the pre-condition, and s' is the post-condition. Also, we use the input underline prefix as an easy way to create new atoms. Each predicate converted to a SQL procedure, wrapped by transaction. In case of failure, a rollback will be applied.

The following predicate enrolls a new student to a course.

```
pred enroll(c, c': Course, _s: Student) {
        c'.roster = c.roster + _s
}
```

We generate a procedure that contains two Insert queries as the following:

```
DELIMITER //
CREATE PROCEDURE `p_enroll`(IN c INT(9), IN _s VARCHAR(100))
BEGIN
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
        ROLLBACK;
END;

START TRANSACTION;
        INSERT INTO student(`value`) VALUES (_s);
        SET _s = LAST_INSERT_ID();
        IF NOT EXISTS
                (SELECT `id` FROM `roster` WHERE `student_id`=_s AND `course_id`=c)
        THEN
                INSERT INTO `roster` (`student_id`, `course_id`) VALUES (_s, c);
COMMIT;
END //
```

This procedure accepts two parameters: an existing course id (c), and a new student value (as mentioned - underline prefix refers to a new atom). The procedure is callable from running a normal query. For example:

```
p_enroll(3, 'Jonathan');
```

It will insert a new row with the value Jonathan to the student table, and another row to the roster table, with the new generated student id and the course id.

Similarly, *Facts* are also converted into procedures. *Facts* don't have inputs, however variables can be defined easily.

The following fact determine that all students must be enrolled to a course. The expression *all* is a quantifier which defines that the fact applies for all rows. Forgery support multiple quantifiers such as *some*, *one* and others. We will discuss them later.

```
fact mustBeEnrolled {
        all c: Course, s: Student |
        s in c.roster
}
```

Also in this case we create a SQL procedure.

```
DELIMITER //
CREATE PROCEDURE `f_mustbeenrolled`(OUT return_value TINYINT UNSIGNED)
BEGIN
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
        SET return_value=1;
END;

IF EXISTS
        (SELECT * FROM student WHERE id NOT IN (SELECT student_id FROM roster))
        THEN
                SET return_value=1
        ELSE
                SET return_value=0
END IF;

END //
```

Facts procedures modify the flag *return_value* to 1 or 0 (logic True or False) which are equivalent to invariant failure or not accordingly. When the data was modified, the procedures are automatically called and flag is being checked. In case of a invariant violation, the changes will be reverted to the previous valid state (using transactions).

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 Quantifiers: Creating SQL procedures for verifying the specified invariants and quantifiers.
5. Procedures Generator for Predicates: Creating SQL procedures for systemic operations according to the specifications.
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.

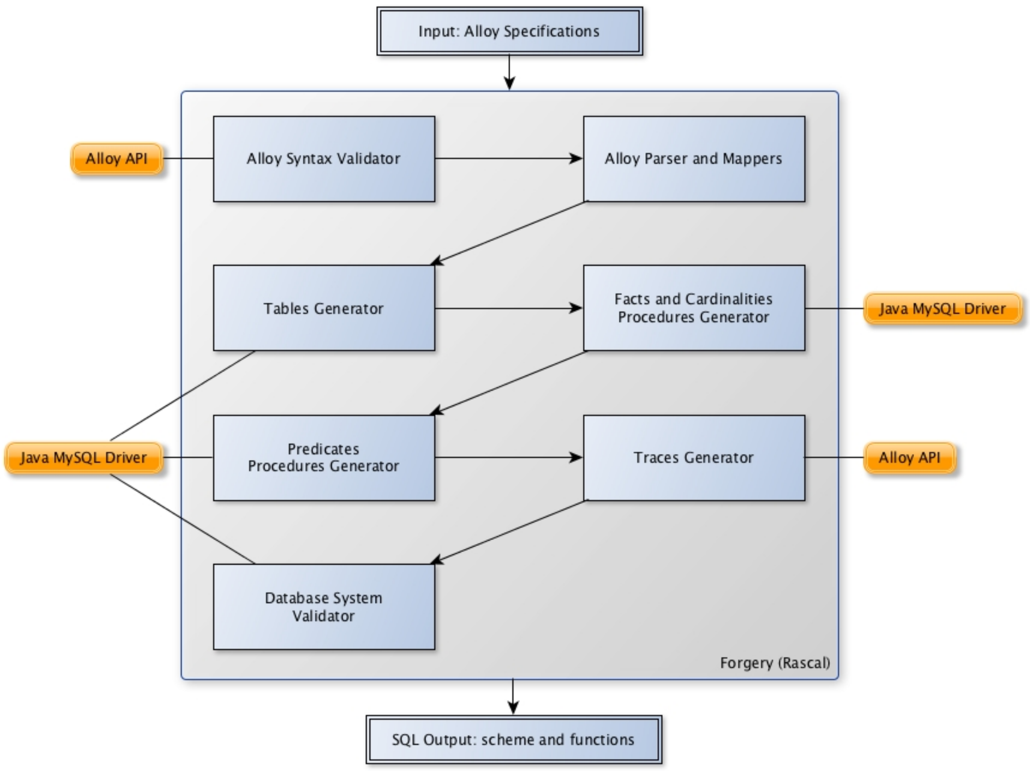

Input: Alloy Specifications
Alloy API
Alloy Syntax Validator
Alloy Parser and Mappers
Tables Generator
Facts and Cardinalities Procedures Generator
Java MySQL Driver
Java MySQL Driver
Predicates Procedures Generator
Traces Generator
Alloy API
Database System Validator
Forgery (Rascal)
SQL Output: scheme and functions

Figure 5.1: Forgery architecture

We implemented *Forgery* using *Rascal*. We chose *Rascal* due to its powerful DSL and AST (Abstract Syntax Trees) tools. Since we used a standard SQL language, any arbitrary database system can be used. We used *MySQL* due to his popularity and the provided set of tools.

5.3 Alchemy Comparison

Although related papers will be presented in a different chapter, there is a unique paper called Alchemy [15] that we have studied. 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.

5.3.1 Forgery uses pure SQL

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

5.3.2 Alchemy has no evaluation and assertions are not supported

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 write predicates. However, those predicate specifications might be in contradiction to invariants. They added an auto-repair functionality that fix the data in case it has a conflict. However, since the specifications are invalid by the nature of Alloy. The assertions and the analyzer cannot be used. Alchemy does not have evaluation method.

5.3.3 Forgery supports atoms control

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

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. We will also remind some definitions in regarding Alloy.

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 but they will be discussed later on.

```
sig Submission {}
sig Grade {}
sig Student {}
sig Course {
        roster: set Student,
        work: roster -> lone Submission,
        gradebook: work -> lone Grade
}
```

Alloy uses **sig**natures (e.g. Submission) to describe a data model [5, p. 30]. Every signature defines a 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.

5.4.1 Atom 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).

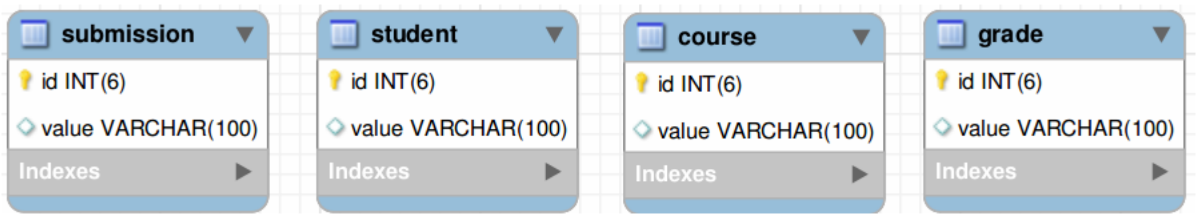


Figure 5.2: Generated tables for Signatures

5.4.2 Field Relations

The signature *Course* defines the following relations: roster (enrolled students), work and gradebook.
$roster = \{< Construction, Guy >, < Construction, Tijs > ..\}$
$work = \{< Construction, Guy, hwk1 > ..\}$
$gradebook = \{< Construction, Guy, homework, 8 > ..\}$

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 $\rightarrow$.

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 simple (see next chapter).

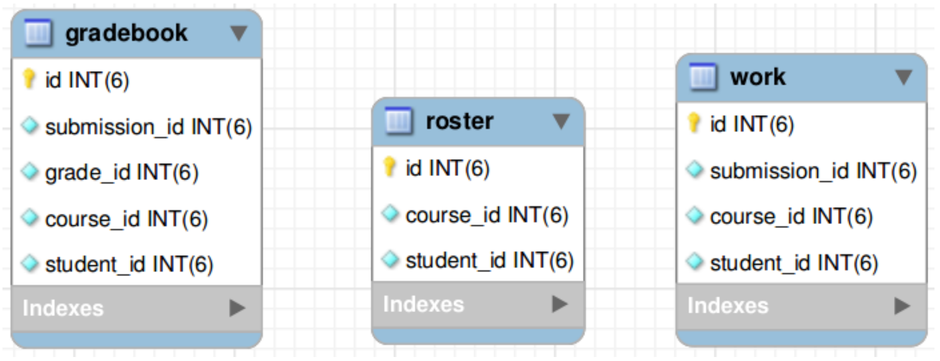


Figure 5.3: Generated tables for Relations

5.5 Procedures Generation

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) [10].

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 the SQL implementation of Quantifiers, Predicates and Facts.

5.5.1 Quantifiers

Alloy syntax supports multiple quantifiers to describe constraints on the data model: ***no*** (zero), ***all*** (for all), ***lone*** (at most one atom), ***some*** (at least one atom) and ***one*** (single atom).

Quantifiers can be used in multiple places and might have slightly different context. E.g. expressions, signatures, facts etc. For example, the constraint *lonee* says that the expression *e* denotes a relation containing at most one tuple. Or *onesigS*, for example, declares S to be a signature whose set contains exactly one element.

For each quantifier we create a procedure. Each procedure contains a query that counts the rows and group it based on the quantifier context and then compare it to the specified quantifier definition ($c <= 1$, $c == 1$, $c > 1$, $c == 0$, $c(*)$).

We implemented quantifiers (or cardinality constraints) as invariants. In other words, these procedures are automatically called every time when the database is changed. If any violation occurs, the data will be reverted and error will be shown (transactional action). The quantifier procedures are stored in the database and can be identified with the prefix *q_*.

The following example applies cardinality constraints on *school* and *work*.

```
sig Student {}
sig Submission {}
one sig School {}
sig Course {
        roster: set Student,
        work: roster -> lone Submission
}
```

The expression *work* : *roster* → *lone Submission*, which allows at most one submission per student, will be converted to the following procedure:

```
DELIMITER //
CREATE PROCEDURE `q_work`(OUT return_value tinyint unsigned)
BEGIN
        IF EXISTS (
                SELECT * FROM `course`
                LEFT JOIN `work` ON `work`.`course_id`=`course`.`id`
                GROUP BY `submission_id`, `course_id`, `student_id`
                HAVING COUNT(`work`.`id`) > 1
        ) THEN
                set return_value = 1;
        ELSE set return_value = 0;
        END IF;
END //
```

Or when it comes to context of signatures, *one sig School* (only one school exists in the system) we generate the following procedure:

```
DELIMITER //
CREATE PROCEDURE `c_work`(OUT return_value tinyint unsigned)
BEGIN
        IF EXISTS (
                SELECT * FROM `school` HAVING COUNT (`id`) != 1
        ) THEN
                set return_value = 1;
        ELSE set return_value = 0;
        END IF;
END //
```

5.5.2 Predicates

In this section we will discuss about the logic behind the interpretation of Alloy predicates and creation of atoms. The same example from the previous sections will be expanded.

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) {
        s not in c'.roster
}
pred 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 AssignGrade (c, c' : Course, s : Student, b : Submission,
g : Grade) {
        c'.gradebook = c.gradebook + (s -> b -> g)
}
```

Alloy Predicates

Alloy uses **pred**icates (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.

predDecl ::= *pred name* [*paraDecls*] *block*
paraDecls ::= (*decl*, $*$)
block ::= *expr*$*$
expr ::= *qualName*
| *expr* [+ | −] *expr* | *expr* [. | →] *expr*
| *expr* [! | *not*] [*in* | =] *expr*
| *expr and expr*
| [*all* | *no* | *lone* | *some* | *one*] *decl*, +*blockOrBar*
decl ::= [*disj*] *name*, + : [*disj*] *expr*
blockOrBar ::= *block* | *bar expr*
bar ::= |

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 [15], 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:

```
INSERT INTO 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 (or using a new line separator). Forgery handles them as a list of operations which performed serially one after one (transactionally - all of them must succeed before the commit). Forgery assumes that the formulas are correct because they are first tested using Alloy Syntax Checker. For example, two operands of an operation must be of the same relation type. Also, each formula refers to a pre- or a post-condition.

Supported operators:

1. (*not*)*in* operator - Used as precondition to check if the element exists or not. [Output: True or False]. 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. Union - the operator $'+'$ is used to compute union of sets. [Output: Set]. Example: $c'.roster = c.roster + s1$

```
INSERT INTO `roster` (`student_id`, `course_id`) VALUES(s1, c);
```

3. Difference - the operator $'-'$ works similarly to union, but with delete statement instead. [Output: A set].
   Example: $c'.roster = c.roster - s1$

```
DELETE FROM `roster` WHERE `student_id`=s1 AND `course_id`=c;
```

4. Join - the operator is represented by square braces [] or using the dot (.) sign (although in other languages it usually means object access).
   Note: $r1.r2 <=> r2[r1]$.
   In *Delete* operations join is performed using the "WHERE" selector.
   In *Insert* operations join is performed like a tuple (see tupling example).
   In other cases, it is used as an independent precondition.
   Example: $no\ c'.work[\_sNew]$

```
 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;
```

   Note: In some cases an inner join would be a better practice as it also checks the values in the relevant foreign tables. For example, it will check that a reference id in one table actually points to an existing row in another table. However, in *Forgery* we use Foreign keys, which forces the data to be consistent, and rows cannot be deleted if other rows reference to them.

```
 IF EXISTS
 (SELECT *
FROM `work`
          INNER JOIN `course` on work.course_id = course.id
          INNER JOIN `submission` on work.submission_id = submission.id
WHERE course_id = c AND student_id = _sNew)
  THEN
          SELECT 'An error has occurred, operation rollbacked
          and the stored procedure was terminated';
          ROLLBACK;
END IF;
```

5. Tupling - represented by $\rightarrow$ symbol. As mentioned before, a tuple is equivalent to a table row (containing attributes / columns). We use it for operations such as insert or delete.
   Example: $c'.work = c.work + (s1 \rightarrow \_b) + (s2 \rightarrow \_b)$

```
INSERT INTO `work` (course_id, student_id, submission_id)
          VALUES (c, s1, _bNew), (c, s2, _bNew);
```

6. Equality - represented by $=$ symbol. We use checksum to see if tables are equal.

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
```

```
        ROLLBACK;
END;

START TRANSACTION;
        INSERT INTO `submission` (`value`) VALUES (atomVal);
        -- facts and quantifiers are included here.
COMMIT;
END //
```

As mentioned before, *Forgery* verifies all the invariants (quantifiers and facts) in each procedure. This rule also applies here.

5.5.3 Facts

A *fact* is a constraint that always holds (that can also be regarded as an assumption). Similarly to quantifiers, we use procedures as an implementation for such invariants. Those procedures are called when the database is changed and revert the data in case of failure. This way, we can guarantee that the data always valid.

factDecl ::= *fact* [*name*] *block*
block ::= *expr*∗
expr ::= *qualName*
| *expr* [. | →] *expr*
| *expr* [! | *not*] [*in* | =] *expr*
| *expr and expr*
| [*all* | *no* | *lone* | *some* | *one*] *decl*, +*blockOrBar*
decl ::= [*disj*] *name*, + : [*disj*] *expr*
blockOrBar ::= *block* | *bar expr*
bar ::= |

A fact consist of a name and the constraint, which is given as a block of algebraic sequence (written like in predicates, see operators in the previous section). We can also define variables and quantifiers criteria. E.g. The constraint *one* x : $S|F$ says that there is exactly one x that satisfies the constraint F.

The following constraint defines that all students must be enrolled.

```
sig Student {}
sig Course {
        roster: set Student,
}

fact mustBeEnrolled {
        all c: Course, s: Student |
        s in c.roster
}
```

We convert this fact into the following procedure:

```
DELIMITER //
CREATE PROCEDURE `f_mustbeenrolled`(OUT return_value TINYINT UNSIGNED)
BEGIN
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
        SET return_value=1;
END;

IF EXISTS
        (SELECT * FROM student WHERE id NOT IN (SELECT student_id FROM roster))
        THEN
                SET return_value=1
        ELSE
```

```
                        SET return_value=0
END IF;

END //
```

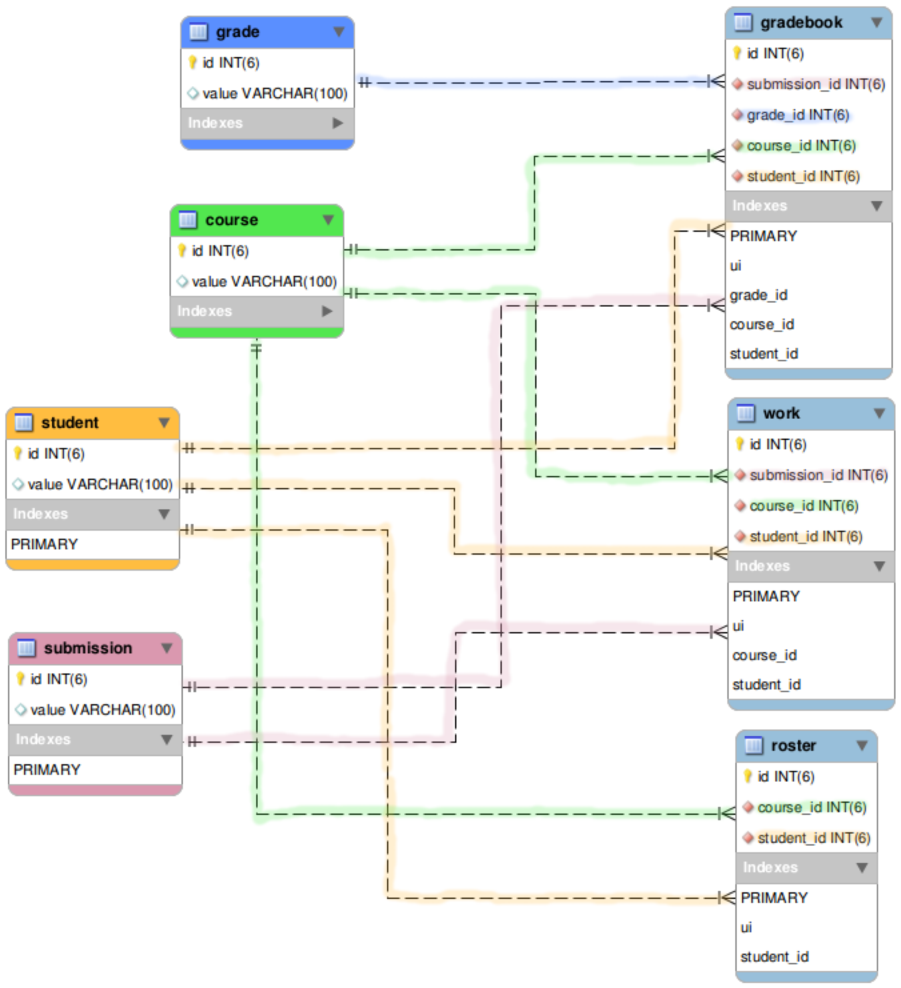


Figure 5.4: Generated Relational Database

5.6 Forgery Algorithms

5.6.1 Scheme Generation

This algorithm generates a SQL code that represents a database schema creation. For each atom we generate a table that contains a pair of identifier and value. For each signature we create a junction table. Each of those junction tables contain fields that points to the relevant atom (using identifiers) based on relations. Since the relations can be recursive (points to another relation) we flatten them so they will point directly to atoms. In addition, we create foreign keys for those pointers and define the identifiers as primary keys.

The scheme is generated based on the signatures, fields and quantifiers.
Let S be a list of signatures, s is a single signature, R is its contained relations, r is a single relation.

Algorithm 1 Returns the database structure

```
function CREATE SCHEME($S$)
    $tables = ()$                                          ▷ map (table name: list of fields)
    for each signature $s$ in $S$ do
        add ($s.name : id, value$) to $tables$
        for each relation $r$ in $s.R$ do
            $fields = \{\}$                                ▷ set of table fields with quantifiers
            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 ($r.name : fields$) to $tables$
        end for
    end for
    return $tables$
end function
function ATOMIC FIELDS($r, tables, S$)
    if $r.name$ in $S.names$ then
        return $r.name$.‘_id‘
    end if
    return $tables[r.name]$
end function
function GENERATE QUERY($tables$)
    $query = ””$
    for $table$ in $tables$ do
        add SQL Statement ("Create Table", $table.name$) to $query$
        for $field$ in $tables[table]$ do
            add SQL Statement ("Create Column", $field.name$) to $query$
            add SQL Statement ("Foreign Key", $field.name$) to $query$
            if $field.quantifier$ is $lone$ or $one$ or $set$ then
                add SQL Statement ("Create Unique Index", $fields$) to $query$
            end if
            if $field.quantifier$ is $some$ or $one$ then
                // Quantifier will be created using procedures
            end if
        end for
        add SQL Statement ("Create Column", ‘$id$‘) to $query$
        add SQL Statement ("Create Primary Key", ‘$id$‘) to $query$
    end for
    return $query$
end function
```

5.6.2 Procedures Generation for Predicates

This algorithm generates a SQL code that represents procedures creation for predicates. Each procedure we wrap with an exception thrower that performs a rollback operation in case of failure in the state changes. For each input of a predicate we check whether it is an identifier or it refers to a new atom. If it refers to a new atom then we insert it to its relevant table and extract its new identifier.

Then we process the body of the predicate, which is a list of algebraic operations. We simplify it to joins and flatten structure. We convert the supported Alloy subset into its equivalent SQL syntax. When it comes to post-condition, each operation can be insert or delete. With pre-conditions we check if the condition exists (e.g. the atom exists or not), and in case such condition violated we revert the changes and stop the execution.

Let p be a predicate, containing its name, its inputs and its body. The body contains the pre- and the post-conditions (operations). l is a line of operation. i is an input. It contains its type, its name and its value.

Algorithm 2 Returns the procedures creation query

```
function GENERATE PREDICATE PROCEDURES(predicates)
    query = ""
    for p in predicates do
        add SQL Statement ("CREATE PROCEDURE p.name (list of p.inputs)") to query
        add SQL Statement ("DECLARE EXIT HANDLER FOR SQLEXCEPTION; BEGIN; ROLLBACK; END; START TRANSACTION;") to query
        for i in p.inputs do    ▷ if it is a new atom, we need to insert it, and get its new generated id.
            if i.name[0, 1] is "_" then
                add SQL Statement ("INSERT INTO i.type SET value='i.value'; SET var_i.name = LAST_INSERT_ID();") to query
            end if
        end for
        for l in p.body do    ▷ Operations over tables
            l = Simplify tuples and joins (l)
            if l.algebraicOperation is + then
                add SQL Statement ("IF NOT EXISTS INSERT INTO l.op1 SET value='p.varInputs[l.op2]'") to query
            else if l.algebraicOperation is − then
                add SQL Statement ("IF EXISTS DELETE FROM l.op1 WHERE value='p.varInputs[l.op2]'") to query
            else if l.algebraicOperation is in then
                add SQL Statement ("IF NOT EXISTS (SELECT * FROM l.op2 WHERE value='p.varInputs[l.op1]') THEN (SELECT 'precondition violated'; ROLLBACK);") to query
            else if l.algebraicOperation is notin then
                add SQL Statement ("IF EXISTS (SELECT * FROM l.op2 WHERE value='p.varInputs[l.op1]') THEN (SELECT 'precondition violated'; ROLLBACK);") to query
            else if l.algebraicOperation is = then
                add SQL Statement ("IF (CHECKSUM TABLE l.op1, l.op2) IS NULL THEN (SELECT 'precondition violated'; ROLLBACK);") to query
            else if l.algebraicOperation is != then
                add SQL Statement ("IF (CHECKSUM TABLE l.op1, l.op2) IS NOT NULL THEN (SELECT 'precondition violated'; ROLLBACK);") to query
            end if
        end for
        add SQL Statement ("COMMIT;") to query
    end for
    return query
end function
```

5.6.3 Procedures Generation for Invariants

This algorithm generates a SQL code that represents procedures creation for invariants. There are two types of invariants that we generate here: facts and quantifiers. Each procedure we wrap with a flag modifier that indicates whether the system is in a valid state or not. When the flag indicates of a violation the transacted operations will rollback. Each fact contains a list of conditions which we check similarly to post-conditions in predicates (see previous section).
In addition, for each quantifier we count the relevant rows in the table and check if it is valid according to the specified quantifier definition.

Let f be a fact, containing its name, its variables, and its body. The body contains the invariants. n is an invariant. Each variable v contains its type, its name and its value.
Let q be a quantifier, containing its type, its context and its restriction domain properties.

Algorithm 3 Returns the invariant procedures creator query

```
function GENERATE INVARIANT PROCEDURES($facts$, $quantifiers$)
    $query$ = ""
    for $fq$ in $facts + quantifiers$ do        ▷ Wrap procedures with invariant violation flag modifier
        add SQL Statement ("CREATE PROCEDURE $fq.name$ (OUT $return_value$ TINYINT UNSIGNED) BEGIN DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN SET $return_value$=1; END; IF EXISTS (") to $query$
        if $fq$ is $fact$ then
            for $l$ in $fq.body$ do        ▷ Operations over tables
                $tmpQuery$ = ""
                l = Simplify tuples and joins (l)
                if $l.algebraicOperation$ is $in$ then
                    add SQL Statement ("SELECT * FROM $l.op2$ WHERE id NOT IN (SELECT $l.op2_id$ FROM $l.op1$)'") to $tmpQuery$
                else if $l.algebraicOperation$ is $notin$ then
                    add SQL Statement ("SELECT * FROM $l.op2$ WHERE id IN (SELECT $l.op2_id$ FROM $l.op1$)'") to $tmpQuery$
                else if $l.algebraicOperation$ is = then
                    add SQL Statement ("(CHECKSUM TABLE $l.op1$, $l.op2$) IS NULL") to $tmpQuery$
                else if $l.algebraicOperation$ is != then
                    add SQL Statement ("(CHECKSUM TABLE $l.op1$, $l.op2$) IS NOT NULL") to $tmpQuery$
                end if
            end for
            if $fq.variables$ then
                extend SQL Statement ("SELECT * FROM (tmpQuery)") to $query$
                $query$+ = Generate Quantifiers (fq)
            else
                $query$+ = tmpQuery
            end if
        else if $fq$ is $quantifier$ then        ▷ Context of the quantifier
            if $fq.context$ is $signature$ then
                extend SQL Statement ("SELECT * FROM fq.tableName") to $query$
            else if $fq.type$ is $field$ then
                extend SQL Statement ("SELECT * FROM fq.tableName LEFT JOIN fq.field ON fq.fieldId='fq.tableName'.'id' GROUP BY fq.tableFields")to $query$
            end if
            $query$+ = Generate Quantifiers (fq)
        end if
        extend SQL Statement (") THEN SET $return_value$=1 ELSE SET $return_value$=0 END IF;") to $query$
    end for
    return $query$
end function
```

```
function GENERATE QUANTIFIERS($q$)
    $query$ = ""
    if $q.type$ is $lone$ then
        extend SQL Statement ("HAVING COUNT('id') > 1") to $query$
    else if $q.type$ is $one$ then
        extend SQL Statement ("HAVING COUNT('id') <> 1") to $query$
    else if $q.type$ is $some$ then
        extend SQL Statement ("HAVING COUNT('id') <= 1") to $query$
    else if $q.type$ is $no$ then
        extend SQL Statement ("HAVING COUNT('id') > 0") to $query$
    else if $q.type$ is $all$ then
        extend SQL Statement ("HAVING COUNT('id') == c(*)") to $query$
    end if
    return $query$
end function
```

5.7 Evaluation

5.7.1 Reverse Engineering

When *Alloy* simulates a model, it generates traces and saves them as a temporary XML file. Those traces describe the data history of the system and the transition between them.

The idea was to use those traces to imitate the same behavior in our generated database system. In other words, use the same initial data, and follow the same operations that Alloy did. Then, we are able to compare the results of Forgery to those in *Alloy*.

We use this reverse engineering technique to make sure our solution is consistent to *Alloy* behavior and evaluates our system that way.

However, we found that the traces miss some important information.

By default, the traces are not ordered and this has to be configured manually. Otherwise, the traces are useless because we cannot identify the order of the operations and imitate them. To enable ordering in Alloy, the ordering library has to be included and initialized using a predicate named *init*. Moreover, the logic behind the transition from one step to another has to be defined. It can be done using a fact named *traces*.

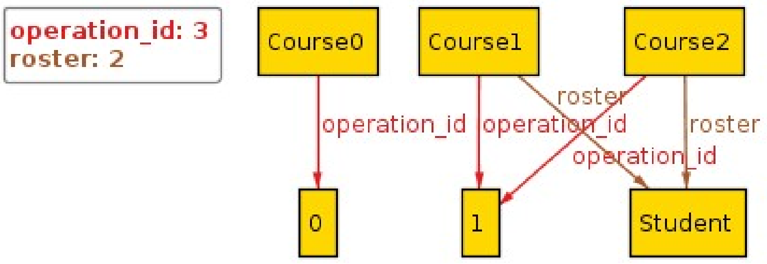
Besides, although we can now have an ordered list of the data history, we unable to identify which predicates were used. It means that we cannot sure which operation to use to imitate the transition between the different steps. In order to deal with it we used a small hack. We added an arbitrary atom called *operation_id* so Alloy will create traces for it. This atom identifies each operation. On each step we use the traces of this identifier to know what operation was used.

Forgery ignores the predicate *init*, the fact *traces* and the field *operation_id* as they are not relevant for generating the database system.

In the following example we added the mentioned tweaks, so we are able to generate the correct traces. The code is included, and later the data flow is explained briefly.

```
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:

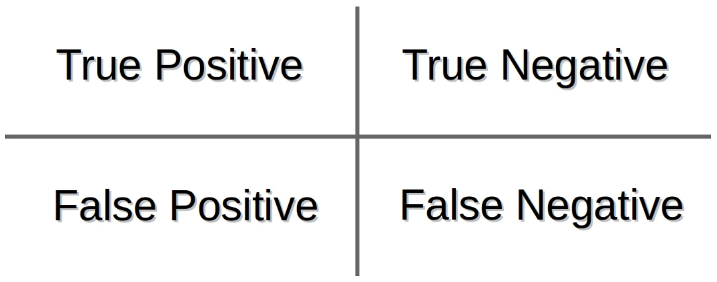


The data flow:

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.

5.7.2 Validation Scenarios

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



True Positive	True Negative
False Positive	False Negative

True Positive - False Negative

- True Positive: Correct models in *Alloy* and in *Forgery*.
- False Negative: Correct models in *Alloy* but incorrect in *Forgery*.

For those two cases, we iterate over all the correct *Alloy* models. And then, we imitate each step using the traces in Forgery 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*.

For those two cases, we use the assertions results (counter-example traces) and imitate each step in Forgery as described before.

Simulating those four scenarios in Forgery for different inputs will support us finding bugs in Forgery, and fix them so it will better matched to Alloy's behavior. Currently, the process itself is not fully automatic, however, we used multiple inputs and compared them in order to cover as many scenarios as possible.

Chapter 6

Related Work

Alloy is increasingly becoming a popular declarative modeling language. It is powerful by providing early error detection, supported by analysis tools for simulating and debugging models [7]. There were few related papers that were written about Alloy.

6.1 From UML to Alloy and Back Again

This paper presents a tool for transforming UML models (diagrams) to Alloy and vice versa. Basically, few techniques which are presented here can be used in our research of transforming Alloy into SQL. For example, for generating ASTs and analysis. In addition, the purpose of this UML-Alloy conversion allows to identify consistencies in those UML models which is similar to what we were trying to achieve with FORS for business models.

6.2 Alchemy: Transmuting Base Alloy Specifications into Implementations

As discussed previously, Alchemy compiles Alloy specifications into implementation of API for persistent databases. Alchemy translates a subset of Alloy predicates into update operations and it converts facts into database integrity constraints. This paper overlaps with some parts of our research. We studied this article in order to create a better tool. We discussed about the main differences and compared Alchemy against Forgery in section 5.3.

6.3 Mapping between Alloy specifications and database implementations

This paper identifies a subset of the Alloy language that is equivalent to a relational database schema with the most conventional integrity constraints. In light with Forgery, we had to review the common Alloy syntax that is used in FORS, and then trying to match them with the equivalent SQL. This paper presents some comparisons which we used for our need and helped us to identify difficulties in the conversation to database schema.

6.4 Towards an Operational Semantics for Alloy

This paper discuss about writing stateful Alloy specifications. It formalize a natural notion of transition semantics for state-based specifications and demonstrates it using examples. Database operations are also state-based and few ideas used for creating Forgery, especially for the evaluation part which we used traces (states) to validate Forgery.

Chapter 7

Conclusions

In this work we presented *Forgery* as a tool that supports the realization of *Alloy*-based specifications. We were able to generate a full-scale database including structural tables, constraints and functions for operations over data. The output is a pure SQL that doesn't require any external API or additional dependencies. 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 [9]. That also applies for *Forgery.* Hence, we developed a validation mechanism with reverse engineering that allows us to compare the results to those in *Alloy.* Moreover, enabling Assertions helps to find issues in the model; A feature that was not available in other related work like *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 or other resources.

There are still challenges to be solved in *Forgery*, for instance, expanding the set of supported *Alloy* syntax and resolving potential exceptions. Especially by continuing the development of the assertions and the reverse engineering system. However, to the best of our knowledge, considering multiple scenarios, *Forgery* has achieved his main purpose.

Bibliography

[1] Khaleel Ahmad and Nitasha Varshney, *On minimizing software defects during new product development using enhanced preventive approach*, IJSCE **2** (2012), no. 5, 9–12.

[2] Barry Boehm and Victor R. Basili, *Software defect reduction top 10 list*, IEEE Computer **34** (2001), no. 1, 135–137.

[3] Michael Brackett, *Data architecture and data structures*, DATAVERSITY, 2013.

[4] Christopher J. Date, *An introduction to database systems*, Pearson, 2003.

[5] Daniel Jackson, *Micromodels of software: lightweight modelling and analysis with alloy*, MIT, 2002.

[6] ———, *Software abstractions: Logic, language and analysis*, revised ed., The MIT Press, 2012.

[7] Pierre Kelsen Loic Gammaitoni and Fabien Mathey, *Verifying modelling languages using lightning: a case study*, University of Luxembourg (2014), 19–28.

[8] T. M. Murali, *Software defect reduction top 10 list*, VirginiaTech Engineering (2010), 1–43.

[9] Ajeet Kumar Pandey and Neeraj Kumar Goyal, *Early software reliability prediction: A fuzzy logic approach*, Springer Publishing Company, 2013.

[10] Tony Patton, *Determine when to use stored procedures vs. sql in the code*, TechRepublic (2005), 1.

[11] Chiel Peters, *Fors: Separating configuration from formal specification*, Master's thesis, University of Amsterdam, 2014, pp. 1–3.

[12] Sushil Ja jodia Sabrina De Capitani di Vimercati, Pierangela Samarati, *Database security**, Wiley (2002), 1–6.

[13] Pete Sawyer and Gerald Kotonya, *Software requirements*, IEEE (2001), no. 1.0, 1–2.

[14] Rob Seater and Greg Dennis, *Tutorial for alloy analyzer 4.0*, MIT (2014).

[15] Kathi Fisler Shriram Krishnamurthi, Daniel J. Dougherty and Daniel Yoo, *Alchemy: Transmuting base alloy specifications into implementations*, FSE (2008), 1–12.

[16] Walid Taha, *Domain-specific languages*, Rice University (2008), 1–3.

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.

8.1.2 Syntax

Set constants 50	
univ	The universal set
none	The empty set

Relation constants 50	
iden	The identity relation

Set operators 52		
Symbol	Name	Result
+	Union	A set
&	Intersection	
-	Difference	
in	Subset	T or F
=	Equality	

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	ˆ R
*	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 Q e
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

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 {     // Signature
  father: lone Man,       //   A declaration
  mother: lone Woman      //   Another declaration
}
sig Man extends Person {
  wife: lone Woman
}
sig Woman extends Person {
  husband: lone Man
}
```

Relationships among signatures			
S in T U in T	subset	Every S is a T, and every U is a T	An S can also be a U
S extends T U extends T	extension		An S cannot also be a U

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 example 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 quantified over the signature's set, and may use this as if it were the variable of this implied quantification.)

Multiplicity constraints in declarations

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)

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
	each element of A maps to n elements of B

Facts

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. In Alloy, a model is "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:

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.

7. 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.

8. 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.

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

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 parameters.

8.1.6 Predicates

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.

8.1.7 Facts

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.

8.1.8 Assertions

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.