Abstract

This project looks at models of a variety of systems that were written in a language called Alloy. The Alloy language is a formal specification language, based on similar language called Z. Formal specification is the maths behind software engineering and is based on the well established set theory in mathematics. While the Alloy language allows system models to be specified, the SAT solver beneath allows them to be analysed. In this project, the modelling and analysis of systems is performed in many contexts, and its usefulness evaluated.
Contents

1 Introduction ........................................... 4
   1.1 Why formalism? ...................................... 5
   1.2 Layout of the Report ................................. 5

2 Set Theory & Logic .................................. 7
   2.1 Sets .................................................. 7
   2.2 Relations .............................................. 9
       2.2.1 Example of a Binary Relation ................. 9
       2.2.2 Cross Product of Sets ......................... 10
       2.2.3 Relations as subsets of a Cross Product .... 11
   2.3 Predicates ............................................ 12
   2.4 Quantifiers ........................................... 12

3 The Alloy Language and Analyser ................. 14
   3.1 Alloy, Basic Syntax ................................ 14
       3.1.1 Signatures ..................................... 15
       3.1.2 Relations ....................................... 18
       3.1.3 Facts ........................................... 19
   3.2 Sokoban, An Introductory State-Based Model ...... 19
       3.2.1 Problem description ............................ 20
Chapter 1

Introduction

State based modelling is a rigorous mathematical practice in which systems are formally modelled with precise notation. There are many who believe the need for such formalism in software design is dire. As Gunawardena [1997] explains in his article, civil, mechanical and electrical engineers have well founded mathematics to support their designs, while software designers hardly use maths in their designs at all. To those who have indulged in formal methods\(^1\), this seems absurd. After all, “[C]omputation is inherently more mathematical than most engineering artifacts; hence, practising software engineers should be at least as familiar with the mathematical foundations of software engineering as other engineers are with the foundations of their own branches of engineering.” [Gunawardena, 1997]

Perhaps the reason for this mathematical void in software engineering is due to the fact that it is a relatively “new branch of Engineering” [Potter et al., 1992, p. 1], which has been around for only the guts of thirty years, in comparison to say construction, which has had millenia to develop [Potter et al., 1992, p. 1]. It is likely that in the years to come, we will witness rise of

\(^{1}\text{Mathematical methods of modelling.}\)
formal methods to popularity among the those in the computing world. For now, however, formalism remains a specialised area, manifesting itself here and there in small pockets of the wider software community.

1.1 Why formalism?

There are a number of reasons to choose formal specification to design software systems over the ad-hoc methods widely used today. Firstly, it is a means of “clarifying the statement of the user requirements” [Potter et al., 1992, p. 8]. When building a software system for a customer, it is good to know *exactly* what the customer wants right from the start, and avoid misunderstandings later in the project. Secondly, formal specification serves as a “precise expression of the fundamental aspects of the designer’s task.” [Potter et al., 1992, p. 8], i.e. practising formal methods constitutes good design practice.

There is a third attraction also to formalism. Modelling a system with formal methods allows it to be treated mathematically and tested for correctness properties. Mathematical models can be run through solvers that search for bugs or solutions to problems with the model. In this report, we will see the benefits of formal specification as it is used to model a collection of systems.

1.2 Layout of the Report

This report is laid out as follows:

- Chapter 2 introduces the set theory upon which formal specification is based.
• Chapter 3 introduces the Alloy modelling language, with the aid of a example model.

• Chapters 4, 5 and 6 depict the use of Alloy in three different settings.

• Chapter 7 discusses the merits and shortcomings of modelling in Alloy.

The objective of this report is to analyse the Alloy language and analyser as they are applied to a variety of problems.
Chapter 2

Set Theory & Logic

Formal logic is strongly based on mathematical set theory. Set theory is ubiquitous in all mathematical fields to some extent, but is particularly prominent in the field of formal specification. Those already familiar with set theory shouldn’t feel the need to cover this entire chapter, though it’s sections will be cross referenced in later chapters and should be consulted should any ambiguities arise therein. To the remaining readers whose knowledge of set theory is in any way hazy, this chapter offers a mathematical foundation for concepts which will arise later, and should be covered if the rest of this report is to make sense. This chapter will cover sets, relations and logic, using the universal language of maths. Later chapters on the Alloy language, chapter 3 in particular, will build on this chapter by explaining Alloy notation in terms of the mathematical notation found here.

2.1 Sets

A set is an abstract mathematical concept used to identify a collection of elements. Every set $S$ can be mathematically defined as follows:
\[ S := \{ e_1, e_2, e_3, \ldots, e_n \} \]  

(2.1)

The set \( S \) is the collection of elements \( e_1 \) up to \( e_n \). The set however need not be finite, i.e. it may contain an infinite amount of elements\(^1\). Elements can be anything: Cats, dogs, paperclips, theories, even sets. Yes, elements can themselves be sets\(^2\). Membership of an element in a set is based on the element possessing some property, or, in more logical terms, obeying some predicate, see section 2.3. These properties can be just about anything: All elements in the set are blue, all elements in the set must run on electricity, all elements of the set are standing upright etc. To define a set, one will most likely use notation such as the following:

\[ S := \{ x | P(x) \} \]  

(2.2)

In words, this reads: \( S \) is the set of all elements \( x \) such that \( P(x) \) is true. \( P(x) \) is the predicate or condition about \( x \) that determines whether \( x \) is a member of \( S \). Since \( x \) can be anything, \( P(x) \) usually contains a sub-condition that \( x \) must be an element of some already known set, call it \( T \). \( S \) is then called a subset of \( T \), a property denoted by \( S \subseteq T \). To define such a set, one would usually write:

\[ S := \{ x | x \in T \land P'(x) \} \]  

(2.3)

In this definition, part of the defining predicate is \( x \in T \), which in words reads “\( x \) is an element of \( T \)”. The rest of the predicate defines more specific conditions that the elements must obey to be in the set \( S \). If no such conditions exist, \( S = T \), i.e. they are the same set.

\(^1\)An example being \( \mathbb{R} \), the set of real numbers.

\(^2\)A property of sets that causes some interesting paradoxes not discussed here.
2.2 Relations

A relation can be thought of as a set tuples. A tuple is a listing of elements, one after another, each element belonging to some set or other. We are all familiar with tuples from coordinate geometry. For example, the geometry of the plane, also known as the set $\mathbb{R}^2$ deals with two-tuples $(x, y)$ with $x, y \in \mathbb{R}$. These two-tuples have two entries, $x$ and $y$ both of type $\mathbb{R}$. Such tuples whose entries are all numbers are more commonly known as vectors. A more general n-tuple, $t$, may be defined as a list of $n$ elements, each belonging to a certain set i.e.

$$t := (e_1, e_2, ..., e_n)$$

(2.4)

where $e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n$ and $E_1, E_2, ..., E_n$ are all sets of any type.

2.2.1 Example of a Binary Relation

Moving back to tuples in $\mathbb{R}^2$, a relation could be defined called lessThan such that the first element of every tuple in the relation is less than the second. Elements such as $(0, 1)$, $(1.8, 22)$ and $(100, 500)$ would be in the relation, whereas $(10, 3)$ for instance, wouldn’t. Clearly, lessThan is a subset of $\mathbb{R}^2$. This is because $\mathbb{R}^2$ is the set of all elements of the form $(x, y)$ with $x, y \in \mathbb{R}$ and hence it contains any relation rel whose tuples take the form $(x, y)$ with $x, y \in \mathbb{R}$.

Since lessThan is such a rel, it is a subset of $\mathbb{R}^2$. Both sets are depicted in 2.1. Obviously, due to the finite bounds of this page, the infinite sets $\mathbb{R}^2$ and lessThan cannot be fully depicted, though hopefully the diagram should be enough to represent them: $\mathbb{R}^2$ should be imagined as the plane stretching out
in all directions, while lessThan is the half-plane (coloured in red) confined to the area above the diagonal line L.

Figure 2.1: The lessThan relation.

The above relation is known as a binary relation, as each of its tuples is composed of two elements. An \( n - ary \) relation is one comprising of n-tuples, and is said to have an arity of \( n \). Since there are only three dimensions, it is hard to visualise relations with an arity higher than three.

### 2.2.2 Cross Product of Sets

Up until now, the cross product of sets has not been mentioned. The cross product of two sets \( A \) and \( B \) is denoted \( A \times B \) and is defined as:

\[
L: y = x
\]
\[ A \times B := \{(a, b) | a \in A, b \in B\} \quad (2.5) \]

In other words, the cross product of two sets \( A \) and \( B \) is the set of all tuples \((a, b)\) that can be made from the two sets, with entries \( a \) and \( b \) of its tuples coming from sets \( A \) and \( B \) respectively. A cross product can be evaluated over any number of sets \( A_1 \) up to \( A_n \) and is given by the following formula:

\[ A_1 \times A_2 \times \ldots \times A_n := \{(a_1, a_2, \ldots, a_n), a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n\} \quad (2.6) \]

Notice how the order in which the sets are multiplied matters in that the \( n^{th} \) entry in every tuple comes from the \( n^{th} \) set in the cross product.

### 2.2.3 Relations as subsets of a Cross Product

Now, the question that arises is “How is the cross product related to relations?” The answer is that every \( n \)-ary relation happens to be a subset of a cross product across \( n \) sets. Usually, a relation is defined by declaring it as a subset of a cross product such that all its elements satisfy a certain predicate. An \( n \)-ary relation \( rel \) can hence be defined as follows:

\[ rel := \{(a_1, a_2, \ldots, a_n) \in A_1 \times A_2 \times \ldots \times A_n | P(a_1, a_2, \ldots, a_n)\} \quad (2.7) \]

Notice here the addition of the predicate, \( P(a_1, a_2, \ldots, a_n) \), constraining the tuples in the relation to only those satisfying it. Let’s take a look back now at the lessThan example. The lessThan relation can be written as:

\[ lessThan := \{(x, y) \in \mathbb{R} \times \mathbb{R} | x < y\} \quad (2.8) \]
In this case, \( \mathbb{R} \times \mathbb{R} \) is the cross product of the set of real numbers with itself\(^3\), and lessThan is a subset of this cross product subject to the predicate \( P(x, y) \equiv (x < y) \) i.e. subject to the condition that \( x \) is less than \( y \).

### 2.3 Predicates

A predicate is a condition on one or more variable(s)/element(s) that evaluates to true when the variable(s) satisfy certain constraints. An example of a predicate \( P(x, y) \) over two variables \( x \) and \( y \) could be something as simple as:

\[
P(x, y) := x \in M \land y \in W
\]

where \( M \) is the set of men, \( W \) is the set of women and \( \land \) is the logical AND operator, conjoining two statements into one such that the overall statement is true iff\(^4\) both are true.

\( P \) would, in words, be the assertion that \( x \) is a man and \( y \) is a woman. Predicates can be evaluated when actual values are plugged in for their variables i.e. \( P(Arthur, Angelina) = true \) or \( P(Colm, TrinityCollege) = false \). In addition to this, predicates can be evaluated over sets of variables, a task requiring the use of quantifiers, which will be discussed in section 2.4.

### 2.4 Quantifiers

Logical statements (predicates) like \( P(X) := \) “\( x \) is bored” are ambiguous over multiple elements of a set \( X \). Does “\( x \) is bored” over \( X \) mean all \( x \) in the set \( X \) are bored? Or does it mean there is a \( x \) in the set \( X \) who is bored?

\( ^3\)\( \mathbb{R} \times \mathbb{R} \) is usually shortened to \( \mathbb{R}^2 \).

\( ^4\)If and only if.
Or does it mean that there is a general feeling of boredom among the set $X$ due to the fact that the majority of $X$ is bored? It can mean any of these, and hence quantifiers are used. A quantifier, when added to a predicate creates a new implicit predicate which takes a set of elements rather than just an element as its argument. Quantifiers specify how many elements of a set must satisfy a predicate so that the resulting implicit predicate is true. Examples of quantifiers are:

- \( \forall \): All elements of the set must satisfy the predicate so that the implicit predicate is true. The mathematical notation for this would be:

$$P'(X) := \forall x \in X | P(x)$$

(2.10)

The predicate $P$ must be true for each and every element in the set $X$ if the implicit predicate $P'$ is to be true.

- \( \exists \): The predicate must be true for one or more elements in the set if the implicit predicate is to be true. In mathematical terms:

$$P'(X) := \exists x \in X | P(x)$$

(2.11)

If any (one or more) $x$ in $X$ can be found such that $P(x) = true$, then $P'(X) = true$.

- ONE: $P' := “P$ is true for exactly one element of $X.”$

- LONE: $P' := “P$ is true for zero or one element(s) of $X.”$

The list of quantifiers could go on, just as the list of natural numbers goes on, though quantifiers other than those listed above are rarely used\(^5\).

\(^5\)At least not in this project.
Chapter 3

The Alloy Language and Analyser

3.1 Alloy, Basic Syntax

This section introduces the Alloy language and briefly covers the main keywords and syntax of the language. This section is not intended as a tutorial on the Alloy language, but rather as an aid to understanding the numerous Alloy models dispersed throughout this report. The information supplied in this section should be enough to guide the reader through the bulk of these models. Still, if any Alloy syntax appears in a model and has not been yet been covered, it will most likely be explained where it first appears. In the unlikely event that an ambiguity should arise in relation to the terminology or syntax used in a model, the reader is advised to refer to the Alloy website designed by Herberte [2009] or the book by Jackson [2006]. These sources each cover the Alloy language in depth and provide many examples of Alloy applications. Much of the information found in this section can be traced back to one or both of these sources, though some is based on experience
writing models for this project.

### 3.1.1 Signatures

In Alloy, a set is known as a signature and an element of a set is known as an atom. Atoms have no internal properties: They are the fundamental building blocks of a system. Atoms only attain meaning through the relations to which they belong and the constraints that are imposed on them. A signature is declared in Alloy as follows:

```
Listing 3.1: Declaring a Signature in Alloy.

sig Type{
    relation1 : Type1
    relation2 : Type2
    ...
}{
    //Appended fact statements go here
}
```

They keyword `sig` identifies Type as a signature of the model. After the signature declaration, there are curly braces. Within these, one may define relations from this signature to other signatures. Notice that this syntax is very similar to that of object oriented programming languages. This is intentional. Though signatures are subtly different to classes, there is a great similarity between the two. The relations from a signature to other signatures are similar to the fields or member variables within a class, which themselves may be primitives or other classes. The example above shows two binary relations from Type to Type1 and from Type to Type2. Relations need not be binary, as will be discussed in section 3.1.2. The second set of
curly braces allows factual statements to be written about the signature in question. These statements together are called an appended fact, due to the fact that they are appended to the end of the signature declaration. Facts will be discussed in more detail in section 3.1.3 but let it just be noted for now that declaring statements in appended facts can often be easier than declaring them elsewhere, due to the fact that statements appearing in an appended fact are within the scope of the signature in question. Analogous to this is the declaration of functions within a class: Member variables may be accessed without the use of an operator, such as the dot operator, since statements are resolved to the scope of the class in question.

Sub Signatures

Subsets of a signature may be declared using the keyword extends. The code example below shows an example of a top level signature\(^1\) and its sub signature. The sub signature is said in this case to extend the top level signature.

Listing 3.2: Extending a Signature

```plaintext
sig Type{
  // relations from Type
}

  // facts about Type
}
sig SubType extends Type{
  // Additional relations from SubType
```

\(^1\)A signature which isn’t a subset of any other signature.
In this example, Type is a top level signature. It is declared in the usual way, with relations in the body and facts in the appendix. A second signature, SubType is then declared extending Type. All relations with Type as their base type and all facts about Type apply to SubType i.e. every SubType is a Type or SubType ⊂ Type. Additional facts and relations can be added to SubType distinguishing it from Type, but these are only optional: the body of an extended signature is allowed to remain empty. A sub signature with an empty body and appendix can be distinguished elsewhere in the model from its super signature\(^2\) and hence is not a complete waste of time. In fact, singleton\(^3\) extending signatures in Alloy are not distinguished from atoms and often come in useful for explicitly “filling up” a set. This will be seen later in the report, particularly in section 6.2.

A keyword abstract can be placed before super signatures. All this does is enforce every atom of the super signature to be a member of some sub signature. In this example, if abstract were placed before Type it would enforce all atoms in Type to also be in SubType, since SubType is the only extending signature. It is arguable that placing the abstract keyword before a signature with no sub signatures should be illegal. However, when this is done Alloy simply ignores the abstract keyword.

\(^2\)The signature which it extends.
\(^3\)Containing only one atom.
3.1.2 Relations

While atoms can be thought of as the pieces making up a model, relations can be thought of as the glue bonding these pieces together. This section will cover the Alloy syntax for declaring relations, but does not discuss their mathematical background or notation. For those unfamiliar with relation theory, an in-depth mathematical discussion and explanation of relations takes place in section 2.2.

There are two steps to constructing a relation in Alloy. The first of these is declaring the relation in terms of the signatures across which it spans. As discussed in section 2.2, a relation is a subset of the cross product of sets. In Alloy, sets are signatures, and hence a relation in Alloy is a subset of the cross product of signatures. The cross product of signatures is the type of the relation. This is all that is specified in the declaration of a relation. Below is an example of how a relation is declared in alloy:

Listing 3.3: Declaring a Relation in Alloy.

```alloy
sig Type0{
    relation1 : Type1 -> Type2 -> Type3
}
```

This relation, called relation1 is of arity 4, i.e. its type is the cross product of four signatures, namely $Type0 \times Type1 \times Type2 \times Type3$. Notice that the first operand of this cross product, henceforth referred to as its base type, is the signature in whose body the relation is declared. The remaining operands are explicitly declared in the body of the base signature after relation1 : . Notice how they are separated using the $->$ operator. This operator is equivalent to the $\times$ operator between operands in the cross product of sets.
Alloy allows relations up to arity 6, though this arity was never reached by relations in this project’s Alloy models.

3.1.3 Facts

Constraints in Alloy are formed using constructs called facts. A fact is a body of logical statements. Each line of a fact constitutes a separate statement. Including a statement in a fact enforces it to be true for all instances of the model. Here is the syntactic structure of a fact in Alloy:

Listing 3.4: Declaring a Fact in Alloy.

```alloy
fact name{
    //Statement1
    //Statement2
    ...
}
```

3.2 Sokoban, An Introductory State-Based Model

Sokoban is a 2D maze game that demands logical thinking and problem solving abilities. It involves pushing boxes around a maze in order to get them into a certain desired configuration. The Sokoban game will be described in this section and an Alloy model of the game will follow. This model serves as an introductory Alloy model, presenting much of the syntax and convention that will be used in later models.
3.2.1 Problem description

A Sokoban board can be thought of as a rectangle divided into $w \times h$ unit squares, where $w$ is the width of the board in units, and $h$ the height. A square can be occupied by at most one object. Different types of objects exist, as listed below:

1. Player: The character controlled by the person who is playing the game. The player can move up, down, left or right, one step at a time. Each time the player moves, the move count increases. A low move count constitutes a high score.

2. Boxes: A box can be moved by the player if the player gets behind the box and pushes it.

3. Walls: Solid immovable entities that occupy the same square for the entire duration of the game.

In addition to this, special squares exist on the board called goal squares. There are as many goal squares in a sokoban game as there are boxes. The player may occupy a goal square, as may a box. The objective of Sokoban is to push every box onto a goal square\(^4\). Figure 3.1 depicts a typical Sokoban level, labelling all the various components of the game. The series of circles labelled “path of player” just show a typical path that the player would take to get to the box at the end of the path. Every step in the path is a mixture of red and yellow: Earlier steps have a low red:yellow ratio; this ratio grows as the path progresses.

\(^4\)Or, from another point of view, fill every goal square with a box.
3.2.2 Sokoban in Alloy

Static Components, Sokoban

The Sokoban board in Alloy is modelled with a set of positions:

Listing 3.5: Signature: The set of Positions.

```alloy
sig Position{
    column: Int,
    row: Int,
    coordinates: Int -> Int, // row -> column
    adjacent: set Position // All positions adjacent to this one (4 adjacency—no diagonals)
}
```

```alloy
int [column] >= 0  // Coordinates range from (0,0) to (w-1, h-1)
```
This set of positions represents the entire 2D board. The board is chopped into a discrete plane. Every discrete point on this plane, i.e. every cell/square on the board, is represented by an element of the set of positions. The appended fact imposes two constraints upon the Position. The first is that the coordinates of the position are within the bounds of the board, which are specified with the following singleton sets:

```plaintext
Listing 3.6: Width and Height sigs, serving as constants.

//Maximum width of a position
one sig Width{
  w: Int
}
  int [w] = 4
}

//Maximum height of a position
one sig Height{
  h: Int
}
  int [h] = 4
}
```
The second constraint in the appended fact just enforces that the coordinates tuple represents the point in the discrete Cartesian plane given by (column, row). A further constraint on positions ensures that no two positions are allowed have the same coordinates, i.e. a set of coordinates determines a position:

Listing 3.7: Fact: No position Duplicated.

```prolog
fact noDuplicatePositions{
    all p1, p2: Position |
    (p1 != p2) => (p1.coordinates != p2.coordinates)
    // coordinates is now a one to one 
    mapping from Position to a subset of Int->Int
}
```

An linear ordering is imposed on the positions similar to the row major ordering used to encode matrices in linear memory, only with gaps. The following code sets up an ordering on the atoms of Position and ensures that this ordering is correct:

Listing 3.8: Ordering the set of Positions.

```prolog
open util/ordering[Position] as position // Sets up the ordering

.............

.............

/* Positions are ordered along rows from left to right. When the end of a row is reached the next successive position is the leftmost column in the next existing row. This is similar to the wraparound representation of a */
matrix, only there may be gaps between positions. For example, it is possible that (3,0) succeeds (1,5) whereas with a matrix the successor to (1,5) would be either (1,6) or (2,0), depending on the row size.

```plaintext
/*
  fact positionOrder{
    all p : Position, p' : p.next{
      ((p'.row == p.row) && (p.column < p'.column)
      ) || (p.row < p'.row)
    }
  }
*/
```

A position p will have at most four positions adjacent to it. Adjacent positions on the board can be thought of as squares sharing a side. Hence four is the limiting factor on the number of positions in the adjacent set. In more mathematical terms:

\[(x, y).\text{adjacent} = \{(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)\}\]  \hspace{1cm} (3.1)

where \((x, y)\) is a position and \text{adjacent} is the relation from position to a set of positions. Note that the adjacent sets of squares like corner squares and edge squares will have less than four elements in them.

Listing 3.9: Fact: Adjacency.

```plaintext
/*Adjacent positions to p are either in the same row as p and in an immediate neighbouring column, or they are in the same column and in an immediate neighbouring row. Here, by "immediate neighbour to p" is meant "a row/column whose Int value is the row/column of p +/- 1".
```
Dynamic Components, Sokoban

Before introducing a State signature to the model, let’s define some status signatures:

```plaintext
abstract sig Status{}

one sig Won, Play extends Status{}
```

Here, the status of the game is assumed to be binary i.e. it can be either won, or in the process of being played. Hence, there are two singleton subsets of Status. A third subset, Dead, could have been added to further extend Status. A dead status means that the game being played can no longer
be won no matter what moves are made. However, it is generally hard to
determine when a game is dead\textsuperscript{5} and hence all states of the game that are not
Won are grouped under the Play status. Now that Status has been covered,
we’ll move on to the State signature.

Listing 3.10: State of Sokoban.

\begin{verbatim}
sig State{
    walls: set Position,
    boxes: set Position,
    goals: set Position,
    player: Position,
    status: Status
}

    walls&(boxes + goals + player) = none //Nothing can exist in a wall
    player&(boxes) = none //The player can’t exist within a box
    #boxes = #goals //Number of boxes must be = number of goals
\end{verbatim}

The state of a sokoban game determines the position of the player and the
boxes, both of which are dynamic properties of the game. It also determines
the status of the game, which is a derived dynamic property\textsuperscript{6} The position
of the player and the positions of the boxes are captured by relations on to
subsets of Position, while the status of the game is represented by a relation
on to one of the two members of Status. The facts appended to State ensure

\textsuperscript{5}With the exception of some simple cases e.g. when a box is pushed into a corner.
\textsuperscript{6}A property that depends on entirely on other dynamic properties, in this case the
positions of all the boxes.

26
that no objects co-exist in the same place as well as enforcing the number of boxes to be equal to the number of goals. The latter constraint is a sanity constraint that eliminates the possibility of games that can never be won due to an unequal number of boxes and goals.

The positions of the walls and goal squares are also defined in State as relations on to subsets of Position. In retrospect, these latter relations were not entirely necessary, as the positions of the walls and the goals are static, and hence remain the same over all states. If the model was to be re-written, these would be removed as relations from State to Position, and simply declared as signatures themselves extending Position:

Listing 3.11: An alternate declaration of walls & goals.

```alloy
sig walls, goals extends Position{}
```

States are meant to capture temporal behaviour of the system, and are thus ordered using the ordering utility offered by Alloy. The ordered set of states can now be treated as a sequence: The successor to a state \( s \) can be accessed with the predefined relation \( s.next \) as can the predecessor with \( s.prev \).

Listing 3.12: Ordering the States.

```alloy
open util/ordering[State] as state
```

We are now in a position to add the transitional constraints to the system. These are added with the aid of predicates. The predicates act like functions taking two parameters, both of type State, and returning a value of true or false. For more information on predicates, see section 2.3. The two state parameters here represent initial and final states and the bodies of the predicates are written to treat them as such. Predicates are not facts, but can be called within a fact statement and forced to be true given the pa-
rameters passed to them in the fact statement. It should become clear soon why predicates are a good choice for conveying dynamic behaviour. The first of these predicates we’ll examine are the invariance constraints on the walls and the goals, ensuring that both stay put:

Listing 3.13: Invariance constraints of walls & goals.

```
pred GoalInvariance [s , s' : State] {
    s.goals = s'.goals
}
pred WallInvariance [s , s' : State] {
    s.walls = s'.walls
}
```

The next constraint covers box pushing and player movement. It will be proved later that, for regular boards\(^7\), the conditions under which a box moves are that the player takes the old place of the box and that the new position of the player is equal to the old adjacent set of the player intersected with the new adjacent set of the box\(^8\). Also, a condition for player movement is that the player always ends up in one of its adjacent squares in the next state. Below is a declaration of these conditions as a predicate between States:

Listing 3.14: Box pushing and player movement constraints.

```
pred AdjacentMovement [s , s' : State] {

```

\(^7\)Boards that are rectangular in shape.

\(^8\)Old and new are here referring to the initial and final states respectively.
let \( p' = s'.player \), \( p = s.player \), \( B = s.boxes \), \( B' = s'.boxes \)

\[
\begin{align*}
&\text{\( p' \in p.adjacent \)} \quad \text{//The player must move into an adjacent square} \\
&\text{(\( p' \not\in B \)) } \Rightarrow \ (B = B') \quad \text{//If player doesn’t move into a box, boxes remain unchanged} \\
&\text{(\( p' \in B \)) } \Rightarrow \ \text{some } b' : \text{Position} \{ \text{//Else, boxes change according to rule} \\
&\quad b' \not\in B \\
&\quad p' = (p.adjacent \& b'.adjacent) \\
&\quad B' = B - p' + b' 
\end{align*}
\]

Bringing all these predicates together, we have one compact fact containing all the dynamic constraints. This fact calls on the three previously declared predicates, quantifying\(^9\) over every state \( s \) and its successor \( s.next \).

Listing 3.15: Valid Transitions Fact.

```prolog
fact ValidTransitions{ 
    all s : State, s' : s.next | 
    AdjacentMovement[s, s'] and 
    WallInvariance[s, s'] and 
    GoalInvariance[s, s'] 
}
```

\(^9\)For more on quantifiers, see section 2.4.
3.2.3 Solving Simple Sokoban

In this section, a simple Sokoban game is solved using this Sokoban model. The game is simple in that it consists of only sixteen positions and three boxes. The game is set up by specifying the positions of the player and the boxes in the initial state as a fact. The Alloy analyser is then asked, through a predicate, to find an instance of the model such that the final state in the sequence has $status = won$. In other words, Alloy is asked to use its SAT solver to solve a simple game of Sokoban.

The initial conditions of the game are declared with the following fact:

```alloy
fact InitGame{
  state/first.player.coordinates = Int[0] -> Int[1]
  state/first.goals&state/first.boxes = none  // Make sure none of the boxes are already on goals
}
```

The coordinates of the boxes and the player are arbitrarily chosen here. Notice how they are initialised using a logical sum. Every tuple in the sum is separated by the $+$ operator and the result of the sum is a set of all its operands. The $+$ operator is equivalent to the Union operator $\cup$ in set theory. Notice also the use of the $->$ operator. This operator does the same job as the comma between two values in a tuple e.g. the comma separating $x$ and $y$ in $(x, y)$. To avoid a trivial case in which the game is already solved in the initial state, a statement is added to the fact ensuring the set of boxes and the set of goals goals are disjoint in the initial state i.e. that no boxes are on
goals to begin with.

3.2.4 Conclusions, Sokoban

This project’s alloy model of sokoban was intended to be as general as possible. The model is not bound by any board size, any number of boxes, or any number of walls. It’s analysis, however, is bounded, due to the fact that board sizes above about 16 unit squares tend to be unfeasible to analyse in terms of both time and memory. For this reason, the model does not aid much in the solving of difficult Sokoban levels\textsuperscript{10}. What the model does achieve is the expression of the rules of Sokoban in a formal manner, and its analysis has uncovered an interesting fact about box pushing.

Box pushing Proof

Before going into the maths of this proof, there is a major assumption that must be stated, which is that every reachable position has a full adjacent set. A reachable position is one which the player or a box can occupy. A full adjacent set means an adjacent set of exactly four positions. This assumption is not very constricting. Say a Sokoban game exists in which there are some reachable positions without full adjacent sets. By padding out the adjacent sets of such positions with walls, such that the adjacent sets in question become full, our assumption becomes valid while the Sokoban game remains wholly unchanged\textsuperscript{11}.

Now let’s get to the notation. A box push happens across two states: An

\textsuperscript{10}i.e. those with a large state space.

\textsuperscript{11}Unchanged in the sense that, where there was once an unreachable nothing, we’re putting unreachable walls, and hence aren’t changing the space in which the boxes and player can move about.
The initial state and a final state. The initial and final positions of the player are given by \( p \) and \( p' \) respectively. The set of boxes in the initial state is given by \( B \) while the set in the final state is given by \( B' \). The relation \( \text{adj} \) gives the set of positions adjacent to a position. Figure 3.2 depicts a box being pushed by the player by showing an initial state and a final state.

The box pushing constraints are as follows:

\[
\exists b' \implies B' = B - p' + b' \\
\forall b' \implies B' = B
\]

(3.2) \hspace{1cm} (3.3)

where \( b' \) is the new position of the box that has just been pushed, if it exists. This can be calculated as follows:

\[ b' \]
\[ b' = 2p' - p \] (3.4)

Finally, a constraint is added to remove the possibility of a box being pushed into another:

\[ \#B' = \#B \] (3.5)

It turns out that there was an equivalent way to express the constraint \( b' = 2p' - p \) that was more suited to Alloy’s set notation:

\[(p' = p.\text{adj} \cap b'.\text{adj}) \equiv b' = 2p' - p \] (3.6)

This has been proven, though the proof is too long to list here. Let it just be noted that it was the Alloy model that inspired this proof.

### 3.3 More Alloy Syntax

#### 3.3.1 Quantifiers in Relations

A relation \( r : A \rightarrow B \) defines a set of tuples \( a \rightarrow b \). (The mathematical notation for tuples is \((a, b)\)). Say A and B are signatures, then \( a \rightarrow b \) is of type \( A \rightarrow B \). Quantifiers can be put on either side of a relation to restrict it somewhat. \( r : A \rightarrow \text{one} B \) for example says that exactly one \( a \) of type A is mapped by the relation onto every \( b \) of type B and one \( b \) of type B is mapped onto every \( a \) of type A. \( r : A \rightarrow \text{one} B \) would mean that every \( a \) in A would map onto one and only one \( b \) of B. This is how functions behave: Each element in the domain maps to only one element in the range. However, elements in the range may be mapped by one element of A, multiple elements of A or even no element of A. For example, say you had already
defined signatures Colour and Dog, and you want a relation dogColour : Dog → one Colour. The one here is a quantifier saying that, for every d of type Dog, there is exactly one tuple, no more, no less, in dogColour. In other words, no two tuples in the relation share their first atom. Since every dog is only represented by one tuple, it only has only one colour associated with it and not a set of colours. Note that multiple dogs may still be mapped to the same colour.

3.3.2 Quantifiers and Trivial Falsity

You have to be careful when adding such quantifiers that you don’t trivially falsify your model. An example of how you could do this is if you declared a one to one mapping between sets of different sizes. Each element in the larger set, by the quantifier, would have to map to exactly one element of the smaller set. This would result in elements of the larger set “sharing” elements of the smaller set i.e. it would result in elements of the smaller set being mapped by more than one element of the larger set, which would contradict the one to oneness of the relation and trivially falsify the model. Similarly, if you were to begin constraining the relation from the side of the smaller set, you would get one element in the larger set mapping to each one in the smaller set, but you would end up leaving atoms of the larger set unmapped. Either way, you would end up with a fallacy. Say for example you were mapping owner one -¿ one car. If there were more cars than owners, either some cars would be left unmapped or they would be owned by the same person, both of which go against the ONE quantifier on the LHS, and hence trivially falsify the model. Similarly, if there were more owners than cars, either some set of owners would be sharing a car (God forbid!) or some would not be mapped to a car at all, in which case they wouldn’t be owners.
at all! This situation would also trivially falsify the model. The rule is, one to one mappings must have a domain and range of equal cardinality i.e. the sets in the mapping must be the same size. Other quantifiers like SET and LONE may also be used, but with care so as not to trivially falsify the model.

3.3.3 Scope

Scope of a solution: An alloy model does not directly produce the SAT equation. A scope must first be specified for each object in the model. The scope of an object is a specification of the number of instances of that object that are to appear in the final solution. When given a scope, Alloy will check instances of the model up to and including that scope, but you may instruct the solver to check only instances of exactly that scope.

Small Scope Hypothesis

Say you were modelling a railroad system. You might want to check that no two trains are at the same platform at a given time. A number of signature declarations and constraints would not give Alloy enough information to test this property. In other words, it would be impossible for Alloy to test the property for any number of trains, stations and platforms (assuming these are all signatures of the model). Alloy requires you also to specify a scope in which to test the property. You might test this one for five trains, eight platforms and two stations. This will give the SAT solver behind Alloy a finite state space to test, since there is only a finite number of ways to combine five trains, eight platforms and two stations.

Due to the finiteness of the check, Alloy will never be able to prove an invariant\textsuperscript{12} such as this one for any size system. However, the developers of

\textsuperscript{12}A property that always holds.
Alloy have discovered time and time again that many errors manifest themselves within surprisingly small scopes (usually scopes less than 6). Based purely on this empirical evidence, these developers have come up with the term “Small Scope Hypothesis”, which suggests that most errors in a design can be found within a certain scope [Jackson, 2006, pp. 15, 144].

Smallest-scope solution

The smallest scope of a solution of a model is the minimum scope needed to reach that solution. Predicates or assertions run with a scope below the smallest-scope will not yield any solution. Once a solution to an Alloy model is found within a large scope, it may be the case that you want to see how small you can make the scope while still maintaining a solution. An example of this would be solution to a sokoban game that takes 15 steps. You may want to explore whether there is a solution which takes less steps, since the aim of the game is to solve a level in as few steps as possible. The mathematical bisection method may be applied here to reduce the scope of a model until you have found the smallest-scope solution. The method basically works by getting a lower bound for the smallest scope, i.e. a value that definitely does not give a solution, and an upper bound, i.e. a value that definitely gives a solution. It then keeps halving the interval bounded by the upper bound and the lower bound until it converges to a value. The method in this case looks like this:

1. Find a scope $n$ that yields a solution$^{13}$. This is the upper bound:
   \[ upper < -n. \]

2. A scope of size 0 definitely doesn’t give a solution, hence we have the

$^{13}$This requires an educated guess.
lower bound: \( \text{lower} < -0 \).

3. Find the mid-value scope \( p = (\text{lower} + \text{upper})/2 \).

4. If \( p \) gives a solution, set it as the upper bound, else set it as the lower bound.

5. If the \( \text{upper} - \text{lower} \) is less than 2, then we’re finished and the smallest scope is \( \text{upper} \). Otherwise, keep refining the interval.

When that loop is finished, the upper bound will be the smallest scope. The immediate value below it is the lower bound and hence does not give a solution, and since there are no values between this and the upper bound, the upper bound must be the smallest scope. Those familiar with search algorithms will notice that this algorithm is more or less the binary chop algorithm, reducing the search space of the scope in logarithmic time.
Chapter 4

Formally Modelling Software

Formal modelling languages like Z and Alloy can be used to model many types of systems, as they are based on the abstract theory of sets and logic. However, they really come into their own when used to describe software systems, which are fundamentally logical by nature. The reason why software systems are so ridden with bugs is that they become exponentially, if not factorially, more complex to analyse as they grow. Take for example a medium sized program that makes 1000 binary decisions. The task of exhaustively checking the state space of such a program would require the execution of $2^{1000}$ test cases, rendering it “simply impractical” [Potter et al., 1992, p. 4]. Hence it is only feasible to rigorously test small software systems. In the words of Tony Hoare, “One way is to make it [software] so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.” [Hoare, 1981]

The aim of a formal software model is to abstract the software to a simpler form and hence allow it to be rigorously checked for deficiencies “In isolation from the details of implementation” [Jackson, 2006]. Distinguishing the abstractions from these “details of implementation” can prove to be the most
testing part of devising such a model. If too much detail is included, a model becomes hardly less complex than the software it aims to simplify. On the other hand, failure to include important detail will most likely result in major design errors slipping past the conceptual stage, where they should be caught, and manifesting themselves while the software is right in the middle of being coded. It is entirely up to the person writing the model to make the right abstraction decisions about the software. These decisions should be based on:

- Experience- Looking at previous models of systems and comparing them to the current system that is to be modelled.

- Intuition- The person modelling the software should use some intuition when making decisions, e.g. The colour of the menu bar of a window usually needn’t be included in a specification.

- Context- What does the system do? Are there any safety critical parts of the system? If so, what are these parts? etc.

### 4.1 A Distributed System Design using Alloy

Sometimes, Alloy models are useful purely for clarifying any aspects of a design that are in any way vague or unclear. This section describes how Alloy was used as a design aid in the design of a distributed file system, which will henceforth be referred to as a DFS. A DFS is a file system which handles an abundance of data spread over a number of machines. It offers two main advantages over a non distributed file system. These are:

1. Unlimited storage- Theoretically, there is no bound on the number of
machines one can add to a DFS for storage\footnote{Of course, in the real world there will always be limiting factors such as cost, space etc.}. In the non-distributed case, the storage would most likely be limited to the hard drive of the host machine.

2. Remote Access- Users of a DFS need not be in its physical vicinity. In fact, a web based DFS could be accessed from anywhere in the world. Compare this to the non-distributed case, where the user must be at the host machine.

The following subsections are arranged as follows: Section 4.1.1 briefly explains what the DFS is supposed to do; section 4.1.2 is a breakdown of the Alloy model of the DFS; section 4.1.3 discusses the merits and shortcomings of the alloy model.

4.1.1 Features of the DFS

The following features are taken from the TCD 4D1 Distributed Systems project specification, 2008-2009. Only a select number of the listed listed in the specification were chosen to be implemented as part of the final system and hence it was only these that were included in the Alloy model. Explained here are those features which were implemented.

**Transparent File Access**

Transparency is an important aspect of a distributed system. It allows users to access the system without having to worry about the details of its implementation e.g. the distribution of its files over a group of machines or the replication of files. Transparent file access in the DFS allows a user to browse
through the files as if they all reside in the one location, organised by a data structure such as a tree. In other words, transparent file access is achieved by an interface between the logical structure of the system presented to the user and the low level structure of the system comprising all the physical nodes in the system on which the files are distributed, as well as nodes used to manage these files in the background.

**Replication**

Replication is a feature of the system which allows multiple copies of the same file to coexist, on different remote machines. The intention of replication is to ensure the secure storage of files: With replication, if one storage node crashes, all files on that node will still exist somewhere else.

**Directory Service**

A directory service is the logical-physical interface of the system that achieves transparent file access. Hence the two are closely intertwined. A directory service builds on top of the scattered files of the system a complete view of the files, organised under the one structure. In essence, the directory service provides a mapping from the logical location of the files to their physical whereabouts.

### 4.1.2 The DFS in Alloy

Before designing the Alloy model for the DFS, a rough design of the system had already been conceived. The design was vague and far from sufficient in terms of a guide for coding, but it presented a basic idea: All features of the DFS were to be implemented as objects. Implementing features of a distributed system as objects is fundamental to the theory of distribution:
Each object runs on a separate machine and does one job only, contributing to the overall functioning of the system. This Alloy model clarifies what objects are needed to implement the various features of the DFS, and, more importantly, how they are related; the latter having been more unclear than the former in the original conception of the system.

Figure 4.1: Single State of a DFS instance.

**Static Components, DFS**

The first objects modelled in the DFS are those making up the logical file system tree. The tree comprises directories and files. Directories can contain other directories or files, but files can only contain data: They are the leaf nodes of the tree. Thinking about the logical organisation of the system first seems sensible: The physical implementation of this logical structure can
come later.

Listing 4.1: The logical File Tree Objects.

```java
abstract sig FSObject{} // An object in the distributed FS (FileSystem)
sig File extends FSObject{} // End of a branch of the FS tree
sig Dir extends FSObject{} // Branch point
one sig Root extends Dir{} // Initial branch point
sig LocalId{} // Local Identifier of File. Unique only on a given machine
sig GlobalId{} // Global File Identifier, unique logical location within the entire filesystem. Note: There may still be replicas of this file.
```

Each of the features of the system was implemented by what is termed a virtual machine. A virtual machine is more or less a process running independently of other processes but interacting with others to form the system. Below are four signatures representing the virtual machines that implement various features of the system:

Listing 4.2: Virtual Machines in the System.

```java
// Machine hosting Physical Files
sig Server{}

// Machine Accessing Files
sig Client{}
```
Dynamic Components, DFS

Now, how each one of the machines in the DFS was to work took a bit of thinking. To allow the system to be as flexible as possible, these machines could not be statically related to one another in any way, and hence all relations between machines were modelled as dynamic. These and other dynamic properties of the system were modelled through a DFS signature:

Listing 4.3: The DFS State signature.

```plaintext
/* Represents the state of the DFS. Changes in the state of the DFS are modelled as mappings from one DFS to another */
sig DFS{
    // Objects in the system
    client : set Client,
}
```
server : set Server,
director : set Director,
file : set File,
dir : set Dir,
fsObject : set FSObject,

// Relations in the system

// Directory mappings
identity : director \rightarrow file \rightarrow GlobalId,
  // Identity of each File in the file tree as a GlobalId. Mapped by director.
child : director \rightarrow dir \rightarrow set fsObject, // Set of children of each directory.

// Replicator mappings
location : Replicator \rightarrow GlobalId \rightarrow UniqueId,

// Client properties
clientDirector : client \rightarrow one director, // The directory service each client talks to. Each client talks to only one director.
currDir : client \rightarrow lone dir, // The current directory path of each client. Just a reference (String).
currFile : client \rightarrow lone file, // The file open by the client. Just a string or an abstract reference.
//Storage: Unique Ids are mapped to sequences of data

contents : UniqueId -> Int -> Data

}\

//Object facts

fsObject = file + dir

//Unique mapping

all u : UniqueId | lone location.u

//Director mapping facts

//all d : director | d.identity //A director never maps more than one File to a GlobalId

all d : director | d.identity . GlobalId in dir.(d. child) //All files given a GlobalId by a director are mapped by the child realtion of that director.

//Actual Storage

all u : UniqueId, i : Int | lone i.(u.contents) //Each index in the contents maps to only one Data block

all u : UniqueId, i : Int | (i in (u.contents . Data)

) => ((i<#(u.contents)) & (i>=0)) //indices form a natural sequence i.e. 1,2,3...

}
An atom of DFS represents the state of the system at some time. In addition to this, DFS represents an object in the system that ties all other objects together. The DFS signature is the key to the structure & operation of the system: All objects register themselves with a central object and then use this object to find out where other objects are. An alternative to this would have been to have every object register itself with every other object with which it would be communicating. However, the centralised server design was a lot more attractive due to its simplicity and hence it was this one that was adopted.

Now, let’s look at how the relations in DFS give meaning to the various virtual machines:

- **Director**: What this must do is provide an interface between the DFS clients and the other machines in the DFS. On the client side, this interface must present the DFS in the form of a logical tree, while in background it flattens the pathnames of objects in the tree to identifiers that can be used easily by the other virtual machines. These identifiers are represented by the *GlobalId* signature, which has been declared in listing 4.1.

- **Client**: The client process browses through the files of the DFS: opening, closing, reading and writing files as it pleases. The set *client* in the DFS is the set of all clients registered with the central DFS machine. In this model, *Client* atoms may exist outside this *client set*, i.e. outside the DFS. These outsider atoms symbolise the client processes that may be running but have not yet joined the DFS. A predicate called *addClient* was written for this model, in which one such process joins the DFS, though it is not listed here. It turns out that similar
predicates could have been written apply to other objects, though one gets the general idea across: objects can dynamically join the system.

- **Server**: A set of servers exists in the DFS. It is on these servers that the physical data making up files is stored. This physical data is modelled using the *Data* signature:

  ```
  sig Data{}
  one sig zeroData extends Data{} //Same as NULL value
  in programming
  ```

  This idea of creating a data signature was taken from an Alloy model of a flash file system [Kang and Jackson, 2008]. A server stores a file at a local address as a sequence of this data. The server “knows” nothing of how the file fits into the bigger picture: This knowledge is held by the director and the Replicators, which will be discussed next.

- **The Replicators** serve as a link between the directory service and the file servers hosting the physical files. Without replication, it would have been fine just to let the directory service link pathnames directly to physical addresses. However, replication complicates things a little, in that each file in the system could have multiple physical addresses. Hence, the director converts to each pathname to globalID, a sort of middle ground between the tree structure presented to the user of the system and the raw data at the back end. The replicators close the gap between the director and the servers. They keep track of the multiple replicas of files, identified by globalIDs, on spread across various servers. In terms of this model, replicators provide a mapping from
each GlobalID to a set of UniqueID. The signature UniqueID has not yet been mentioned. It is basically an object consisting of a server ID and a local ID on that server. Together, these two IDs represent a unique physical storage slot in the system.

Listing 4.5: The set of Cubes.

```alloy
/* A unique physical location in the filesystem */
sig UniqueId{
    mac : Server,
    id : LocalId
}
```

- Physical data in the system is modelled with a relation contents from UniqueID to a sequence of Data. The data stored at a UniqueID can change with time, hence why it is modelled as part of the DFS. The sequence is constructed by inserting an Int into the cross product type of the contents relation. Without this int, the data would just be a set, which is unordered, i.e. the data in the file would have no address or place in the file. In fact, if the data elements are imagined to be just ones and zeros, then without order a file would make no sense whatsoever: All files would most likely just contain the same set of data: \{1,0\}.

### 4.1.3 Conclusions about the Alloy model

Modelling a DFS in Alloy was an exercise in design. It was an example of how Alloy could be used solely as a design aid. The SAT solver was not used here to prove a property or to solve a problem, but rather to generate
instances of the model as it was being designed. These instances could then be visualised and design flaws spotted. Perhaps the greatest achievement of this model was the clarification of the relationships between different virtual machines in the system: In the original rough conception of the system these had been wholly unclear. In conclusion, modelling this DFS in Alloy revealed Alloy as a design tool.
Chapter 5

Formally Modelling Hardware

It may be argued that the correctness and thoroughness of a hardware design is even more crucial than that of a software design. The basis for this argument [Potter et al., 1992, pp. 2-3] is that software can be easily corrected with periodic updates after it’s been sold, whereas physical devices such as hardware devices, once manufactured, cannot be so simply amended on the fly and are hence “seldom enhanced in service” [Potter et al., 1992, p.2]. For this reason, the major flaws of any physical device must be purged at the design stage.

Design is a recursive process and often entails building many prototypes, each successive prototype improving on its predecessor, until an acceptable end-product has been achieved. Certainly, finding a mistake in a prototype is expected, and far less severe than finding one in a circulating product. However, each physical prototype requires substantial resources to build. Hence it is desirable to catch as many errors as possible at an even earlier stage and thus reduce the number of physical prototypes needed to reach the finished product. This is where formal modelling comes in useful.

A formal model of a hardware device can be thought of as a “virtual
prototype”, a phrase coined by Chandrashekar et al. [2008]. The model, if written correctly, should capture a certain level of detail of the hardware and all errors within this level of detail can then be caught and removed before any physical prototype is built. Certain properties, e.g. reactions to temperature/pressure, of the device must be tested by means other than formal methods, but most logical interactions between device sub-components can be formally specified and tested. In short, a formal specification of hardware in Alloy allows you to play around with its design quickly and easily, and test for errors as you go, until reach a final design that meets the design specifications and is logically sound.

5.1 Modelling a Cache in Alloy

In this project, a hardware system known as a cache was modelled. A cache is a bank of memory that is physically close to the CPU of a computer. The aim of the cache is to keep frequently/recently used data close to the CPU so that it can be retrieved faster than data stored in main memory. If the reader is already familiar with the operating principles of the cache, the next section may be skipped. Otherwise, let it serve as a basis for the Alloy model which will follow.

5.1.1 Basic Operating Principles of a Cache

A cache is a memory subsystem which stores replicated data and allows it to be fetched more quickly than the data it is replicating. In a computer, a cache stores copies of data chunks that are in memory in an effort to speed up memory accesses. The cache operates on the principle of spatio-temporal locality: If data in memory has been recently accessed it, along with the
(spatially local) data near it, is likely to be accessed at some (temporally local) time in the near future. When data is accessed for the first time it is loaded into the cache and thereafter is accessed more quickly from the cache until it needs to be replaced. How exactly cache data is replaced depends on the replacement policy being used.

**Replacement Policy**

There are a number of different replacement policies that differentiate caches from each other. A simple replacement policy is to randomly throw out data from the cache when new data in memory is accessed. This is a simple policy to implement and model and is fast, but sometimes it’s better to be more selective when throwing out data from the cache. A common replacement policy is to throw out the chunk of data from the cache that has been sitting around unused for the longest amount of time compared to all other chunks. This policy is called “Least Recently Used” or LRU. This policy seems to make the most sense since by temporal locality, the more recently accessed (temporally local) data is likely to be accessed again sooner (at a more temporally local time) than the less recently accessed data. Hence turfing out the least recently used data means throwing out the data that has the least probability of being accessed again. However, LRU is tricky to implement exactly and is often approximated. Replacement policy was not modelled as part of this project, though there is room for the model to be extended to include replacement policy.

**Lines**

Data is stored in the cache as lines. A line is a series of bytes. The number of bytes per line, a.k.a. the line size, varies between caches, and depends on
how much spatial locality is assumed. When a byte in memory is accessed and it is not already in the cache, the line to which it belongs is pulled into a particular slot the cache i.e. a cache line. Every time a line has to be pulled in like this a lot of time is wasted, and we say that a cache miss occurs. Further accesses to the line while it is in the cache are termed “cache hits”, and no longer entail the delay of accessing main memory. When a line is in the cache, it represents a chunk of bytes in main memory. It is given a tag that identifies what chunk of memory it represents. When data is written to the a byte in the chunk, the entire line is considered dirty i.e. differing from the chunk it represents in memory. Depending on the cache write policy, a dirty line is either written synchronously out to main memory (write-through) and hence cleaned or allowed to remain dirty for a period (write-back) before being asynchronously written out to main memory. Both policies have their advantages and disadvantages.

**Cache Organisation**

Cache Organisation is a loose term and may refer to many aspects of a cache. Here, it is to be taken as the property of the cache which determines how lines in the cache are grouped. Caches may be divided into three main classes based on their organisation: fully associative caches, direct mapped caches, and set associative caches. Figure 5.1 depicts how a line in memory would be mapped under each of the three organisations.

A fully associative cache organisation allows any line in memory to be replicated by any line in the cache. Hence, when a cache miss occurs, the line being accessed will only replace a line in the cache if the cache is full.

---

1The exception is a cache with a write-through policy: with such a policy, writes to an address have to be written through to main memory.
The direct mapped organisation is quite the opposite. With this type of organisation, each line in memory is allocated one and only one line in cache into which it may be copied. This creates a competitive environment for the lines in memory in that many lines compete for each slot in the cache. This competition in turn causes some “popular” slots to become sites of frequent cache misses and frequent line replacement, while other “unpopular” slots are rarely used.

Logically, fully associative caches are simpler than direct mapped caches, in that they have no mapping rules attached to them. Also, with less conflict misses\(^2\) they tend to outperform direct mapped caches. However, the catch with fully associative caches is that they are physically more complicated than direct mapped caches due to the large multiplexing hardware they require to parallel-check the tags of all the lines at once. This parallel-checking, aside from being complex to wire, slows down the clock speed of the computer and hence impacts on performance. Most cache organisations fall somewhere between direct mapped and fully associative. The name given to these hybrid types of cache organisation is set associative. With a set associative organisation, each line in memory is mapped to a select set of lines in the cache. A set associative cache attempts to find the right balance between the hardware simplicity and speed of a direct mapped cache and the conflict free environment of a set associative cache.

\(^2\)Misses due to competition over a slot in the cache.
5.1.2 The Cache in Alloy

Static Components, Cache

The lines in cache and memory constitute the main signatures of the model:

Listing 5.1: Declaration of two Line Signatures.

```plaintext
/*A line in actual memory*/
sig Line{}

/*A line in the cache*/
sig CacheLine{}
```

Figure 5.1: The three main classes of cache.

[Bolotoff, 2007]
Sets of lines in the cache are modelled using a signature \textit{CacheSet}. This may be a little confusing as the atoms of this signature are themselves meant to represent sets containing elements of the type \textit{CacheLine}:

\begin{verbatim}
/*A set of lines in the cache*/
sig CacheSet{}
\end{verbatim}

A member of \textit{CacheSet} represents a set of cache lines. It also has a group of lines in memory associated with it, i.e. those lines in memory mapped to it. The set of lines it contains and those in memory which are mapped to it are modelled with relations, which will be seen later in the \textit{Execution} signature.

Recall from the cache description cache hits and cache misses: A cache hit occurs when a memory address accessed is in the cache, while a miss occurs when it is not. A simple signature, \textit{Status}, has been declared to represent these two events:

\begin{verbatim}
abstract sig HitStatus{}

one sig Hit, Miss extends HitStatus{}
\end{verbatim}

**Dynamic Components, Cache**

Most of the cache model is focused around dynamic behaviour. Firstly, let’s introduce the State signature which is the central dynamic component of the system:

\begin{verbatim}
Listing 5.4: The State Signature.
\end{verbatim}
/* The state of the cache at a certain time. Incorporates the next memory line to be accessed. Also includes a hit status i.e. whether executing this State caused a hit or a miss. Note on the last State: The last State has no hitStatus as it is not considered as part of the execution. Although it has a nextLine field, this is ignored and this last State is to be considered as the State where the execution stops i.e. it is NOT part of the execution. */

sig State{
    hitStatus : lone HitStatus,    // Whether nextLine caused a hit in the next State or a miss
    lineMap : CacheLine lone -> lone Line,    // A map of what line (if any) in memory is mapped to each cache line
    nextLine : one Line    // The next Line in memory to be accessed. This will be in the cache in the next State.
}

Now, if a single cache organisation were to be tested, this State signature would be enough. A sequence of these States would simulate a sequence of memory accesses. Looking over this sequence of states, one could gather information such as the hit count of the sequence. Looking at one state in particular, one could see which lines in memory were replicated in the cache and where. However, difficulty would arise using such a model to compare different organisations, which is the aim of this model. Hence this model
allows multiple sequences of states to coexist, different sequences potentially operating under different cache organisations. The key to allowing this is the Execution signature, shown below:

Listing 5.5: The Execution Signature.

```haskell
/*An execution from the point of view of the cache i.e. a sequence of memory accesses and the state of the cache at each access. Note: The last State in the execution is the terminating State and is not considered to be part of the execution.*/
sig Execution{
    numMisses : one Int, //The number of cache misses during this execution
    associatedLines : CacheSet lone -> set Line, //Maps each CacheSet to a set of lines in memory
    associatedCacheLines : CacheSet one -> set CacheLine, //Maps each CacheSet to a set of CacheLines
    states : seq State
}

let S = states.elems{
    //All states in this execution
    all s : S{
        //The lineMap relations of states in this execution only maps lines to cacheLines in //the associated set, which is given in the associatedLines
```
relation of this execution.

all cL : CacheLine | some cL.(s.
lineMap) => cL.~
associatedCacheLines = cL.(s.
lineMap).~associatedLines

// Set of all indices of the sequence of states
whose elements' statuses were misses
let missSet = {i : Int | (states[i]).hitStatus = Miss}
{
    #missSet = numMisses
}

The Execution signature allows multiple cache organisations to be tested in parallel. An Execution consists of a few things:

- A Sequence of states, each state signifying a memory access.

- A miss count, which is given by the relation numMisses. A miss count is a way of measuring the performance of an execution: the more misses, the worse the performance.

- A relation associatedCacheLines, which maps each CacheSet to a set of lines in the cache. This is the containment relation for the cache set i.e. the relation between the set and the elements it contains. Not every cache set has lines associated with it: some are empty. This allows the
number of lines per cache set to vary across different executions i.e. it allows flexibility between executions as to how the cache is organised.

- A relation associatedLines which relates each cache set to the set of lines in memory that are associated with it. Lines in memory associated with a cache set may occupy only the cache lines contained in that cache set.

The Execution signature also has an appended fact attached. This covers the following rules:

- The first rule constrains the storage of lines in the cache so that every line from memory may be stored only in its associated cache set.

- The second rule defines how the miss count is calculated. The calculation is done by creating a temporary set, filling it up with all states whose status was a miss, and then counting the number of elements in this set using the cardinality operator #.

The dynamic behaviour of a cache system is based mainly on the coming and going of lines to and from the cache. When a line is accessed in memory that is not in the cache, it must be added to the appropriate cache set and, if that set is full, some line already in the set must be discarded to make room for the incoming line. This behaviour is captured by the following predicate:

```
Listing 5.6: Predicate: Addition of Line to Cache.

// s is the previous State and s' the next State. s'
// contains the nextLine of s in its cache.
// This predicate models the transition between one state
// and the next. Here, s' is the resulting
```
// State when nextLine of s is accessed.
pred addLine[s, s': State, e: Execution] {
    let map = s.lineMap, map' = s'.lineMap, next_line = s.nextLine, next_set = next_line.(e. associatedLines)
    let cache_lines = next_set.(e.associatedCacheLines)
        , free = cache_lines - map.Line{
            (some map.next_line) =>
            {
                (map' = map)  // If the next line is in cache, then don’t change it.
                s.hitStatus = Hit  // If the line was in cache, we have a cache hit.
            }
            (no map.next_line) =>
            {
                one c : cache_lines | map' = map ++ c -> next_line
                // Else map one cache line to next line
                (some free) => one c : free |
                map' = map + c -> next_line
                // If there is some free cache line in the set corresponding to next_line, use it
            }
        }
}
This predicate takes two state as parameters: \( s \) and \( s' \). \( s \) is intended here to be the predecessor of \( s' \). What the predicate says is that the line given by the \text{nextLine} relation in \( s \) must be added to the cache so that it is in the cache in \( s' \). Below is a fact \text{validTransitions}, which is used to enforce this predicate:

Listing 5.7: Fact: Valid Transitions.

This fact says that \text{addLine} must hold between consecutive states.

Rules, Cache

Much of the cache model is comprised of stand-alone facts. The first of these prevent lines and states from being wasted. While wasted states would not
affect the result of the model, excluding them with a constraint does tidy things up a bit. Wasted lines and cache lines are excluded from the model for the same reason, but also to keep multiple executions consistent with each other in terms of the memory and cache lines they use.

Listing 5.8: Facts preventing Wastage.

```prolog
// Every State is in some execution (Not really a necessary constraint but gets rid of useless states)
fact allStatesInExecutions
{
    Execution.states.elems = State
}

// No lines or cache lines are floating around not in a set.
fact allLinesInSets
{
    all e : Execution | CacheSet.(e.associatedLines) = Line and CacheSet.(e.associatedCacheLines) = CacheLine
}
```

The next fact forces every cache set to contain the same number of cache lines for a given execution. It also enforces the number of lines in memory associated with a cache set to be constant across all cache sets, again for a given execution.

Listing 5.9: Fact: Equal Division of Lines among Cache Sets.

```prolog
// All cache sets have the same number of lines and cache lines associated with them, within an execution.
```
fact equalsSetSize{
    all e : Execution, c, c' : CacheSet{
        // If both sets have lines associated with
        // them, constrain them having equal lines
        // and cache lines
        (c.(e.associatedLines) != none and c'.(e.
        associatedLines) != none) =>
        {
            #c.(e.associatedLines) = #c
            '.(e.associatedLines)
            // All sets have
            same number of lines
            associated with them
            #c.(e.associatedCacheLines)
            = #c'.(e.
            associatedCacheLines) //
            All sets have same
            number of cacheLines
            associated with them
        }
        // A set must either contain both lines and
        cache lines or nothing at all
        (c.(e.associatedLines) = none) <=> (c.(e.
        associatedCacheLines) = none)
        (c'.(e.associatedLines) = none) <=> (c'.(e.
        associatedCacheLines) = none)
    }
}
A Cache Competition

In general, increasing the associativity of a cache reduces the competition between frequently used lines over “popular” cache sets, and hence improves the performance\(^3\) of an execution. For this reason, one would not intuitively expect a line of a certain associativity to outperform one of higher associativity; however, in some very particular circumstances, this can happen. A simple predicate specified in this model demonstrates how a certain sequence of instruction accesses causes a direct mapped cache to outperform a two way associative cache:

Listing 5.10: Predicate: Direct Mapped Vs. Two Way.

```plaintext
// Tries to find an execution that would cause a direct
// mapped Cache to out perform a
// two way associative cache
pred directMapVersusTwoWay{
    #Execution = 2
    some e, e' : Execution, c, c' : CacheSet{
        (e != e')
        #c.(e.associatedCacheLines) = 2 // e is the
        // two way
        #c'.(e'.associatedCacheLines) = 1 // e
        ' is the direct mapped

        // The two execution share the same number of states
        and the same memory accesses
        all i : Int{
```

\(^3\)Performance here refers only to the ratio of cache hits to cache misses.
The terms of this predicate are as follows:

- There are two executions in the model.
- One execution has an associativity of 2, while the other has an associativity of 1 (direct mapped).
- Both executions contain the same sequence of instructions. Had this not been the case, the comparison would not have made sense: It would have been like comparing the lap times of two athletes running different marathons.
- The final term is that the execution of the sequence of instructions with the two way cache has a greater number of misses than that with the direct mapped cache. This term is the crucial part of the predicate asking Alloy to find a situation where a direct mapped cache outperforms a two way cache.

Within a number of seconds, Alloy finds an instance satisfying this predicate, and in doing so proves that caches of some associativity do not always outperform those of lower associativity. The instruction sequence that shows
this less than obvious result is surprisingly quite a simple one. Figures 5.2 and 5.3 are graphical representations of the layouts of the direct mapped and two way caches respectively, as per the solution to the competition predicate.

**Cache layout, Direct mapped**

Figure 5.2: Direct mapped Organisation.
Cache Layout, 2 way

Set 1

<table>
<thead>
<tr>
<th>CacheLine 0</th>
<th>Line 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheLine 1</td>
<td>Line 2</td>
</tr>
<tr>
<td></td>
<td>Line 0</td>
</tr>
<tr>
<td></td>
<td>Line 3</td>
</tr>
</tbody>
</table>

Figure 5.3: Two Way Organisation.

The instruction sequence by which the direct mapped cache outperforms the two way cache is as follows:

$$Seq = (1, 3, 2, 1, 3)$$

where \(Seq\) is the sequence of instructions and the numbers in the brackets are the numbers of the successive lines accessed.

Notice that the instruction sequence can be continued so that it forms a repeating pattern of the first three elements \((1, 3, 2, ...)\). The reason why this pattern causes the cache of higher associativity to perform badly is that just before each line is accessed, it is thrown out of the cache due to an incoming line; with the direct mapped cache, line 3 is the only line accessed in its set,
and hence never gets thrown out of the cache. Sure, the miss rate for the
direct mapped cache is $2/3$, due to lines 1 & 2 constantly replacing each other,
but it is still better than the 100% miss rate of the two way cache. In short,
what we have learnt here is that there are particular loops that can seriously
impede the performance of a cache with some associativity. However, the
occurrence of such a loop is very rare and is thus not to be worried about
too deeply. Another thing to note about this model is the scope in which
the solution was found. Only four lines in memory and two cache lines were
needed to solve the problem. This concurs with the small scope hypothesis
on which Alloy is based (section 3.3.3).
Chapter 6

A Simple Rubik’s Cube in Alloy

The power of Alloy extends beyond software and hardware Engineering. This chapter looks at the diversity of Alloy as it is used to solve an everyday puzzle, namely Rubik’s cube. The abstract nature of set notation, which underlies the Alloy language, is the key to this diversity of Alloy: sets can describe just about anything (see chapter 2).

The Rubik’s cube is a well known 3D toy puzzle, invented by a man called Erno Rubik. The toy is simply a cube, consisting of a number of smaller coloured cubes. The idea of the cube is simple: Through a series of twists and turns, one must arrange the smaller cubes so that each face of the large cube consists of one colour only. Solving the cube, however, proves to be quite challenging to all but the experts, as there “is only 1 correct answer and 43 quintillion wrong ones for Rubik’s Cube” [Bellis, 2008].
6.1 Problem Description

In this section, the Rubik’s cube puzzle is put forward. Describing the puzzle entails describing the structure of the cube and also describing its mechanics. Notice that these correspond to modelling the static and dynamic properties of the cube respectively. A description of the structure of the cube in section 6.1.1 will explain what the cube is in terms of its components, shape, and colour and will also define the solved state of the cube, i.e. the solution to the puzzle. A description of the cube’s dynamics in section 6.1.2 covers how the cube can be manipulated from one state to another.

6.1.1 Structure

To imagine a Rubik’s cube, first imagine a solid cube. Now, imagine this cube was sliced twice from top to bottom, with the blade of the cutting device oriented parallel to the left and right faces of the cube. Imagine the slices to be spaced such that they divide the cube into three equally sized slabs. Let the cube be again sliced from top to bottom, only this time with the blade parallel to the front and back faces of the cube. Again, let the slices be spaced such that they divides each slab into three equally sized columns, and the entire cube into nine equally sized columns. Finally, let the cube be sliced twice from left to right, so that every column is divided into three equally sized unit cubes. Now the entire cube consists of 27 equally sized unit cubes.

Forget about the unit cube at the centre of the main cube as it is irrelevant to the problem. The unit cubes that are important are those with at least one visible face, henceforth referred to as visible unit cubes, or VUCs. Visible unit cubes can be classed into three groups, according to their position in
the main cube:

1. Corner cubes: Located at one of the eight corners of the main cube. These have three visible faces.

2. Edge cubes: Located at any one of the twelve edges of the main cube. These have two visible faces.

3. Centre cubes: Located at the centre of any one of the six faces of the main cube. These have only one visible face.

Each face of the main cube is made up of nine VUC unit faces: 4 corner cube faces, 4 edge cube faces, and 1 centre cube face. Each of these unit faces can be one of six colours. When the Rubik’s cube is solved, each face of the main cube is composed of nine equally coloured unit faces, and no other face shares this colour. Getting the cube to this solved state from an unsolved state is the crux of the Rubik’s cube problem. Below are images of a Rubik’s cube in its solved and unsolved states. Note that there is only one solved state as per the image, whereas the unsolved state shown is one of about 43 quintillion.
Figure 6.1: An Unsolved Rubik’s cube: Each face consists of a mixture of colours.

[Garza, 2009]
6.1.2 Dynamics

To understand the dynamics of the Rubik’s cube, let’s impose upon it a three dimensional Cartesian coordinate system. Looking at the cube head on, i.e. at its “front”\(^1\), a viewer would be looking in the direction of the positive z axis. From this same viewpoint, the positive x axis would point from left to right and the positive y axis from bottom to top.

Now, imagine again the division of the cube into slabs. Each slab is the result of the cube being sliced down either the \(xy\), \(xz\) or \(yz\) planes. The mechanics of the cube allow any such slab to be rotated by \(\pm 90^\circ\) relative to

\(^1\)Although the nature of the cube is such that it has no inherent front, back, up, down, left or right, one must assign names to its sides so that it can be described in a systematic and consistent manner.
the rest of the cube. Figure 6.3 depicts such a slab that can be rotated in the $yz$ plane. Notice that the location of this slab is given by a value in $x$, i.e. $x = 0$. This is a discrete value, and in fact represents the interval $[0, 1)$ of the continuous $x$ dimension. The slab immediately to the right of this one is represented by $x = 1$ and the one to the right of that by $x = 2$, though these are not shown in the diagram. In a similar fashion, the three slabs in the $xz$ plane and the three in the $xy$ plane would be represented by $y$ and $z$ values respectively.

![Slab](image)

Figure 6.3: A slab in the $yz$ plane at $x = 0$.

### 6.2 Rubik’s Cube in Alloy

Now that the layout and mechanics of the cube have been established, let’s move on to expressing these in Alloy. However, instead of modelling a cube
of size\(^2\) 3 let’s stick with a simpler one of size 2. The mechanics are the very same for this cube as they are for the size 3 cube, only now there are only two slabs per axis. The reason why this smaller size has been adopted is that the state space of the conventional size 3 cube, comprising roughly “43 quintillion” [Bellis, 2008] states, is too vast for the Alloy analyser to explore even partially. Alloy was built for small scope analysis, see section 3.3.3; hence, attempting to analyse such a large state space using Alloy would be analogous to attempting to scrub the floor of a large hall with a tooth brush. The idea behind this model is not to attempt to solve the cube with Alloy, but rather to model its mechanics mathematically, and solve partial cube problems with reasonably-sized state spaces, e.g. “What series of moves will amount to the swapping of the bottom-left-front unit cube with the top-right-back unit cube, with all other cube positions/orientations preserved?”.

**Static Components, Rubik’s Cube**

The size 2 Rubik’s Cube in Alloy is modelled as a collection of eight mini cubes. These mini cubes, henceforth just referred to as cubes, correspond to sections of the physical cube. In the solved state of the puzzle, each cube occupies a position, which will be called its home position, given by a coordinate vector in \( \mathbb{R}^3 \). In other states, at least two cubes will be out of their home positions\(^3\). The cubes were modelled in Alloy as a set:

```java
abstract sig Cube{}
```

\(^2\)Size here refers to the length in units of any edge of the cube, i.e. a cube of size three would contain \(3^3 = 27\) unit cubes.

\(^3\)If one is out of position, then it must be in the position of another, which in turn must be out of its position.
Notice the abstract keyword is used here, enforcing atoms in *Cube* to belong to an extending signature if one or more exists. Eight singleton signatures extend Cube, corresponding to the eight cubes mentioned already. Remember from section 3.1.1 that singleton subsets are just treated as atoms in Alloy and hence extending *Cube* amounts to filling it up with elements:

```
Listing 6.2: The set of Cubes.

one sig c000, c001, c010, c011, c100, c101, c110, c111
   extends Cube{}
```

The cubes in *Cube* are named according to their home position coordinates i.e. their $\mathbb{R}^3$ coordinates in the solved state. Each cube name is of the form $c_{xyz}$ where $xyz$ corresponds to the $(x, y, z)$ coordinates of the cube in the solved state. In each dimension, a cube can be in one of two places: 0 and 1. Hence there are $2^3 = 8$ cubes.

**Dynamic Components, Rubik’s Cube**

The State of a Rubik’s Cube puzzle depends on two things:

1. The relative positions of all the cubes.
2. The relative orientations of all the cubes.

To keep things straightforward, every cube is assigned a position and orientation relative to a fixed global coordinate system. This global coordinate system is referred to as ether in the model, and corresponds to the world to which the cube belongs. Since only relative movements change the state of the puzzle, a local coordinate system could have been chosen such that one cube remained fixed while all others moved relative to it. However, this alternative seems less straightforward than the global system and hence it is the latter that is employed.
Listing 6.3: The State Signature.

```plaintext
sig State{
    s000 : one Cube,  // Relations from the host 
    // space to the guest cube
    s001 : one Cube,
    s010 : one Cube,
    s011 : one Cube,
    s100 : one Cube,
    s101 : one Cube,
    s110 : one Cube,
    s111 : one Cube,
    xAxis : Cube -> one Axis,  // Axis of Ether that hosts the cube's x axis
    yAxis : Cube -> one Axis,
    zAxis : Cube -> one Axis,
    rotation : lone Rotation  // The most recent rotation that was done to get here
}
```

Above is the State signature. It incorporates all that has been already discussed. Each of the eight home positions of cubes in the fixed coordinate system are mapped to exactly one physical cube. The cube located at the home position of a cube $c_{xyz}$ is given by the relation $s_{xyz}$. In the solved state, all $s_{xyz}$ will map to a corresponding $c_{xyz}$.

The other relations need a bit more explaining. First, let's consider the orientation of a cube. To do so, we need to assign a local coordinate system to the cube. Now, let's say that in the solved state, each cube is oriented the same as ether i.e. the axes of each cube in the solved state point in the
same directions as the corresponding axes of ether. Now, when a slab of cubes gets rotated, the axes of each cube in the slab no longer match those of ether. All the same, it follows from the fact that all slab rotations are $90^\circ$ that the each local axis of each cube will always point in the same or exact opposite direction as some axis of ether. Since there are three axes in $\mathbb{R}^3$, each axis of each cube can point in six directions: $x, y, z$ and the inverses of each i.e. $-x, -y$ and $-z$. Hence, the orientation of a local axis of a cube can be compactly expressed as one of six values. The orientation of the entire cube is then a combination of the orientations of each of its axes\(^4\).

In the State signature, the $xAxis, yAxis$ and $zAxis$ relations signify the mappings from the local $x, y$ and $z$ axes of the cubes respectively to some direction in ether. For instance, $xAxis$ signifies the mapping of the local $x$ axis of every cube to some ether axis. In the solved state, these relations map every local cube axis to the corresponding axis of ether. Again taking the $xAxis$ relation as an example, this would map all cubes to the $x$ axis of ether in the solved state. Let it be noted that the local axes are implicit in the names of the relations: They map $State->Cube->Axis$ but what is implied by each is a higher arity relation $State->Cube->Axis->Axis$. The extra $Axis$, signifying a local cube axis, does not appear in any of these relations as it is implicit in the relation name. Below is a declaration of the $Axis$ signature from the model, accompanied by a declaration of its singleton subsets/elements:

Listing 6.4: The Axis Signature.

```java
abstract sig Axis{
    xTransform : Axis,
}
```

\(^4\)In fact, to know the orientation of the cube it is sufficient to know the orientation of any two axes.
The `Axis` signature is abstract and is extended into six singleton subsets. These correspond to the three global axes and their inverses. The relations in the body of the signature are used to model what happens to an axis when the local coordinate system to which it belongs is rotated relative to some fixed coordinate system. The three possible rotations in the Rubik’s cube are about the three global axes. The three `Transform` relations in the body of `Axis` deal with rotation about the three global axes. They assume that local coordinate system is originally oriented the same as the global one, and then specify where the local Axis in question will go when a rotation takes place. These `Transform` relations are built by explicitly declaring their components:

Listing 6.5: The Transform Relations.

```prolog
fact {
    xTransform = {x -> x + x' -> x' + y + z' + y' -> z + z -> y + z' -> y'}
    yTransform = {x -> z + x' -> z' + y + y' -> y' + z -> x' + z' -> x}
    zTransform = {x -> y' + x' -> y + x + y' -> x' + z -> z + z' -> z'}
}
```
Each transform relation is an rule about where all the local axes go when rotation takes place about the global axis it represents. With every rotation, the axis of rotation remains stationary while the other axes move. This should be clear after an examination of the Transform relations. Notice again for these that, just like the Axis relations earlier, the Axis of rotation is implicit in the relation name. Had it been included explicitly in the relation, there would only need to be one relation of triples(three tuples).

Looking back on the State signature, the last relation listed in the body maps to a Rotation. This relation keeps track of what move got the system from the last state to this one. A move here simply means the rotation of a slab by 90°. Since every state records the last move made, a sequence of moves can be inferred from the state sequence of an instance. This is useful if the sequence of states constitutes a solution to some problem and it is desired to know this solution in terms of a sequence of rotations. Let’s introduce the Rotation signature now:

Listing 6.6: The Rotation Signature.

```plaintext
abstract sig Rotation{
    inverse : Rotation  //The inverse rotation to this one
}

fact{
    let l = x0 -> x0' + x1 -> x1' + y0 -> y0' + y1 -> y1' + z0 -> z0' + z1 -> z1'{
        inverse = l + ~l
    }
}
```

82
Every member of rotation represents the rotation of one of the six slabs. Imagine looking at the face of a slab in the direction of the positive axis that is perpendicular to the slab. From this perspective, the slab can be rotated either clockwise or anti-clockwise. Clockwise rotations of slabs are represented by the unprimed members of \( \text{Rotation} \), while anti-clockwise rotations are represented by the primed members. The inverse relationship between clockwise and anticlockwise rotations is captured by the \textit{inverse} relation. Notice here the use of the \texttt{operator}, which reverses the order of tuples in a relation. Arguably only one direction of rotation is needed because any \( 90^\circ \) anticlockwise rotation of a slab about its axis can be achieved by rotating it by \( 270^\circ \) in the opposite direction. However, \( 270^\circ \) rotation constitutes 3 moves as opposed to just one, leading to more drawn out solutions; thus bidirectional rotation is allowed. The rules for rotation are given as predicates between states. One such predicate will be listed here and the layout of the others can be deduced by analogy since they are very similar. Below is the predicate governing the clockwise rotation of the \( x = 0 \) slab about the \( x \) axis.

Listing 6.7: Predicate: Rotating a slab.

```plaintext
pred rotX0 [s, s’ : State] {
  s’ . s100 = s . s100  // First swap positions
  s’ . s101 = s . s101
  s’ . s110 = s . s110
  s’ . s111 = s . s111
  s’ . s000 = s . s010
}
```
The first thing this predicate does is a position swap: all positions in the slab take the place of their clockwise successor. Then the slab is selected as a subset of the entire set of cubes, given by the set $P$. Each cube in $P$ is rotated as per the rules of the x transform. This predicate, and the others similar to it, are enforced using the `validTransitions` fact:
Listing 6.8: Fact: Valid Transitions.

```haskell
// Valid transitions are x, y and z rotations. Only consider
one plane for each since all motion is
// relative and there are only two planes per axis.

fact validTransitions{
    all s : State, s' : s.next{
        // Forward rotations
        (rotX0[s, s'] and s'.rotation = x0) or
        (rotY0[s, s'] and s'.rotation = y0) or
        (rotZ0[s, s'] and s'.rotation = z0) or

        // Inverse rotations
        (rotX0[s', s] and s'.rotation = x0') or
        (rotY0[s', s] and s'.rotation = y0') or
        (rotZ0[s', s] and s'.rotation = z0')
    }

    all s : State, s' : s.next{
        noCancelledRot[s, s']
        noCancelledRot[s', s]
    }
}
```

Notice another predicate `noCancelledRot` at the end of this fact. This predicate ensures that no rotation is immediately followed by its inverse: such an occurrence would be wasteful in terms of moves, since inverse rotations cancel each other out (effect to doing nothing) when performed one after the other. Below is the predicate that rules out these wasteful occurrences:

Listing 6.9: Fact: Ruling out Cancelled Rotations.
Solving a Simple Cube problem

As already mentioned, it would be unrealistic to expect Alloy to solve a size 3 Rubik’s cube. It turns out that even a size 2 cube would be too much for Alloy to solve, unless the SAT solver got really lucky. For this reason, a more attainable goal was set for Alloy’s hidden solver to work towards. In words, the goal was: “To find a series of moves that will amount to the swapping of the bottom-left-front unit cube with the top-right-back unit cube, with all other cube positions/orientations preserved.” The reasoning behind this goal was that if one knew how to swap cubes with each other, without disturbing the rest of the puzzle, a full solution could be reached just by iterating the swapping procedure. Of course, different pairs of cubes would require different sets of moves to swap them, but the idea was to use alloy to get a hint as to how to solve swapping problems. After all, it is often useful to simplify an unmanageable problem until it can be solved and then work it back to its original complexity from there.

The specification for the swapping problem in Alloy is listed below:

```alloy
// If a rotation occurs in s, its inverse does not occur in s'
pred noCancelledRot [s, s' : State]
{
    let r = s.rotation, r' = s'.rotation, pair = r → r
    '{
        no pair & inverse
    }
}
```
Listing 6.10: Specification of the Swapping Problem.

```plaintext
fact init{
  first.s000 = c000
  first.s001 = c001
  first.s010 = c010
  first.s011 = c011
  first.s100 = c100
  first.s101 = c101
  first.s110= c110
  first.s111 = c111

  Cube.(first.xAxis) = x
  Cube.(first.yAxis) = y
  Cube.(first.zAxis) = z
}

pred test{
  last.s000 = c110
  last.s001 = c001
  last.s010 = c010
  last.s011 = c011
  last.s100 = c100
  last.s101 = c101
  last.s110= c000
  last.s111 = c111

  all p : Cube - c110 - c000  { p.(last.xAxis) = x
}```
The fact *init* constrains the initial state of the cube to be the solved state\(^5\). The predicate *test* then asks Alloy to find a State sequence that brings ends with a state in which two cubes have swapped home positions, while the rest of the cubes remain in their home positions and maintain their original orientations also. Running this predicate yielded a solution of 20 states, while the SAT solver conclusively failed to find solution in a scope of 10 states. Hence it can be concluded that the minimum scope solution lies somewhere in the interval between 11 and 20 states inclusively. Performing an approximate binary chop of this interval, the predicate was run at a scope of 15, with indeterminate results after four hours running. Due to the NP complete nature of SAT solving, there is no predictability as to how long it would take to search the state space at scope 15. Thus, for now, a solution of 20 states will have to do.

It was found that the solution yielded by the SAT solver was one of many\(^6\). The first solution that was found, written as a product of rotations, was as follows:

\[
Y_0^{-1}Z_0X_0Z_0^{-1}X_0Y_0^{-1}X_0^{-1}Z_0X_0^{-1}Z_0^{-1}Y_0^{-1}X_0^{-1}Z_0^{-1}X_0Z_0^2Y_0^{-1}Z_0^{-1}X_0 \tag{6.1}
\]

where \(Z_0^2\) is just a shorthand for writing \(Z_0Z_0\).

\(^5\)In this case, working the solution backwards from a solved to an unsolved state seemed to make sense.

\(^6\)Alloy supports stepping through multiple satisfying instances.
This is by no means a minimal solution in terms of the number of moves involved, but it is a solution nonetheless. Sometimes, when solving a problem, all one needs is a push in the right direction from Alloy’s SAT solver and the rest they can do themselves. This solution, though coarse, is at least a start for one who wishes to solve this cube problem: Rather than attempt to solve it from scratch one can take a head start and begin refining Alloy’s solution.
Chapter 7

Conclusion

In conclusion, modelling in Alloy has three benefits:

1. Clarifying a design.
2. Purging a design of its bugs.
3. Finding solutions to small scope problems.

The Alloy tool has been proven to be flexible: It could handle hardware systems and miscellaneous systems such as the Rubik’s cube. Modelling these systems yielded some interesting conclusions: Problems were solved and properties proved.

However, the Alloy tool is limited. The small scope hypothesis upon which it is based is based solely on observation and as such gives no guarantee that all errors will indeed be caught at a small scope analysis of a model. For this reason, it seems that other methods of property checking should be employed, e.g. theorem proving, when the safety of a system or subsystem is critical.

Another limitation to Alloy, particularly as a solver, is that it cannot solve very big state spaces. As is the case with any system, specialisation to
a task always improves performance but constrains flexibility. Alloy has a huge amount of flexibility in terms of the fact that almost any problem can be specified with the language and sent to the SAT solver. However, it is this very flexibility that means it is never as good as a solver designed with a particular problem in mind.

Stepping back to a more general view of formal methods, it seems that these still have a long way to go in the world of software engineering. It may be a while before the majority of software developers are using formal methods as standard practice, and not just as a specialised tool. However, there is no denying the growth of the formal methods community. It is only a matter of time before formal specification becomes so deep rooted in software engineering that people will look back and wonder how software systems were ever designed without them in the early 21st century.
Bibliography


