Translating Alloy Using Boolean Circuits

by

Samuel Isaac Daitch

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

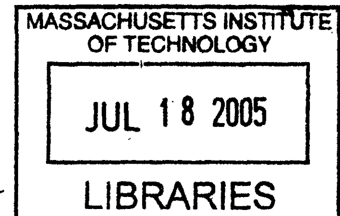
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

© Samuel Isaac Daitch, MMIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.



Author ....
Department of Electrical Engineering and Computer Science
July 1, 2004

ARCHIVES

Certified by ....
Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Translating Alloy Using Boolean Circuits

by

Samuel Isaac Daitch

Submitted to the Department of Electrical Engineering and Computer Science
on July 1, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Alloy is a automatically analyzable modelling language based on first-order logic. An Alloy model can be translated into a Boolean formula whose satisfying assignments correspond to instances in the model. Currently, the translation procedure mechanically converts each piece of the Alloy model individually into its most straightforward Boolean representation.

This thesis proposes a more efficient approach to translating Alloy models. The key is to take advantage of the fact that an Alloy model contains patterns that are used repeatedly. This makes it natural to give a model a more structured Boolean representation, namely a Boolean circuit. Reusable pieces in the model correspond to circuit components. By identifying the most frequently used components and optimizing their corresponding Boolean formulas, the size of the overall formula for the model would be reduced without significant additional work. A smaller formula would potentially decrease the time required to determine satisfiability, resulting in faster analysis overall.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

Acknowledgments

I am tremendously grateful to Prof. Daniel Jackson for his encouragement and patience during the process of writing this thesis. The abundance of advice that he has dispensed to me over the past several years has been quite helpful, and his confidence in me has been a great source of motivation.

I thank Ilya Shlyakhter for suggesting the idea that led to this thesis, and for discussing with me various issues that arose over the course of my research.

The kindness and generosity of all my friends at MIT are what have made my stay at this institute so enjoyable. I am indebted to them for being a source of comfort during stressful times.

Finally, I would like to thank my parents and my sister for their constant, unconditional love and support. It is a great blessing to have such a wonderful family. All that I have accomplished, I owe to them.

Contents

1 Introduction **13**

2 Alloy **15**

2.1 Relations 15

2.2 Operators 17

2.3 Quantifiers 18

2.4 Abstract Syntax Tree 19

3 Boolean Circuits **21**

3.1 Constants 22

3.2 Variables 22

3.3 Compound Circuits and Gates 23

3.3.1 Connector Gates 24

3.3.2 Simple Gates 24

3.3.3 Compound Gates 26

3.4 Variable Setters 28

4 Translation Strategies **31**

4.1 Basic Idea 31

4.1.1 Variables 32

4.1.2 Constant Expressions 32

4.1.3 Operators 32

4.1.4 Quantifiers 33

4.2 Optimized Gates 34
4.3 Combined Gates 36

5 Implementation **39**
5.1 Creating the Circuit 39
5.2 Simplification 40
5.3 Converting into CNF 45

6 Results **47**

7 Conclusion **55**
7.1 Future Work 55

A Standard Translations **57**
A.1 Operators 57
A.2 Constants 66
A.3 Quantifiers 66

B BLIF-MV **69**

List of Figures

2-1 An Alloy model 16

2-2 An Alloy AST 19

3-1 A compound gate 27

3-2 Desugaring a circuit containing a variable setter 29

4-1 Translating an quantified Alloy expression 34

4-2 Optimizing a gate for a specific configuration of inputs 36

4-3 Combining multiple Alloy operations into a single gate 36

4-4 Combining a gate node with two of its child nodes in the circuit . . . 38

5-1 The procedure printBlifMVTables() 42

5-2 The procedure constructGate() 44

List of Tables

6.1 Gate simplification times for individual Alloy operators 50

6.2 Gate simplification times for combined gates 51

6.3 Results for dijkstra model 52

6.4 Results for handshake model 52

6.5 Results for stable-mutex-ring model 52

Chapter 1

Introduction

The uniqueness of the Alloy modeling language is that it can be automatically analyzed. The Alloy Analyzer tool can definitively disprove a property of an Alloy specification by generating a concrete counterexample (or alternatively, it can illustrate a property by generating a concrete example). The analysis works by translating an Alloy specification into a Boolean formula whose satisfying assignments correspond to an Alloy instance that fulfills the specification.

The main computation involved in the analysis of an Alloy model is the search for satisfying assignments of the Boolean formula, which is accomplished by converting the formula to CNF form and feeding it to any of a number of off-the-shelf SAT solvers. Any technique that translates an Alloy model into a simpler Boolean formula is potentially helpful, in that it results in faster analysis, and even may enable analysis of models that were previously intractable.

This thesis describes a system for translating Alloy models using a specially designed Boolean circuit representation. This representation reflects the design of the Alloy language, in that it is separable into components that correspond to individual or small groups of Alloy operators. Just as Alloy operators are reused many times throughout an Alloy model, so too do the Boolean circuit components representing these operators reappear many times within the circuit translation of an Alloy model.

The advantage of such a system is the possibility of simplifying the Boolean representation of these frequently used circuit components, resulting in a smaller overall

Boolean translation of the Alloy model. Experiments using the MVSIS logic analysis tool to simplify components yielded moderate success in creating improved translations and demonstrate the potential of using this method.

The remainder of this thesis will be organized as follows: Chapter 2 will give an overview of the Alloy language. Chapter 3 will introduce our concept of Boolean circuits. Chapter 4 will describe the natural translation of an Alloy model into a Boolean circuit, and paradigms for creating useful circuit components. Chapter 5 will fill in the details of implementing a system that uses Boolean circuits to translate Alloy models, simplifies circuit components using the MVSIS tool, and finally produces a useful formula. Chapter 6 will present the experimental results of translating Alloy models with this system, using various implementation choices. Chapter 7 will conclude and suggests avenues for further work.

Chapter 2

Alloy

2.1 Relations

The universe of an Alloy model is organized around a set of basic types which are defined in the model. Each type is essentially a set of elements, or "atoms". Other than booleans and nonnegative integers, the value of every expression in Alloy is a relation between these basic types. Each relational expression has a specific relational type, which indicates the arity of the relation and the basic types of the atoms that may appear in each column of the relation.

Consider the model in Figure 2-1, inspired by the Simon and Garfunkel song "One Man's Ceiling is Another Man's Floor". This model describes a group of men located in rooms that have ceilings and floors. Each man is standing on exactly one floor and is directly under exactly one ceiling. The same building platform can serve as one man's ceiling and another man's floor.

There are two basic types defined in the model: Man and Platform. In addition, there are two relations defined: ceiling and floor. These are each binary relations that have type Man->Platform, and they indicate which Platform serves as the ceiling and which serves as the floor of each Man. Man itself can be used as an expression in Alloy, and it is considered to be a unary relation of type Man, containing a tuple for

```
sig Platform {}
sig Man {
   ceiling: Platform
   floor: Platform
}

fun someoneIsAtTheBottom() {
   some x:Man | x.*(ceiling.~floor) = Man
}

run someoneIsAtTheBottom for 3 Man, 4 Platform
```

Figure 2-1: An Alloy model

each Man atom that is present.

To automatically analyze a model, we write an Alloy formula that expresses conditions on the values of the relations defined, e.g. someoneIsAtTheBottom() in the above example. In this formula, the expression ceiling. ~floor has type Man->Man and relates Man a to Man b iff the ceiling of a is the floor of b. x.*(ceiling.~floor) is then the set of Man atoms reachable from x by following the ceiling.~floor relation zero or more times. Thus, our formula expresses the condition that there is some man such that the set of men zero or more floors above him is the entire set of men. The analyzer converts this Alloy formula into a Boolean formula, which is true exactly when the Alloy formula is true, and furthermore in which a satisfying Boolean assignment directly corresponds to a particular set of values for the Alloy relations in the model that satisfy the formula.

In order for an Alloy model to be convertible into a finite Boolean formula, a scope assigning a finite bound to each type must be specified. This indicates the maximum number of atoms that may belong to a basic type. In the example above, the command specifies that the scope of Man is 3. This allows us to allocate 3 Boolean variables to represent all possibilities of the value of the type Man. Each variable indicates whether one of the three potential Man atoms actually exists in the model.

Similarly, we can allocate Boolean variables to represent all possible values of each relation in the model. For example, the possible values of the relation floor are all sets of tuples of atoms with type Man->Platform. There are $3 \cdot 4 = 12$ such tuples, so we can represent floor with 12 Boolean variables, each one to indicate whether one of the tuples is contained in the relation.

We see now that a Boolean translation of the above Alloy model will contain a minimum of 31 variables: 3 to represent the possible values of Man, 4 to represent Platform, and 12 each for ceiling and floor. The Boolean formula may contain other variables as well, but only these 31 variables determine the Alloy instance that a particular satisfying assignment represents.

2.2 Operators

The Alloy language contains operators whose inputs and output are either relation-, boolean-, or integer-valued, allowing the formation of compound expressions. For example, the union operator (+) takes in two relations, and produces a new relation containing all tuples that are in either of the two input relations. As another example, the cardinality operator (#) takes a relation as input, and produces an integer indicating the number of tuples in the relation.

Alloy's static typing rules define a type for every compound relational-valued expression. For example, the expression ceiling+floor has the type Man->Platform, because that is the type of the subexpressions ceiling and floor. Thus, when translated to Boolean, the value of expression ceiling+floor can be represented by a finite number of bits, as many as are necessary to represent a relation of type Man->Platform. In particular, each of these bits is a Boolean formula whose value depends on the Boolean variables assigned to represent the basic types and relations in the model.

Similarly, every compound integer-valued expression has a maximum size. For

example, assuming Man has scope 3, the expression #Man can take on values between 0 and 3 inclusive. Thus, again, this expression can be represented by a finite number of bits, 2 bits in this case, since the integers from 0 to 3 can be represented with 2 bits. Each bit is a boolean formula containing the variables assigned to represent the type Man.

2.3 Quantifiers

Alloy also has constructions that introduce quantified variables. For example, the expression

```
some x:Man | x.*(ceiling.~floor) = Man
```

introduces the variable x and is true iff there is some atom of type Man such that if x has this atom as its value then the expression x.*(ceiling.~floor) = Man is true.

An expression containing quantified variables, such as x.*(ceiling.~floor) = Man in the above example, cannot be translated directly into Boolean formulas because the value of the expression does not depend solely on the values of the relations defined in the model; it also depends on the value of x. The quantified formula must be "grounded out" in order to give definite value to its subexpressions. In other words, if we call the possible atoms of type Man by the names Man_0, Man_1, and Man_2, then

```
some x:Man | x.*(ceiling.~floor) = Man
```

is equivalent to

```
(Man_0.*(ceiling.~floor) = Man)
  or (Man_1.*(ceiling.~floor) = Man)
  or (Man_2.*(ceiling.~floor) = Man)
```

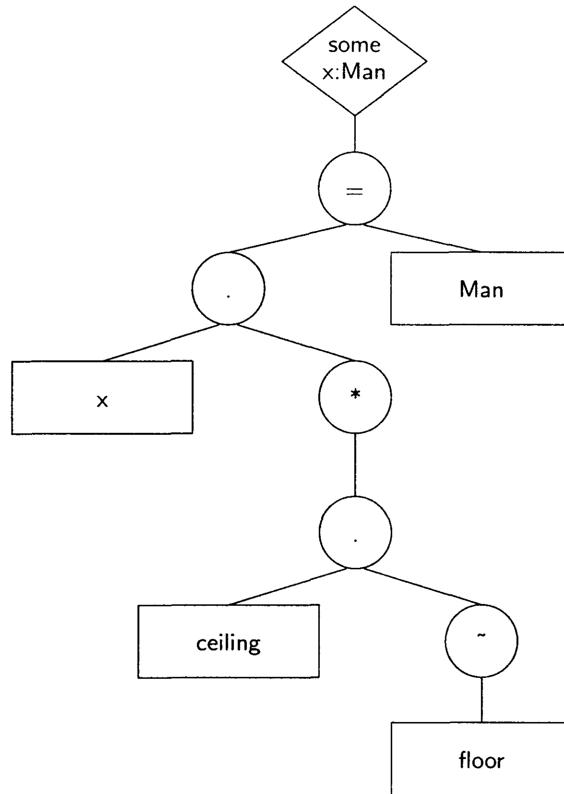


Figure 2-2: An Alloy AST

This grounded out formula can now be directly translated into a Boolean formula as described above.

2.4 Abstract Syntax Tree

An Alloy model can be parsed into an abstract syntax tree (AST) whose internal nodes are either operators or quantifiers. Every leaf is either the name of a relation declared in the model, or the name of a quantified variable introduced by one of the leaf's ancestors, or a constant-valued expression. Every node in the AST can be thought of as having either a boolean, relational, or integer value. The value of the root node of the AST is always a boolean, so that the model corresponds to single boolean formula indicating the presence (or absence) of the property described in the model. As an example, Figure 2-2 gives the AST for the formula someoneIsAtTheBottom() from the

model in Figure 2-1.

Note that in an Alloy model, the same name may be used for different quantified variables introduced in different locations in the model. We will assume that parsing the model into the AST includes giving unique identifications to every distinct variable, even if they were referred to by the same name in the model.

Chapter 3

Boolean Circuits

Let us now formalize the concept of a *Boolean circuit.* We shall model a Boolean circuit as a rooted directed acyclic graph. Each edge of the graph can be thought of as a bundle of wires, where each wire carries a single Boolean value. Thus, the edge itself holds a list of Boolean values. We will refer to the number of values in this list as the *width* of the edge.

All the nodes in the graph, with the exception of the root node, have at least one output edge, and zero or more input edges. For nodes with more than one input edge, we must specify an ordering of the input edges, so that it is possible to refer to the first input, second input, and so on.

Each node has a list of Booleans as its output value, which is a function of the values from the input edges. This output value is carried by all of the node's output edges. Thus, all output edges from a particular node must have the same width, as determined by the node.

The output value of the root node is considered to be the output value of the entire circuit.

Note that each node in the graph defines a Boolean sub-circuit, which consists of the subgraph rooted at that node.

The leaf nodes are themselves Boolean circuits. These simple circuits come in one of two types: *constants* and *variables.*

3.1 Constants

A *constant-valued* node always emits the same list of Boolean values. We use the notation

$$\texttt{CONSTANT}\langle b_0, ..., b_{n-1}\rangle$$

where $b_i \in \{TRUE, FALSE\}$, to define C to be a constant circuit with output width n, whose ith output is the value b_i.

We also use the notation

$$\texttt{CONSTANT_INT}\langle n, K\rangle$$

as an equivalent to $\texttt{CONSTANT}\langle k_0, ..., k_{n-1}\rangle$, where $0 \leq K < 2^n$ and $k_{n-1}k_{n-2}...k_0$ is the binary representation of K.

3.2 Variables

A *variable-valued* node emits values determined by a list of Boolean variables. Each of these variables may independently take on the value $TRUE$ or $FALSE$. We use the notation

$$X := \textbf{makeVariable}(n)$$

to define X to be a variable circuit with output width n.

In general, a Boolean circuit can be seen as defining a function whose inputs are all the variable-valued circuits found in the graph.

3.3 Compound Circuits and Gates

Circuits consisting of multiple nodes are called *compound circuits.* The root node of a compound circuit represents a *Boolean gate.* A Boolean gate calculates some function which has a list of Boolean values as input and a list of Boolean values as output.

For a gate G, we use the notation $G.eval$ to refer to the function which G calculates.

A gate's input list of Boolean values is assembled by concatenating the lists of Boolean values coming in from all its input edges. For example, consider a gate with three input edges, where the first edge has width 5, the second has width 7, and the third has width 9. This forms a list of 21 Boolean values as input to the gate. We assign these values indices starting with zero. Thus, if the gate's Boolean function makes reference to input value 0, it would use the first Boolean value in the first input edge. A reference to input value 13 would use the second Boolean value in the third input edge.

In this way, a particular Boolean gate does not require a specific number of input edges with specific widths. Rather it only requires some lower bound on the total width of all its input edges. For example, if the highest-indexed input value that a gate makes reference to is that with index 24, then to use this gate the sum of the widths of the input edges must be at least 25. We call this minimum total width of the input edges the *input size* of the gate. If gate G has input size n, then for all $m \geq n$:

$$G.eval(b_0, b_1, ...b_{m-1}) = G.eval(b_0, b_1, ...b_{n-1})$$

That is, G ignores all inputs beyond its input width.

On the other hand, every gate does produce an output edge of a specific width, which we call the *output size.*

Let us sharpen this model by examining specific types of Boolean gates.

3.3.1 Connector Gates

The simplest type of Boolean gate is a *connector gate* which does nothing more than route values from particular input wires directly to particular output wires. No actual Boolean calculation is performed by such a gate.

We will denote a connector gate using the notation $\texttt{CONNECT}\langle i_0, i_1, ..., i_{n-1}\rangle$. This represents a gate with n outputs, where the value of the kth output is the value of the i_kth input. Thus we have:

$$\texttt{CONNECT}\langle i_0, i_1, ..., i_{n-1}\rangle.eval(b_0, b_1, ..., b_{m-1}) = (b_{i_0}, b_{i_1}, ..., b_{i_{n-1}})$$

A common use of connector gates will be of the form $\texttt{CONNECT}\langle 0, 1, ..., n-1\rangle$. Such a gate takes its (first) n input wires and bundles them into a single edge. We will use $\texttt{BUNDLE}\langle n\rangle$ as a shorthand name for this gate.

More generally, we will have need for a gate that selects a certain range of input wires and bundles them together. We will use $\texttt{RANGE}\langle n, i\rangle$ as a shorthand for $\texttt{CONNECT}\langle i, i+1, ..., i+n-1\rangle$, which bundles together n consecutive input wires, starting with the ith input wire.

3.3.2 Simple Gates

Our most basic nontrivial gates will calculate a disjunction of conjunctions of the inputs and/or negations of the inputs. These *simple gates* will have an output width of 1. We will specify such a gate by its *on-set*, i.e. a set of lists of input values that suffice to make the output true. In particular, we denote an on-set as a matrix with entries in the set $\{1, 0, -\}$, with the number of columns equal to the number of input values. Each row designates an input configuration sufficient to make the output true, where the symbol in the ith column indicates the required value for the ith input. 1 indicates a required true input, 0 indicates false, and - indicates no requirement.

This is best illustrated with an example. The following notation:

$$\texttt{ONSET}\left\{\begin{array}{cccc} 1 & 0 & - & - \\ 0 & - & 1 & - \\ - & 1 & - & 1 \end{array}\right\}$$

represents a gate with 4 inputs, which calculates the function:

$$eval(b_0, b_1, b_2, b_3) = (b_0 \wedge \neg b_1) \vee (\neg b_0 \wedge b_2) \vee (b_1 \wedge b_3)$$

where b_i is the value of the ith input.

Note that any Boolean function can be specified by its on-set. For a function with n inputs, there are 2^n possible lists of input values, so one valid on-set is simply the set of all the input lists that make the function true. Of course, such an on-set representation is not likely to be the most compact way to represent the function.

We may also specify a Boolean function by its *off-set*, i.e. a set of lists of inputs that make the function false. By the same reasoning as above, any Boolean function can be represented by its off-set. Thus every on-set has an equivalent off-set, and vice-versa.

The off-set notation for a gate equivalent to the on-set gate above is:

$$\texttt{OFFSET}\left\{\begin{array}{cccc} 0 & 0 & 0 & - \\ 0 & - & 0 & 0 \\ 1 & 1 & - & 0 \end{array}\right\}$$

which calculates the function:

$$eval(b_0, b_1, b_2, b_3) = \neg((\neg b_0 \wedge \neg b_1 \wedge \neg b_2) \vee (\neg b_0 \wedge \neg b_2 \wedge \neg b_3) \vee (b_0 \wedge b_1 \wedge \neg b_3))$$

3.3.3 Compound Gates

Compound gates are constructed by combining simpler gates together. The notation $G_C := G_{top}(G_0, G_1, ..., G_{n-1})$ defines G_C to represent the gate which computes its function by taking its input list, feeding that same list into each of the gates G_i, concatenating the outputs from these gates into a single list, feeding this list of values into the gate G_{top}, and returning the output from G_{top}. Thus, the output width of this gate is the output width of G_{top}, and the input width is the maximum of the input widths of all the G_i.

Formally, the gate G_C calculates the following function:

$$G_C.eval(B) = G_{top}.eval(G_0(B); G_1(B); ...; G_{n-1}(B))$$

where B is a list of inputs and ; represents list concatenation.

We introduce some definitions which will assist in reasoning about compound gates. Using the above notation, we define the *top gate* of a compound gate G_C to be the gate G_{top} and the *bottom gates* to be the set of gates G_i. For convenience, we extend this definition to all gates by saying that non-compound gates have no bottom gates. With this in mind, we define the *sub-gates* of a gate G to be the set that includes G and the sub-gates of all bottom gates of G.

Our definitions stipulate that to compute the function of any gate G, we use the input list of G as the input list of all bottom gates of G. Inductively, when we feed a list of inputs into the gate G, the computation involves feeding this same list of inputs into all the sub-gates of G. Thus, if a particular sub-gate G' occurs several times within the gate G (i.e. G' is a sub-gate of several different bottom gates of G), the same inputs are being fed into each of these occurrences of G', so it is only necessary to perform this G' computation once.

This allows us to think of a compound gate as a rooted DAG, where the root node

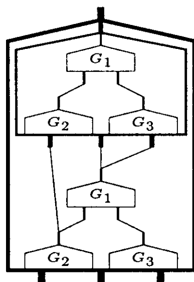


Figure 3-1: The compound gate $G_4(G_2, G_4, G_4)$, where $G_4 = G_1(G_2, G_3)$

represents the top gate, and the DAGs rooted at the children of this node represent the bottom gates. If there are multiple occurrences of a sub-gate, they can all be represented by a single subgraph.

For example, consider the compound gate $G_5 = G_4(G_2, G_4, G_4)$, where $G_4 = G_1(G_2, G_3)$ is itself a compound gate, while G_1, G_2, and G_3 are not compound. Thus G_1, G_2, G_3 have no sub-gates but themselves. G_4's sub-gates are G_4, G_2, G_3. G_5's sub-gates are G_5,G_2,G_3,G_4.

We can consider G_5 as the DAG depicted within the outermost trapezoid shape in Figure 3-1. The root of the DAG is the top gate of G_5, namely the compound gate G_4, represented as a single node (the inner bold trapezoid). The input edges to G_4 come from DAGs representing the bottom gates, G_2, G_4, and G_4. G_2 is a sub-gate of G_5 in three ways, once directly as a bottom gate of G_5 and twice within G_4, but G_2 only appears as a single node in the G_5 graph. Similarly, G_4 is twice a bottom gate of G_5, but only appears as a single subgraph in the graph of G_5. However, this is distinct from the single node in the graph of G_5 which represents G_4 as a whole as the top gate of G_5.

3.4 Variable Setters

As a convenience, we provide an additional type of internal circuit node called a *variable setter.* A variable setter node has exactly one input edge and has an output width the same as its input width. However, unlike a gate node, the output value of a variable setter is not directly the result of computing a function of the input values. In this respect, a variable setter is not a true circuit component. It is only provided as a convenience, allowing a compact representation for a group of circuits that differ only in the value of certain constant-valued sub-circuits.

The output of a variable setter is what the output value of its input circuit would be if a particular variable were replaced with a particular constant. We use the notation

$$\texttt{SETTER}\langle X, K\rangle$$

for a variable setter node that replaces the variable circuit X with the constant circuit $\texttt{CONSTANT_INT}\langle n, K\rangle$, where n is the output width of X.

Any circuit containing variable setters can be desugared into a circuit free of variable setters by applying the following replacement algorithm for each sub-circuit rooted at a variable setter. Let C be the input circuit to the variable setter. Simply replace the sub-circuit with a copy of C that has the variable being set replaced with the appropriate constant, if the variable appears in the sub-circuit. To make a copy of C, we copy all the internal nodes, but use the same variable leaf nodes (with the exception of the variable being set). See Figure 3-2 for an example of desugaring a circuit containing a variable setter.

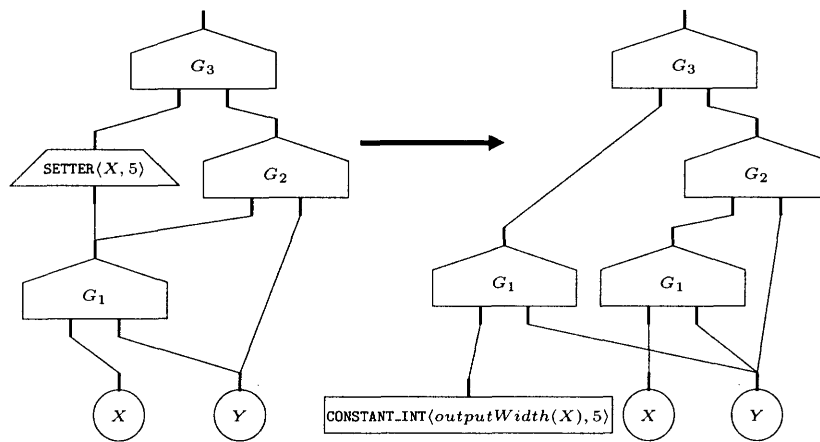


Figure 3-2: Desugaring a circuit containing a variable setter

Chapter 4

Translation Strategies

The basic strategy for translating an Alloy AST into a Boolean circuit is very straightforward. Recall the structure of an Alloy AST. The leaves represent either variables or constant expressions, and the internal nodes represent operations or quantifiers. We will examine each of these types of nodes in turn, and show how they can be directly mapped to nodes of a Boolean circuit. We can thus create a Boolean circuit with a structure roughly isomorphic to the AST that it translates.

4.1 Basic Idea

Let us first keep in mind that all nodes in the AST have either relational, integer, or Boolean values. The Boolean circuit model described in Chapter 3 requires that every node have a list of Boolean values as its output. A Boolean value in Alloy can trivially be represented as a list of a single Boolean value. Relational values and integer values can also both be represented by lists of Boolean values, as mentioned in chapter 3

In particular, an n-ary relation with column types $T_0, ..., T_{n-i}$ where type T_k has scope s_k, can be represented by an $s_0 \times ... \times s_{n-1}$ array of Boolean values. The value with index $(i_0, ..., i_{n-1})$ indicates whether the tuple $(x_0, ..., x_{n-1})$ is in the relation,

where x_k is the i_kth element of type T_k. We can organize an n-dimensional array into a list in a canonical way. For example, we can define the following function to give the canonical list index of the element indexed by $I = (i_0, ..., i_{n-1})$ in an array of dimensions $S = (s_0, ..., s_{n-1})$:

$$flatIndex(I, S) = \sum_{k=0}^{n-1} \left(i_k \prod_{k'=k+1}^{n-1} s_{k'} \right)$$

The relational value can thus be represented as a list of $\prod_{k=0}^{n-1} s_k$ Boolean values.

4.1.1 Variables

Every Alloy variable has a particular relational type. Thus for each variable we encounter in the AST we create a variable-valued circuit of the size necessary to represent this relational type. For a variable x whose relational type has column scopes $s_0, ..., s_{n-1}$, we create the circuit $C_{\mathsf{x}} :=$ **makeVariable**$(\prod_{k=0}^{n-1} s_k)$. Every time we encounter a particular variable in the AST, we use this same circuit.

4.1.2 Constant Expressions

Constant-valued Alloy expressions such as 5, and such as none, are translated as constant-valued circuits. The details of translating such expressions are described in Appendix A.

4.1.3 Operators

In the Alloy AST, an operator has a list of Boolean, relation, or integer values as inputs, and a Boolean, relation, or integer value as output. Naturally, we will translate an Alloy operator as a gate in the Boolean circuit. The only complication is that an Alloy operator can be used on inputs of various sizes, whereas a Boolean gate expects

inputs of specific sizes. In particular, we must specify the scopes of all columns of relation-valued inputs, and the number of bits of all integer-valued inputs.

As an example, consider Alloy's union operator +. This operator takes two relations of the same type as input, and produces another relation of this same type as output. The output relation contains exactly those tuples that are in at least one of the input relations. To translate the union operator into a gate, we must create a gate with a specific input and output size, which of course depends on the column scopes of the input and output relations. Since the inputs and output of the union operator all have the same relational type, the design of the gate only actually depends on the column scopes of this one type. Thus, we actually need a family of gates parameterized by this list of scopes. We use the notation `UNION`$\langle s_0, s_1, ..., s_{n-1}\rangle$ for a gate that calculates the union of relations whose type has column scopes $(s_0, ..., s_{n-1})$.

We similarly need a family of gates to translate Alloy's integer addition operator (which also looks like +). `SUM`$\langle n, m\rangle$ is the gate that adds an n-bit integer and an m-bit integer.

Precise gate translations of all Alloy operators can be found in the appendix.

4.1.4 Quantifiers

The most straightforward Boolean circuit translation of quantifiers involves the use of variables setters. For example, consider a quantified expression of the form some x: Foo | F, where F is any Boolean-valued expression. Let n be the scope of Foo, C_{x} be the circuit representing the variable x, and C_{F} be the circuit translation of F. To obtain the value of the quantified expression, we must take the disjunction of the values given by F when x takes on every atomic value of type Foo.

Recall that an atomic value in Alloy is really a unary relation containing a single element. Thus, as a Boolean circuit, an atomic value is represented by an n-bit constant with exactly one $TRUE$ bit. We can set x to have all such values by using

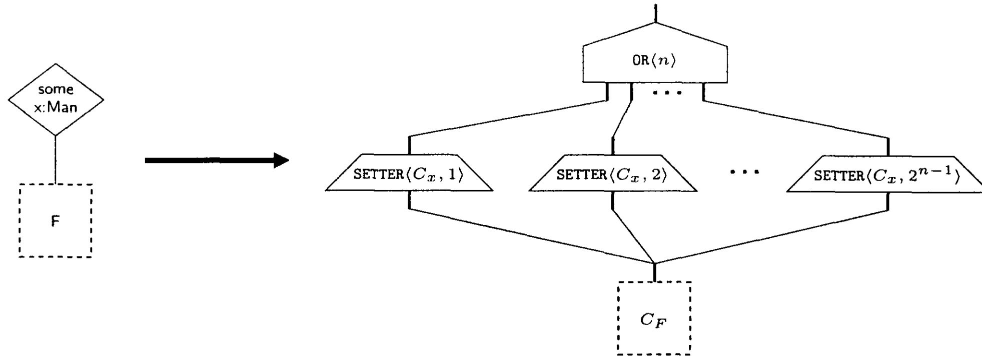


Figure 4-1: Translating an quantified Alloy expression

the variable setters of the form $\texttt{SETTER}\langle C_{\mathsf{x}}, 2^i\rangle$, for $0 \leq i < n$. The circuit representing the entire quantified expression is then constructed by applying these variable setters to the circuit C_{F} and taking their disjunction using the gate $\texttt{OR}\langle n\rangle$. This example is depicted in Figure 4-1.

Descriptions of how to translate all Alloy quantified expressions into Boolean circuits can be found in the appendix.

4.2 Optimized Gates

One way to create gates that may have more room for simplification is to make gates that translate Alloy operators under specific assumptions about the inputs. In particular, we may make gates that assume that certain of their inputs are identical.

For example, consider a gate G with 3 inputs of width n_1, n_2, and n_3 respectively. We may create a version of G that works under the assumption that the first and third inputs have the same value, and then use this version of G in any context where the first and third inputs to G are the same circuit.

To create this version of G, we can simply have G look at its first input to obtain both its first and third input values. In other words, we create the gate:

$$G(\texttt{RANGE}\langle n_1, 0\rangle, \texttt{RANGE}\langle n_2, n_1\rangle, \texttt{RANGE}\langle n_1, 0\rangle)$$

This gate captures the fact that it only actually depends on n_1+n_2 Boolean values, rather than all $n_1 + n_2 + n_3$ input values. In general, if a gate has several identical input circuits, we may create an optimized version of this gate by having the gate use the location of the first of the identical inputs to obtain the values of all the identical inputs.

Additionally, we may create gates that assume particular inputs have constant values. To achieve this we will need to use the notation $\texttt{CONST}\langle n, K\rangle$ to refer to a constant-valued gate of output width n, whose output values are the bits representing the integer K. (This is similar to the notation we use for constant-valued circuits.) These constant-valued gates can be created as a bundle of simple gates with no inputs. A precise definition can be found in Appendix A.

For an example of optimizing for constant values, here is a version of G that assumes that its first input is the circuit $\texttt{CONSTANT_INT}\langle n_1, K_1\rangle$ and its third input is the circuit $\texttt{CONSTANT_INT}\langle n_3, K_3\rangle$:

$$G(\texttt{CONST}\langle n_1, K_1\rangle, \texttt{RANGE}\langle n_2, n_1\rangle, \texttt{CONST}\langle n_3, K_3\rangle)$$

This gate may be used in the same way as the unoptimized version, but the optimized gate ignores the actual first and third input circuit values.

Figure 4-2 shows a how a gate with 4 inputs is optimized both to expect its first and third inputs to be identical, and also to assume that its fourth input has a particular constant value. Note that the overall is circuit is not modified in any way other than replacing the original gate with its optimized version. The same circuits are fed in as inputs to the optimized gate; the difference is that certain inputs are ignored by the new gate.

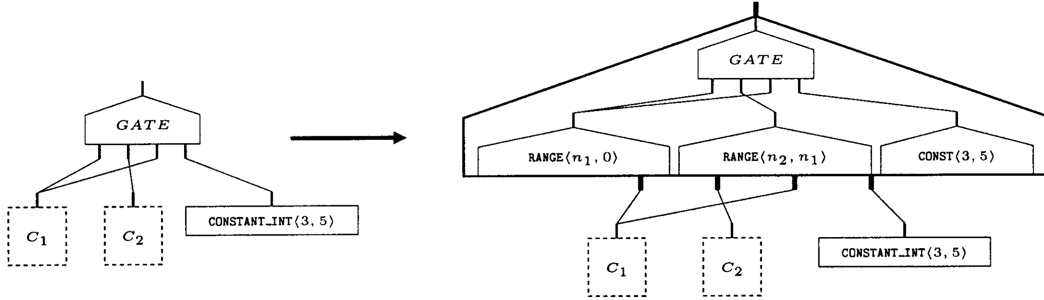


Figure 4-2: Optimizing a gate for a specific configuration of inputs

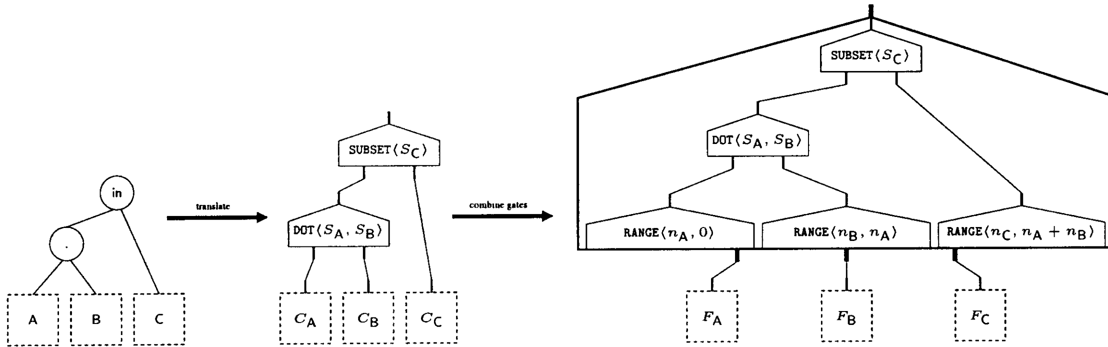


Figure 4-3: Combining multiple Alloy operations into a single gate

4.3 Combined Gates

We may also create gates that are prime candidates for simplification by combining multiple Alloy operations into a single gate. For example, consider the Alloy expression A.B in C, where A, B, and C are are any appropriately typed subexpressions. Let S_A, S_B, and S_C respectively be lists of the finite bounds on the column types of the relations A, B, and C.

In the Alloy AST, this expression takes the form depicted in the leftmost tree in Figure 4-3. The straightforward circuit translation has a graph representation isomorphic to the AST, with the gates DOT$\langle S_A, S_B \rangle$ performing the . operation and SUBSET$\langle S_C \rangle$ performing the in operation. This circuit is depicted as the middle tree in Figure 4-3. C_A, C_B, and C_C are the circuits representing the formulas A, B, and C, respectively. Let n_A, n_B, and n_C refer to output size of these circuits.

Alternatively, we could have a single gate perform both Alloy operations. The following gate, also depicted as the root node of the rightmost circuit in Figure 4-3, would perform this calculation:

$$\texttt{SUBSET}\langle \mathsf{S_C} \rangle(\texttt{DOT}\langle \mathsf{S_A}, \mathsf{S_B} \rangle(\texttt{RANGE}\langle n_\mathsf{A}, 0\rangle, \texttt{RANGE}\langle n_\mathsf{B}, n_\mathsf{A}\rangle), \texttt{RANGE}\langle n_\mathsf{C}, n_\mathsf{A} + n_\mathsf{B}\rangle)$$

This gate expects 3 inputs circuits of output size n_A, n_B, and n_C. It feeds the values from the first two of these circuits into the $\texttt{DOT}\langle \mathsf{S_A}, \mathsf{S_B} \rangle$, and then feeds the outputs from this gate and the values from the third input into $\texttt{SUBSET}\langle \mathsf{S_C} \rangle$.

More generally, consider a circuit containing a gate node G with k input circuits C_0,...,C_{k-1} whose output widths respectively are n_0,...n_{k-1}. For each C_i that is a compound circuit, let G_i be the root gate node of C_i and let C_{i0},...,$C_{i(k_i-1)}$ be the input circuits to G_i and let n_{i0},...,$n_{i(k_i-1)}$ be the output widths of these circuits.

Say that we wish to combine G with certain of the G_i, and in particular let I be the set of indices of the gates we wish to combine with G.

Let us use the notation $G[A_0, ..., A_{k-1}]$ to refer to the combined gate we desire, where $A_i = G_i$ if $i \in I$ and $A_i = n_i$ if $i \notin I$.

We create this combined gate as follows:

$$\text{Let } n'_i = \begin{cases} \sum_{j=0}^{k_i-1} n_{ij} & \text{if } i \in I \\ n_i & \text{if } i \notin I \end{cases}$$

$$\text{Let } N'_i = \sum_{i'=0}^{i-1} n'_i$$

$$\text{Let } N'_{ij} = \sum_{j'=0}^{j-1} n_{ij'}$$

$$\text{Let } G'_i = \begin{cases} G_i(\texttt{RANGE}\langle n_{i0}, N'_i + N'_{i0}\rangle, ..., \texttt{RANGE}\langle n_{i(k_i-1)}, N'_i + N'_{i(k_i-1)}\rangle) & \text{if } i \in I \\ \texttt{RANGE}\langle N'_i, n'_i\rangle & \text{if } i \notin I \end{cases}$$

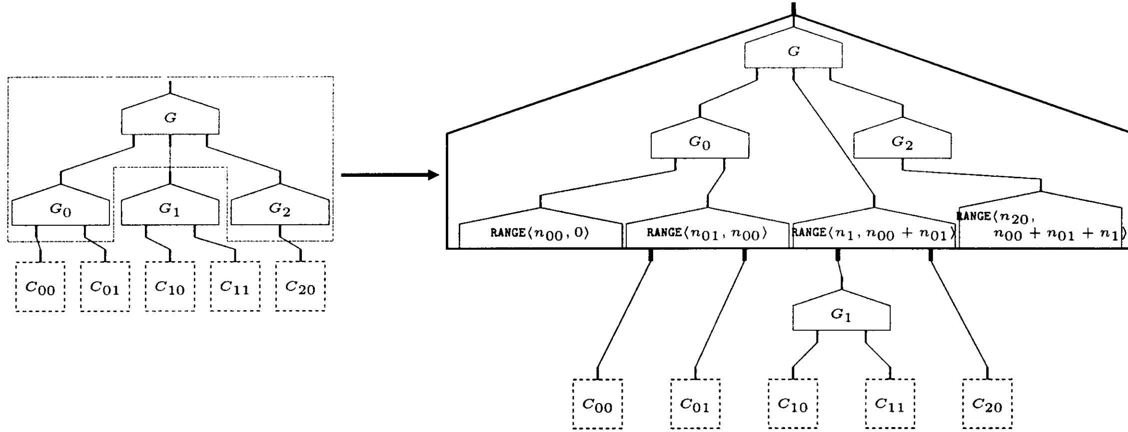


Figure 4-4: Combining a gate node with two of its child nodes in the circuit

Then the combined gate we desire is $G(G'_0, ..., G'_{k-1})$. We reassemble the circuit by replacing G with this combined gate, and feeding consecutively as inputs into the combined gate the circuits C_{i0},...,$C_{i(k_i-1)}$ if $i \in I$ and C_i otherwise.

Figure 4-4 illustrates an example of this procedure, where G has three input circuits and is combined with the root gates of the first and third input circuits. The formal notation for this combined gate is $G[G_0, n_1, G_2]$.

Chapter 5

Implementation

Recall that the overall goal of this thesis is to translate Alloy models into CNF formulas. The steps involved in this process are the following:

1. Parsing the Alloy model to create an AST.

2. Converting the AST into a Boolean circuit.

3. Simplifying components of the Boolean circuit.

4. Converting the Boolean circuit into a CNF formula.

The existing Alloy framework takes care of the first step. The basics of step 2 were described in the previous chapter, but some of the details need to be filled in. In particular, we need to decide when to create combined and/or optimized gates. In this chapter we will fill in the remaining details of step 2 and also describe steps 3 and 4.

5.1 Creating the Circuit

Once an Alloy model has been parsed, we traverse the AST in depth-first manner, translating variables, constant expressions, compound expressions, and quantified ex-

pressions as described in the previous chapter. In particular, when we translate a compound expression, consisting of an operator and some subexpressions, we first find the circuit translations of all the subexpressions, and we obtain the canonical gate translation of the operator as listed in Appendix A. Next, if any of the circuit translations of the subexpressions are compound circuits, we may choose to combine the gates at the root of these compound sub-circuits with the current gate, as described in the previous chapter. Finally, once we have optionally created the combined gate and assembled the compound circuit for the current AST node, we may check for possible gate optimizations. That is, we check to see if any of the inputs to the gate are constant-valued circuits, or if any of the inputs are identical circuits, and replace the gate with a version optimized for this configuration of inputs.

5.2 Simplification

As we create gates, we choose to find simplified representation for certain of them. Circuit simplification is achieved using the MVSIS logic minimization tool [5] developed at the University of California, Berkeley. MVSIS provides a number of functions for manipulating and simplifying logic circuits. MVSIS uses the BLIF-MV format to specify circuits. BLIF-MV [4] is a robust format which allows for nondeterminism and multi-valued circuit nodes. We will be using only a small subset of the BLIF-MV format, in that we will only be dealing with deterministic circuits containing Boolean-valued nodes.

Appendix B describes the syntax of the subset of BLIF-MV used here. Of note is the fact that the BLIF-MV table construct describes an on-set or off-set, and thus it is straightforward to print a simple gate as a BLIF-MV table.

Simplifying a Boolean gate requires three steps: writing the gate in BLIF-MV format, running MVSIS simplification functions, and recreating a gate from the sim-

plified BLIF-MV model.

Let us first consider writing the gate in BLIF-MV. We must list the names of the input and output nodes, and then print all the tables that define the network that relates the input and output nodes. To print the tables we define the procedure **printBlifMVTables**, shown in Figure 5-1, that takes a gate and arrays of names to use for input and output nodes, and prints the series of BLIF-MV tables that together form a network that performs the gate's calculation. To allow for the procedure to be used recursively for printing compound gates, there is an additional input, which is a map from each previously encountered sub-gate to an array of the names used as the sub-gates output nodes in the network. If the compound gate is not a tree (i.e. sub-gates are reused), this map will be used by the procedure to recall the output node names used the first time the sub-gate is encountered.

The procedure will generate fresh names for any internal nodes that are needed. The $outputVars$ parameter may optionally be *null*, in which case the procedure will also generate fresh names for the output nodes, if necessary. However, if any of the gate's outputs are directly connected from the inputs, as in the case of connector gates and certain compound gates composed of connector gates, the procedure will not generate new BLIF-MV nodes for these outputs but rather return the name of the input node as the name of the output node. This prevents the BLIF-MV representation of a gate from being unnecessarily large when it contains connector gates, whose purpose is convenience rather than performing a calculation.

To invoke the **printBlifMVTables** procedure, we pass in arrays of canonical names for the input and output nodes (i.e. `in-0`, `in-1`, ... and `out-0`, `out-1`, ...), and an empty map.

Once the gate is in BLIF-MV, we can load the gate into MVSIS and apply circuit simplification commands. The most powerful simplification method available in MV-SIS tool is called *mfs* [5] which uses an algorithm created by Mishchenko and Brayton

```
String[] printBlifMVTables(Gate g,
                           String[] inputVars,
                           String[] outputVars,
                           Map<Gate->String[]> gateVarMap) {
  if (g is-a SimpleGate) {
    if (outputVars == null) {
      outputVars = [generate array of fresh names]
    }

    [ print a single table representing the on-set or off-set ]

    return outputVars
  } else if (g is-a ConnectorGate) {

    [ assume g has the form CONNECT<i_0,i_1,...,i_{n-1}> ]

    if (outputVars == null) {
      return {inputVars[i_0],inputVars[i_1],...,inputVars[i_{n-1}]}
    } else {
      for (k = 0 to n-1) {
        [ print a single table with a single input inputVars[k]
          and output outputVars[i_k], which causes the output
          to have the same value as the input ]
      }
      return outputVars
    }
  } else if (g is-a CompoundGate) {

    [ assume g has the form G_top(G_0,G_1,...,G_{n-1}) ]

    for (k = 0 to n-1) {
      if (gateVarMap.containsKey(G_k)) {
        subVars[k] = gateVarMap.get(G_k)
      } else {
        subVars[k] =
          printBlifMVTables(G_k, inputVars, null, gateVarMap)
        gateVarMap.put(G_k -> subVars[k])
      }
    }
    topInputVars = concatenate arrays subVars[0],...,subVars[n-1]
    return printBlifMVTables(G_top,topInputVars,outputVars,EMPTY_MAP)
  }
}
```

Figure 5-1: The procedure printBlifMVTables()

[6]. One of the main techniques used, developed by Savoj [7], is the calculation of the flexibility of internal nodes in the form of don't care values. That is, roughly, it finds situations for which the values of particular nodes in the circuit don't matter, and uses this to simplify the nodes.

Unfortunately, as currently implemented, *mfs* times out on the BLIF networks of some of the larger gates, particularly the combined gates. (see Chapter 6). In the cases when *mfs* times out, we try a simpler version (*fullsimp*) of this algorithm, and if this also times out, we leave the network as is.

As we simplify gates, we would like to remember the simplified representation of each gate we simplify, so that when we encounter the same gate in the future, we can avoid repeating the same simplification work that was already done. To this end, after running the MVSIS simplification algorithm, we save the simplified network in BLIF-MV format in a file whose name is taken from the Boolean gate we are simplifying. The next time we encounter this gate, we see that a file exists containing the simplified network, and so we skip directly to the final step of recreating a Boolean gate from this BLIF-MV model. In this way, we actually build up a database of pre-simplified Boolean gates that will be available not only later in the construction of the same Alloy model, but even for future Alloy sessions.

Finally, let us consider the task of parsing the simplified BLIF-MV model to reconstruct a new, simplified Boolean gate. Recall that the BLIF-MV model consists of a list of tables, one for each output node and one for each internal node. We construct a map of node names to BLIF-MV tables. We then assemble the tables into a Boolean gate using the procedure **constructGate**, shown in Figure 5-2.

In particular, we first invoke **constructGate** on the name of the first output node, the map of BLIF-MV tables constructed by parsing the BLIF-MV model, and an empty map of gates. We then proceed to invoke the procedure for the name of each subsequent output node, passing in the same maps, the latter of which is

```
Gate constructGate(String outputVar,
                   Map<String->String> tableMap,
                   Map<String->Gate> gateMap) {
  if (gateMap.containsKey(outputVar)) {
    return gateMap.get(outputVar)
  } else {
    String table = tableMap.get(outputVar)

    [ parse table to produce the following values:
      Gate G = simple gate based on the table's on-set or off-set
      String[] inputVars = list of names of input nodes
      int n = number of input nodes ]

    for (int k = 0 to n-1) {
      if (inputVars[k] is one of the network's input nodes) {

        [ assume inputVars[k] is the ith input node ]

        childGates[k] = CONNECT<i>
      } else {
        childGates[k] =
          constructGate(inputVars[k], tableMap, gateMap)
      }
    }
    Gate resultGate = G(childGates[0],...,childGates[n-1])
    gateMap.put(outputVar -> resultGate)
    return resultGate
  }
}
```

Figure 5-2: The procedure constructGate()

accumulating all the gates constructed for the internal nodes of the circuit. Finally, once we have created gates for each output node, we bundle these together using a connector gate.

5.3 Converting into CNF

Finally, we have a Boolean circuit representing the Alloy model, with various gates having been simplified as desired. It remains to convert this Boolean circuit into conjunctive normal form (CNF), so that it can be solved by a SAT solver tool.

One way to accomplish this would be to start by desugaring out all the variable setters that appear in the circuit. Once the circuit contains only variables and gates, it would be straightforward to print the entire circuit in BLIF-MV. We could then mechanically translate the BLIF-MV network into CNF by creating CNF variables for each node in the network, and creating clauses to establish the correct relationship between the inputs and outputs of each table in the network.

However, if possible, it would be desirable to take advantage of other translation optimizations that are already used in the Alloy Analyzer. In particular, the current Analyzer tool makes use of shared sub-formulas in quantified expressions, to create more compact CNF formulas [3]. This optimization in fact achieves a significant reduction in CNF size, to the point of making manageable formulas out of Alloy models that would otherwise have Boolean representations too large to be reasonably processed by a SAT solver.

In our Boolean circuit, quantified expressions are expressed by the use of multiple variable setters to set values in a single sub-circuit. By desugaring out the variable setters, we would lose the opportunity to benefit from the sharing analysis. Thus, we instead translate the circuit into a Boolean formula representation that includes a construct equivalent to variable setters, which is used in the current implementation

of Alloy. By putting our circuits into this form, we can take advantage of the infrastructure that already exists to detect sharing in these formulas and then to create CNF translations.

Chapter 6

Results

To measure the success of the proposed Boolean circuit translation system, we tested its effectiveness on the following three Alloy models:

- **dijkstra**: A model that demonstrates that Dijkstra's mutex ordering criterion prevents deadlocks. We check that the criterion succeeds in preventing deadlocks for 5 processes and 5 mutexes in traces of up to length 5.

- **handshake**: A model of Paul Halmos's puzzle involving handshakes among couples at a dinner party. We find a solution in the case of 5 couples, i.e. 10 people.

- **stable-mutex-ring**: A model of Dijkstra's self-stabilizing mutual exclusion algorithm for rings. We find a non-repeating trace of the algorithm with 3 nodes and 5 time-steps.

In choosing scopes, it was desirable to keep them small enough to produce gates that could be handled by the MVSIS simplification tools. With respect to gates representing single Alloy operators, the ^ (transitive closure) and * (reflexive transitive closure) operators, which are unary operators whose input is a binary relation with

two columns of the same type, proved to be the most troublesome. The *mfs* simplification function timed out on the gates representing these operators when the scope of the column types was 6 or higher. As a result, we avoided using scopes above 5 for the **dijkstra** and **stable-mutex-ring** models. The **handshake** model does not use either of these operators, making it possible to set a scope of 10 for the Person type without timing out on any gates representing single operators.

We chose to simplify all gates that represented combinations of Alloy operators. For the gates representing single Alloy operators we only simplified those that were complex enough such that it was not obvious that the straightforward translation was the simplest.

We tested each of the following paradigms for deciding which gates to combine:

- **basic simplified**: Do not combine any gates. This will illustrate the benefit just from simplifying gates corresponding to single Alloy operators.

- **unary-up**: Combine each gate with any child nodes in the circuit that are unary gates.

- **unary-down**: Combine every unary gate whose child node is a gate with the child gate.

- **level2**: For each gate, if the sub-circuit rooted at this gate had depth 2 and contains no variable setters, combine the gate with the other gates in the sub-circuit.

We also performed the following additional tests, that do not make use of any circuit simplification:

- **unsimplified**: Use our circuit translation, but do not simplify any Boolean gates.

- **no circuits**: Use the preexisting Alloy translation implementation. This should in general be very similar to the unsimplified circuit translation.

In Tables 6.1 and 6.2, we list the time taken by the MVSIS tool to simplify the gates used in the circuit translations of the above three models. Under the **basic simplified** paradigm, the only gates used are the ones in Table 6.1 and other gates representing single operators that are too trivial to be worth simplifying. The combined gates listed in Table 6.2 were created in accordance with one or more of **unary-up**, **unary-down**, or **level2**.

In Tables 6.3, 6.4, and 6.5, we list the size of the CNF generated when using each of the above paradigms. We also provide the time taken by two different SAT solvers, Berkmin and Mchaff, to find solutions for the CNF formulas, or conclude that there is no solution in the case of the **dijkstra** model. For each test, 5 trials were conducted under Red Hat Linux 8.0, on a 2GHz Pentium IV laptop with 256MB RAM. The times listed in the tables are the average of 5 trials. While the times are somewhat subject to statistical variation, essentially all the trials fell within 10% of the average, indicating that the times are fairly accurate.

We provide the results both with and without using the sharing optimization. Keep in mind that in the **basic simplified** trial, all occurrences of a particular Alloy operator (with particular input sizes) have the same translation. However, when gates are combined, this is no longer the case, so the opportunities for sharing are reduced. Thus, combining gates is only worthwhile if the gain from the additional simplification outweighs the loss of sharing.

The results in the tables correspond to performing the translation without optimizing any gates. Unfortunately, optimizing gates did not generate any change in CNF size on any of the examples used here. It seems likely that optimizing gates to expect constant inputs would not generally result in much improvement in CNF size, since the presence of constants will reduce CNF size anyway. Optimizing gates to

gate	simplification time (sec)
AT_MOST⟨3, 2⟩	0.10
CARDINALITY⟨10⟩	0.92
CARDINALITY⟨3⟩	0.24
CARDINALITY⟨4⟩	0.10
CARDINALITY⟨5⟩	0.10
REFLEXIVE_TRANSITIVE_CLOSURE⟨5⟩	5.31
SOLE⟨1⟩	0.09
SOLE⟨3⟩	0.21
SOLE⟨4⟩	0.10
SOLE⟨5⟩	0.11
SOLE⟨10⟩	0.22
TRANSITIVE_CLOSURE⟨3⟩	0.13
TRANSITIVE_CLOSURE⟨4⟩	0.33
TRANSITIVE_CLOSURE⟨5⟩	6.02

Table 6.1: Gate simplification times for individual Alloy operators

expect certain inputs to be identical seems more promising, although this might only be helpful on gates that were too large to be simplified by our use of MVSIS.

Overall, the results here are promising. In every case when sharing is not used, simplifying only the gates for basic Alloy operators results in a CNF size reduction of 10% or more. The unary-up paradigm produces a further small improvement. However, the unary-down and level2 paradigms unfortunately did not generate further improvement. This is due in part to the fact that a number of the combined gates created by the level2 paradigm caused MVSIS to time-out. With a more robust simplification method that could handle large gates, it is quite possible that a significant improvement could be obtained.

The timing benefits are not as apparent in these results. It should be noted that in the model that yields no instance (**dijkstra**), there is generally a direct correspondence between CNF size and solution time, since the SAT solver must always search through the entire space of possible boolean assignments. However, the solution time is less predictable for the other two models, since the solver stops as soon as it finds the instance.

Also, it is important to realize that time taken to simplify the Boolean gates, as listed in Tables 6.1 and 6.2, are not negligible relative to the solution time. Thus, if we were to perform the simplifications anew during each Alloy translation run, any

gate	simplification time (sec)
AT_MOST⟨3, 2⟩[CARDINALITY⟨4⟩, CARDINALITY⟨3⟩]	0.12
AND⟨2⟩[AND⟨1⟩, AND⟨5⟩]	0.34
AND⟨2⟩[NOT, 1]	0.10
AND⟨2⟩[NOT, SOLE⟨1⟩]	0.09
AND⟨2⟩[NOT, SOLE⟨3⟩]	0.10
AND⟨2⟩[NOT, SOLE⟨4⟩]	0.10
AND⟨2⟩[NOT, SOLE⟨5⟩]	0.10
AND⟨2⟩[NOT, SOLE⟨10⟩]	0.98
CARDINALITY⟨10⟩[DOT⟨(10), (10, 10)⟩]	*mfs* and *fullsimp* timed out
DIFF⟨5⟩[5, DOT⟨(1), (1, 5)⟩]	0.21
DIFF⟨5⟩[5, DOT⟨(5), (5, 5)⟩]	0.16
DOT⟨(3), (3, 4)⟩[3, DOT⟨(5), (5, 3, 4)⟩]	0.39
DOT⟨(3), (3, 3)⟩[3, TRANSITIVE_CLOSURE⟨3⟩]	0.14
DOT⟨(3), (3, 4)⟩[DOT⟨(3), (3, 3)⟩, DOT⟨(5), (5, 3, 4)⟩]	0.99
DOT⟨(4), (4, 4)⟩[4, TRANSITIVE_CLOSURE⟨4⟩]	0.59
DOT⟨(5), (5, 5)⟩[5, DOT⟨(1), (1, 5, 5)⟩]	0.182
DOT⟨(5), (5, 5)⟩[5, DOT⟨(5), (5, 5, 5)⟩]	0.521
DOT⟨(5), (5, 5)⟩[5, REFLEXIVE_TRANSITIVE_CLOSURE⟨5⟩]	4.07
DOT⟨(5), (5, 5)⟩[5, TRANSITIVE_CLOSURE⟨5⟩]	7.87
DOT⟨(5), (5, 5)⟩[5, TRANSPOSE⟨5, 5⟩]	0.11
DOT⟨(5), (5, 5)⟩[DOT⟨(1), (1, 5)⟩, DOT⟨(1), (1, 5, 5)⟩]	0.35
DOT⟨(5), (5, 5)⟩[DOT⟨(5), (5, 5)⟩, DOT⟨(5), (5, 5, 5)⟩]	3.10
DOT⟨(5), (5, 5, 5)⟩[DOT⟨(1), (1, 5)⟩, 125]	0.235
DOT⟨(10), (10, 10)⟩[DOT⟨(10), (10, 10)⟩, 100]	*mfs* and *fullsimp* timed out
EQUALITY⟨5⟩[5, DOT⟨(1), (1, 5)⟩]	0.22
EQUALITY⟨10⟩[10, DOT⟨(10), (10, 10)⟩]	48.5
EQUALITY⟨10⟩[DOT⟨(10), (10, 10)⟩, 10]	100.13
EQUALITY⟨10⟩[DOT⟨(10), (10, 10)⟩, DOT⟨(10), (10, 10)⟩]	*mfs* and *fullsimp* timed out
EQUALITY⟨12⟩[DOT⟨(5), (5, 3, 4)⟩, DOT⟨(5), (5, 3, 4)⟩]	*mfs* and *fullsimp* timed out
EQUALITY⟨25⟩[25, TRANSPOSE⟨5, 5⟩]	0.27
EQUALS⟨4, 4⟩[CARDINALITY⟨10⟩, 4]	0.20
EQUALS⟨4, 4⟩[CARDINALITY⟨10⟩, CARDINALITY⟨10⟩]	0.53
NOT[EQUALITY⟨3⟩]	0.10
NOT[EQUALITY⟨4⟩]	0.1
NOT[EQUALITY⟨10⟩]	0.13
NOT[SUBSET⟨1⟩]	0.09
NOT[SUBSET⟨3⟩]	0.09
NOT[SUBSET⟨4⟩]	0.10
NOT[SUBSET⟨5⟩]	0.09
NOT[SUBSET⟨10⟩]	0.16
OR⟨2⟩[NOT, 1]	0.09
OR⟨2⟩[NOT, NOT]	0.09
REFLEXIVE_TRANSITIVE_CLOSURE⟨5⟩[DOT⟨(1), (1, 5, 5)⟩]	4.89
SUBSET⟨3⟩[3, DOT⟨(5), (5, 3)⟩]	0.15
SUBSET⟨3⟩[DOT⟨(3), (3, 3)⟩, 3]	0.10
SUBSET⟨3⟩[DOT⟨(5), (5, 3)⟩, 3]	0.26
SUBSET⟨4⟩[DOT⟨(4), (4, 4)⟩, 4]	0.17
SUBSET⟨5⟩[DOT⟨(1), (1, 5)⟩, 5]	2.68
SUBSET⟨5⟩[DOT⟨(5), (5, 5)⟩, 5]	0.17
SUBSET⟨5⟩[INTERSECTION⟨5⟩, 5]	0.16
SUBSET⟨10⟩[10, DIFF⟨10⟩]	0.16
SUBSET⟨10⟩[10, DOT⟨(10), (10, 10)⟩]	9.63
SUBSET⟨10⟩[DOT⟨(10), (10, 10)⟩, 10]	14.32
SUBSET⟨10⟩[INTERSECTION⟨10⟩, 10]	0.17
SUBSET⟨12⟩[DOT⟨(5), (5, 3, 4)⟩, ARROW⟨3, 4⟩]	209.55
SUBSET⟨25⟩[DOT⟨(1), (1, 5, 5)⟩, ARROW⟨5, 5⟩]	*mfs* timed out; *fullsimp* time: 12.5
SUBSET⟨25⟩[DOT⟨(5), (5, 5, 5)⟩, ARROW⟨5, 5⟩]	*mfs* and *fullsimp* timed out
TRANSITIVE_CLOSURE⟨5⟩[DOT⟨(1), (1, 5, 5)⟩]	9.04
TRANSPOSE⟨5, 5⟩[DOT⟨(1), (1, 5, 5)⟩]	0.11
TRANSPOSE⟨5, 5⟩[DOT⟨(5), (5, 5, 5)⟩]	0.23
UNION⟨10⟩[10, DOT⟨(10), (10, 10)⟩]	0.28

Table 6.2: Gate simplification times for combined gates

	no sharing				sharing			
	CNF size		solution time (sec)		CNF size		solution time (sec)	
	vars	clauses	Berkmin	Mchaff	vars	clauses	Berkmin	Mchaff
no circuits	27984	73076	33.4	56.9	15347	42281	33.4	56.9
unsimplified	29367	76975	35.6	244.3	16557	45480	7.7	46.9
basic simp.	26104	69194	23.1	153.6	13278	37659	9.5	31.1
unary-up	25972	68960	30.1	884.4	13162	37465	6.2	23.2
unary-down	26104	69194	21.8	552.9	13278	37659	9.5	34.4
level2	26104	69194	30.0	1241.9	13343	37789	6.9	48.8

Table 6.3: Results for dijkstra model

	no sharing				sharing			
	CNF size		solution time (sec)		CNF size		solution time (sec)	
	vars	clauses	Berkmin	Mchaff	vars	clauses	Berkmin	Mchaff
no circuits	15846	53395	251.8	192.2	6977	19261	11.5	187.2
unsimplified	14467	43157	9.7	timeout	7313	17589	1.57	10.1
basic simp.	12141	32768	198.7	15.6	10849	25781	18.3	23.2
unary-up	12135	32763	153.4	49.3	12035	32753	127.9	165.2
unary-down	13669	40323	40.5	timeout	6605	14755	1.68	20.3
level2	12141	32768	93.6	16.0	10849	25781	34.6	23.2

Table 6.4: Results for handshake model

	no sharing				sharing			
	CNF size		solution time (sec)		CNF size		solution time (sec)	
	vars	clauses	Berkmin	Mchaff	vars	clauses	Berkmin	Mchaff
no circuits	7600	20614	0.56	0.12	5606	13898	0.49	0.09
unsimplified	8312	22927	0.87	0.27	6105	15624	0.53	0.10
basic simp.	6637	18907	0.57	0.13	4422	11584	0.43	0.10
unary-up	6570	18787	0.55	0.11	4363	11484	0.39	0.08
unary-down	6637	18907	0.57	0.14	4422	11584	0.43	0.10
level2	6636	18904	0.59	0.12	4441	11621	0.43	0.08

Table 6.5: Results for stable-mutex-ring model

timing benefits from the shorter solution time might be canceled out by the time taken by the MVSIS tool for simplification. Fortunately, as described in Chapter 6, as we perform simplifications, we create a database of pre-simplified BLIF-MV networks. By providing users of the Alloy tool with a database containing pre-simplified versions of commonly used gates, we can avoid losing time to simplification. In fact, even providing pre-simplified gate representations only of individual Alloy operators, such as those listed in Table 6.1, would be useful, since much of the CNF size reduction exhibited in these experiments came just from simplifying these gates.

Chapter 7

Conclusion

There is certainly potential benefit to using simplified boolean representations of various Alloy components. Simply finding reduced versions of basic Alloy operators in models of relatively small scope led to some reduction in CNF size.

However, there is clearly more research to be done with regards to simplifying larger-size boolean networks. The boolean gates that represent larger Alloy operators and combinations of operators, which are produced by the circuit translation implemented here, provide good benchmarks with which to challenge researchers in logic minimization.

7.1 Future Work

Ilya Shlyakhter has developed an optimization that uses symmetry-breaking predicates to reduce the CNF size for an Alloy model [8]. If certain instances of an Alloy model are symmetric, we are only interesting discovering one of them. Given a predicate that is true for one instance in every symmetry class, it is only necessary to search for instances that satisfy this predicate. A number of such predicates can be found by analyzing an Alloy model.

These symmetry-breaking predicates may provide another way to optimize gates

in our boolean circuit representation. The BLIF-MV format supports the inclusion of don't care networks in a circuit specification. It should be possible to use the don't care networks to indicate that we don't care about output values in situations where the symmetry-breaking predicates don't apply. This would provide additional flexibility in gate simplification, which could bring further reduction in boolean circuit size.

Appendix A

Standard Translations

A.1 Operators

We provide standard translations of Alloy compound expressions by representing operators as boolean gates.

We use the following shorthand notation:

$$GATE[i_0, ..., i_{n-i}] := GATE(\texttt{CONNECT}\langle i_0 \rangle, ..., \texttt{CONNECT}\langle i_{n-1} \rangle)$$

We also make use of the following gates:

$$\texttt{NOT} := \texttt{ONSET}\left\{\ 0\ \right\}$$

$$\texttt{AND}\langle k \rangle := \texttt{ONSET}\overbrace{\left\{\ 1\ \ 1\ \ \ldots\ \ 1\ \right\}}^{k \text{ columns}}$$

$$\texttt{OR}\langle k \rangle := \texttt{OFFSET}\overbrace{\left\{\ 0\ \ 0\ \ \ldots\ \ 0\ \right\}}^{k \text{ columns}}$$

$$\texttt{IMPLIES} := \texttt{OFFSET}\left\{\ 1\ \ 0\ \right\}$$

$$\texttt{IFF} := \texttt{ONSET}\left\{\begin{array}{cc} 1 & 1 \\ 0 & 0 \end{array}\right\}$$

$$\texttt{TRUE} := \texttt{OFFSET}\left\{\right\}$$

$$\texttt{FALSE} := \texttt{ONSET}\left\{\right\}$$

$$\texttt{CONST}\langle n,k\rangle := \texttt{BUNDLE}\langle n\rangle(\begin{aligned}[t]&\texttt{CONST_BIT}\langle 0,k\rangle,\\ &\texttt{CONST_BIT}\langle 1,k\rangle,\\ &\vdots\\ &\texttt{CONST_BIT}\langle n-1,k\rangle)\end{aligned}$$

$$\texttt{CONST_BIT}\langle i,k\rangle := \begin{cases}\texttt{FALSE} & \text{if } (k \bmod 2^{i+1}) < 2^i\\ \texttt{TRUE} & \text{otherwise}\end{cases}$$

$$\texttt{BIT_A}\langle i,n,m\rangle := \begin{cases}\texttt{CONNECT}\langle i\rangle & \text{if } 0 \leq i < n\\ \texttt{FALSE} & \text{if } i \geq n\end{cases}$$

$$\texttt{BIT_B}\langle i,n,m\rangle := \begin{cases}\texttt{CONNECT}\langle i+n\rangle & \text{if } 0 \leq i < m\\ \texttt{FALSE} & \text{if } i \geq m\end{cases}$$

The following functions give a canonical ordering of elements in a multidimensional array:

$$flatIndex((i_0,...,i_{n-1}),(s_0,...,s_{n-1})) = \sum_{k=0}^{n-1}\left(i_k \prod_{k'=k+1}^{n-1} s_{k'}\right)$$

$$realIndex(i,(s_0,...,s_{n-1})) = \\ \text{the } (i_0,...,i_{n-1}) \text{ such that } flatIndex((i_0,...,i_{n-1}),(s_0,...,s_{n-1})) = i$$

$$flatSize((s_0,...,s_{n-1})) = \prod_{k=0}^{n-1} s_k$$

Here are the translations of Alloy operators to boolean gates:

A && B → AND⟨2⟩

A and B are booleans

A || B → OR⟨2⟩

A and B are booleans

A <=> B → IFF

A and B are booleans

A => B → IMPLIES

A and B are booleans

$\boxed{\mathsf{!A} \rightarrow \texttt{NOT}}$

A is a boolean

$\boxed{\mathsf{A\ in\ B} \rightarrow \texttt{SUBSET}\langle n\rangle}$

A and B both are relations with column scopes S

$n = flatSize(S)$

$$\begin{aligned}\texttt{SUBSET}\langle n\rangle := \texttt{AND}\langle n\rangle(&\texttt{IMPLIES}[0, n],\\ &\texttt{IMPLIES}[1, n+1],\\ &\vdots\\ &\texttt{IMPLIES}[0, n])\end{aligned}$$

$\boxed{\mathsf{A = B} \rightarrow \texttt{EQUALITY}\langle n\rangle}$

A and B both are relations with column scopes S

$n = flatSize(S)$

$$\begin{aligned}\texttt{EQUALITY}\langle n\rangle := \texttt{AND}\langle n\rangle(&\texttt{IFF}[0, n],\\ &\texttt{IFF}[1, n+1],\\ &\vdots\\ &\texttt{IFF}[0, n])\end{aligned}$$

$\boxed{\mathsf{no\ A} \rightarrow \texttt{NO}\langle n\rangle}$

A is a relation with column scopes S

$n = flatSize(S)$

$$\texttt{NO}\langle n\rangle := \texttt{EQUALS}\langle\lfloor\log n\rfloor + 1, 1\rangle(\texttt{CARDINALITY}\langle n\rangle, \texttt{CONST}\langle 1, 0\rangle)$$

$\boxed{\mathsf{sole\ A} \rightarrow \texttt{SOLE}\langle n\rangle}$

A is a relation with column scopes S

$n = flatSize(S)$

$$\texttt{SOLE}\langle n\rangle := \texttt{AT_MOST}\langle\lfloor\log n\rfloor + 1, 1\rangle(\texttt{CARDINALITY}\langle n\rangle, \texttt{CONST}\langle 1, 1\rangle)$$

$\boxed{\mathsf{one\ A} \rightarrow \texttt{ONE}\langle n\rangle}$

A is a relation with column scopes S

$n = flatSize(S)$

$$\texttt{ONE}\langle n\rangle := \texttt{EQUALS}\langle\lfloor\log n\rfloor + 1, 1\rangle(\texttt{CARDINALITY}\langle n\rangle, \texttt{CONST}\langle 1, 1\rangle)$$

two A $\rightarrow$ $\mathtt{TWO}\langle n\rangle$

A is a relation with column scopes S
$n = flatSize(S)$

$$\mathtt{TWO}\langle n\rangle := \mathtt{EQUALS}\langle\lfloor\log n\rfloor + 1, 2\rangle(\mathtt{CARDINALITY}\langle n\rangle, \mathtt{CONST}\langle 2, 2\rangle)$$

some A $\rightarrow$ $\mathtt{SOME}\langle n\rangle$

A is a relation with column scopes S
$n = flatSize(S)$

$$\mathtt{SOME}\langle n\rangle := \mathtt{GREATER_THAN}\langle\lfloor\log n\rfloor + 1, 1\rangle(\mathtt{CARDINALITY}\langle n\rangle, \mathtt{CONST}\langle 1, 0\rangle)$$

A.B $\rightarrow$ $\mathtt{DOT}\langle S, T\rangle$

A is a relation with column scopes $S = (s_0, ..., s_{p-1})$
B is a relation with column scopes $T = (t_0, ..., t_{q-1})$
$k = s_{p-1} = t_0$
the resulting expression has relational type $R = (s_0, ..., s_{p-2}, t_1, ..., t_{q-1})$
$n = flatSize(R)$

$$\begin{aligned}\mathtt{DOT}\langle S, T\rangle := \mathtt{BUNDLE}\langle n\rangle(&\mathtt{DOT_COMPONENT}\langle realIndex(0, R), S, T\rangle,\\ &\mathtt{DOT_COMPONENT}\langle realIndex(1, R), S, T\rangle,\\ &\vdots\\ &\mathtt{DOT_COMPONENT}\langle realIndex(n-1, R), S, T\rangle)\end{aligned}$$

$$\begin{aligned}&\mathtt{DOT_COMPONENT}\langle(i_0, ..., i_{p+q-2}), S, T\rangle :=\\ &\quad \mathtt{OR}\langle k\rangle(\mathtt{AND}\langle 2\rangle[\ flatIndex((i_0, ..., i_{p-2}, 0), S),\\ &\qquad\qquad flatIndex((0, i_{p-1}, ..., i_{p+q-2}), T) + flatSize(S)\ ],\\ &\qquad \mathtt{AND}\langle 2\rangle[\ flatIndex((i_0, ..., i_{p-2}, 1), S),\\ &\qquad\qquad flatIndex((1, i_{p-1}, ..., i_{p+q-2}), T) + flatSize(S)\ ],\\ &\qquad \vdots\\ &\qquad \mathtt{AND}\langle 2\rangle[\ flatIndex((i_0, ..., i_{p-2}, k-1), S),\\ &\qquad\qquad flatIndex((k-1, i_{p-1}, ..., i_{p+q-2}), T) + flatSize(S)\ ])\end{aligned}$$

A + B → $\mathtt{UNION}\langle n \rangle$

A and B both are relations with column scopes S
$n = flatSize(S)$

$$\begin{aligned}\mathtt{UNION}\langle n \rangle := \mathtt{BUNDLE}\langle n \rangle(&\mathtt{OR}\langle 2 \rangle[0, n],\\ &\mathtt{OR}\langle 2 \rangle[1, n+1],\\ &\vdots\\ &\mathtt{OR}\langle 2 \rangle[n-1, 2n])\end{aligned}$$

A & B → $\mathtt{INTERSECTION}\langle n \rangle$

A and B both are relations with column scopes S
$n = flatSize(S)$

$$\begin{aligned}\mathtt{INTERSECTION}\langle n \rangle := \mathtt{BUNDLE}\langle n \rangle(&\mathtt{AND}\langle 2 \rangle[0, n],\\ &\mathtt{AND}\langle 2 \rangle[1, n+1],\\ &\vdots\\ &\mathtt{AND}\langle 2 \rangle[n-1, 2n])\end{aligned}$$

A - B → $\mathtt{DIFF}\langle n \rangle$

A and B both are relations with column scopes S
$n = flatSize(S)$

$$\begin{aligned}\mathtt{DIFF}\langle n \rangle := \mathtt{BUNDLE}\langle n \rangle(&\mathtt{DIFF_BIT}[0, n],\\ &\mathtt{DIFF_BIT}[1, n+1],\\ &\vdots\\ &\mathtt{DIFF_BIT}[n-1, 2n])\end{aligned}$$

$$\mathtt{DIFF_BIT} := \mathtt{ONSET}\{\ 1\quad 0\ \}$$

A -> B → $\texttt{ARROW}\langle S, T\rangle$

A is a relation with column scopes $S = (s_0, ..., s_{p-1})$
B is a relation with column scopes $T = (t_0, ..., t_{q-1})$
the resulting expression has relational type $R = (s_0, ..., s_{p-1}, t_0, ..., t_{q-1})$
$n = flatSize(R)$

$$\begin{aligned}\texttt{ARROW}\langle S,T\rangle := \texttt{BUNDLE}\langle n\rangle(&\texttt{ARROW_COMPONENT}\langle realIndex(0,R),S,T\rangle,\\ &\texttt{ARROW_COMPONENT}\langle realIndex(1,R),S,T\rangle,\\ &\vdots\\ &\texttt{ARROW_COMPONENT}\langle realIndex(n-1,R),S,T\rangle)\end{aligned}$$

$$\begin{aligned}&\texttt{ARROW_COMPONENT}\langle(i_0,...,i_{p+q-1}),S,T\rangle :=\\ &\qquad \texttt{AND}\langle 2\rangle[flatIndex((i_0,...,i_{p-1}),S),\\ &\qquad\qquad flatIndex((i_p,...,i_{p+q-1}),T)+flatSize(S)\quad]\end{aligned}$$

~A → $\texttt{TRANSPOSE}\langle s_0, s_1\rangle$

A is a binary relation with column scopes (s_0, s_1)
$n = flatSize((s_1, s_0))$

$$\begin{aligned}\texttt{TRANSPOSE}\langle s_0,s_1\rangle := \texttt{BUNDLE}\langle n\rangle(&\texttt{TRANSPOSE_BIT}\langle realIndex(0,(s_1,s_0)),(s_0,s_1)\rangle,\\ &\texttt{TRANSPOSE_BIT}\langle realIndex(1,(s_1,s_0)),(s_0,s_1)\rangle,\\ &\vdots\\ &\texttt{TRANSPOSE_BIT}\langle realIndex(n-1,(s_1,s_0)),(s_0,s_1)\rangle,\end{aligned}$$

$$\texttt{TRANSPOSE_BIT}\langle(i_1,i_0),(s_0,s_1)\rangle := \texttt{CONNECT}\langle flatIndex((i_0,i_1),(s_0,s_1))\rangle$$

^A → $\texttt{TRANSITIVE_CLOSURE}\langle s\rangle$

A is a binary relation with column scopes (s, s)
$n = flatSize((s, s))$

$$\texttt{TRANSITIVE_CLOSURE}\langle s\rangle := \texttt{SQUARED}\langle\lceil\log s\rceil, s\rangle$$

$$\texttt{SQUARED}\langle i,s\rangle := \begin{cases}\texttt{BUNDLE}\langle n\rangle & \text{if } i=0\\ \texttt{SQUARER}\langle s\rangle(\texttt{SQUARED}\langle i-1,s\rangle) & \text{if } i>0\end{cases}$$

$$\texttt{SQUARER}\langle s\rangle := \texttt{UNION}\langle n\rangle(\texttt{BUNDLE}\langle n\rangle, \texttt{DOT}\langle(s,s),(s,s)\rangle(\texttt{BUNDLE}\langle n\rangle,\texttt{BUNDLE}\langle n\rangle))$$

$\boxed{\text{*A} \rightarrow \texttt{REFLEXIVE_TRANSITIVE_CLOSURE}\langle s\rangle}$

A is a binary relation with column scopes (s, s)
$n = flatSize((s, s))$

$$\texttt{REFLEXIVE_TRANSITIVE_CLOSURE}\langle s\rangle := \texttt{UNION}\langle n\rangle(\texttt{IDENTITY}\langle s\rangle, \texttt{TRANSITIVE_CLOSURE}\langle s\rangle)$$

$$\begin{aligned}\texttt{IDENTITY}\langle s\rangle := \texttt{BUNDLE}\langle n\rangle(&\texttt{IDENTITY_BIT}\langle realIndex(0, (s, s))\rangle,\\ &\texttt{IDENTITY_BIT}\langle realIndex(1, (s, s))\rangle,\\ &\vdots\\ &\texttt{IDENTITY_BIT}\langle realIndex(n-1, (s, s))\rangle)\end{aligned}$$

$$\texttt{IDENTITY_BIT}\langle(i_0, i_1)\rangle := \begin{cases}\texttt{TRUE} & \text{if } i_0 = i_1\\ \texttt{FALSE} & \text{if } i_0 \neq i_1\end{cases}$$

$\boxed{\text{if A then B else C} \rightarrow \texttt{IF_THEN_ELSE}\langle n\rangle}$

A is a boolean
B and C both are relations with column scopes S
$n = flatSize(S)$

$$\begin{aligned}\texttt{IF_THEN_ELSE}\langle n\rangle := \texttt{BUNDLE}\langle n\rangle(&\texttt{IF_THEN_ELSE_BIT}[0, 1, n+1],\\ &\texttt{IF_THEN_ELSE_BIT}[0, 2, n+2],\\ &\vdots\\ &\texttt{IF_THEN_ELSE_BIT}[0, n, 2n])\end{aligned}$$

$$\texttt{IF_THEN_ELSE_BIT} := \texttt{ONSET}\left\{\begin{array}{ccc}1 & 1 & -\\ 0 & - & 1\end{array}\right\}$$

$\boxed{\mathsf{A} = \mathsf{B} \rightarrow \texttt{EQUALS}\langle n, m\rangle}$

A is an n-bit integer
B is an m-bit integer

$$\begin{aligned}
&\texttt{EQUALS}\langle n, m\rangle := \\
&\quad \texttt{AND}\langle max(n,m)\rangle(\texttt{IFF}(\texttt{BIT_A}\langle 0, n, m\rangle, \texttt{BIT_B}\langle 0, n, m\rangle), \\
&\qquad\qquad \texttt{IFF}(\texttt{BIT_A}\langle 1, n, m\rangle, \texttt{BIT_B}\langle 1, n, m\rangle), \\
&\qquad\qquad \vdots \\
&\qquad\qquad \texttt{IFF}(\texttt{BIT_A}\langle max(n,m) - 1, n, m\rangle, \texttt{BIT_B}\langle max(n,m) - 1, n, m\rangle))
\end{aligned}$$

$\boxed{\mathsf{A} <= \mathsf{B} \rightarrow \texttt{AT_MOST}\langle n, m\rangle}$

A is an n-bit integer
B is an m-bit integer

$$\texttt{AT_MOST}\langle n, m\rangle := \texttt{COMPARATOR}\langle max(n,m) - 1, n, m\rangle$$

$$\texttt{COMPARATOR}\langle i, n, m\rangle := \begin{cases} \texttt{TRUE} & \text{if } i = 0 \\ \texttt{AND}\langle 2\rangle(\texttt{COMPARATOR}\langle i-1, n, m\rangle, \\ \quad \texttt{IMPLIES}(\texttt{PREV_EQUAL}\langle i, n, m\rangle, \\ \quad\quad \texttt{IMPLIES}(\texttt{BIT_A}\langle i, n, m\rangle, \\ \quad\quad\quad \texttt{BIT_B}\langle i, n, m\rangle))) & \text{if } i > 0 \end{cases}$$

$$\texttt{PREV_EQUAL}\langle i, n, m\rangle := \begin{cases} \texttt{TRUE} & \text{if } i = 0 \\ \texttt{AND}\langle 2\rangle(\texttt{PREV_EQUAL}\langle i-1, n, m\rangle, \\ \quad \texttt{IFF}(\texttt{BIT_A}\langle i-1, n, m\rangle, \\ \quad\quad \texttt{BIT_B}\langle i-1, n, m\rangle)) & \text{if } i > 0 \end{cases}$$

$\boxed{\mathsf{A} < \mathsf{B} \rightarrow \texttt{LESS_THAN}\langle n, m\rangle}$

A is an n-bit integer
B is an m-bit integer

$$\texttt{LESS_THAN}\langle n, m\rangle := \texttt{AND}\langle 2\rangle(\texttt{AT_MOST}\langle n, m\rangle, \texttt{NOT}(\texttt{EQUALS}\langle n, m\rangle))$$

$\boxed{\mathsf{A} >= \mathsf{B} \rightarrow \texttt{AT_LEAST}\langle n, m\rangle}$

A is an n-bit integer
B is an m-bit integer

$$\texttt{AT_LEAST}\langle n, m\rangle := \texttt{NOT}(\texttt{LESS_THAN}\langle n, m\rangle)$$

$\boxed{\mathsf{A} > \mathsf{B} \rightarrow \texttt{GREATER_THAN}\langle n, m\rangle}$

A is an n-bit integer
B is an m-bit integer

$$\texttt{GREATER_THAN}\langle n, m\rangle := \texttt{NOT}(\texttt{AT_MOST}\langle n, m\rangle)$$

$\boxed{\mathsf{A} + \mathsf{B} \rightarrow \texttt{SUM}\langle n, m\rangle}$

A is an n-bit integer
B is an m-bit integer

$$\begin{aligned}\texttt{SUM}\langle n, m\rangle := \texttt{BUNDLE}\langle max(n, m) + 1\rangle(&\texttt{SUM_BIT}\langle 0, n, m\rangle,\\ &\texttt{SUM_BIT}\langle 1, n, m\rangle,\\ &\vdots\\ &\texttt{SUM_BIT}\langle max(n, m), n, m\rangle)\end{aligned}$$

$$\begin{aligned}\texttt{SUM_BIT}\langle i, n\rangle := \texttt{SUMMER}(&\texttt{BIT_A}\langle i, n\rangle,\\ &\texttt{BIT_B}\langle i, n\rangle,\\ &\texttt{CARRY_BIT}\langle i, n\rangle)\end{aligned}$$

$$\texttt{CARRY_BIT}\langle i, n\rangle := \begin{cases}\texttt{FALSE} & \text{if } i = 0\\ \begin{aligned}\texttt{CARRIER}(&\texttt{BIT_A}\langle i-1, n\rangle,\\ &\texttt{BIT_B}\langle i-1, n\rangle,\\ &\texttt{CARRY_BIT}\langle i-1, n\rangle)\end{aligned} & \text{if } i > 0\end{cases}$$

$$\texttt{SUMMER} := \texttt{ONSET}\left\{\begin{array}{ccc}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\\ 1 & 1 & 1\end{array}\right\}$$

$$\texttt{CARRIER} := \texttt{ONSET}\left\{\begin{array}{ccc}1 & 1 & -\\ 1 & - & 1\\ - & 1 & 1\end{array}\right\}$$

$\#\mathsf{A} \rightarrow \texttt{CARDINALITY}\langle n \rangle$

A is a relation with column scopes S
$n = flatSize(S)$
$mid = \lfloor \frac{n}{2} \rfloor$

$$\texttt{CARDINALITY}\langle n \rangle := \begin{cases} \texttt{CONNECT}\langle 0 \rangle & \text{if } n = 1 \\ \texttt{SUM}\langle \lfloor \log mid \rfloor + 1, \lfloor \log(n - mid) \rfloor + 1 \rangle \\ \quad (\texttt{CARDINALITY}\langle mid \rangle(\texttt{RANGE}\langle mid, 0 \rangle), & \text{if } n > 1 \\ \quad \texttt{CARDINALITY}\langle n - mid \rangle(\texttt{RANGE}\langle n - mid, mid \rangle)) \end{cases}$$

A.2 Constants

Here are translations of constant-valued Alloy expressions into constant circuits:

$\mathsf{none\ A} \rightarrow \texttt{CONSTANT_INT}\langle n, 0 \rangle$

A is a relation with column scopes S
$n = flatSize(S)$

$\mathsf{univ\ A} \rightarrow \texttt{CONSTANT_INT}\langle n, 2^n - 1 \rangle$

A is a relation with column scopes S
$n = flatSize(S)$

$$\mathsf{iden\ A} \rightarrow \texttt{CONSTANT_INT}\langle s^2, \sum_{i=0}^{s-1} \sum_{j=0}^{s-1} \begin{cases} 2^{i+js} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \rangle$$

A is a relation with column scopes (s, s)

Furthermore, an integer literal K is translated as the circuit $\texttt{CONSTANT}\langle \lfloor \log K \rfloor + 1 \rangle$.

A.3 Quantifiers

some x:T | F

F is a boolean-valued expression
The translation for this form of quantified expression is described in Chapter 4.

all x:T | F

F is a boolean-valued expression
The translation for this form of quantified expression is the same as with the some quantifier except that the OR gate is replaced with an AND gate.

`sum x:T | N`

N is an integer-valued expression

The translation for this comprehension expression is the same as the expression with the some quantifier except that the OR gate is replaced with an SUM gate, so that the overall expression has an integer value.

`{x:T | F}`

F is a boolean-valued expression

The translation for this comprehension expression is the same as the expression with the some quantifier except that the OR gate is replaced with an BUNDLE gate, so that the expression has the same relational value as T.

Appendix B

BLIF-MV

This appendix describes the small subset of BLIF-MV used in this thesis. A BLIF-MV *model* is a unit that represents a boolean network with any number of input and output nodes. As such, a model captures our concept of a boolean gate.

Here is the syntax of a model in BLIF-MV:

```
.model <model-name>
.inputs <input-list>
.outputs <output-list>
<table>
...
<table>
.end
```

<model-name> is a string that names the model.

<input-list> is a white-space separated list of strings that name the network's input nodes. (If there are no input nodes, this line may be omitted.)

<output-list> is a white-space separated list of strings that name the network's output nodes.

A <table> determines the value of a particular node from a list of other nodes using on-set/off-set notation, just like our concept of a simple gate. The syntax for a table is:

```
.table <in-1> <in-2> ... <in-n> <out>
.default <default-val>
<relation>
...
<relation>
```

The <in-*i*>s name the inputs nodes for this table, and <out> names the output node.

`<default-val>` is either `1` or `0`. It indicates what the value of `<out>` will be if none of the `<relation>` lines hold, with `1` indicating $TRUE$ and `0` indicating $FALSE$. Thus, if `<default-val>` is `0`, the `<relation>`s will specify an on-set, and if `<default-val>` is `1`, the `<relation>`s will specify an off-set.

Each `<relation>` is a white-space separated list of $n+1$ characters. The last character must be `0` if `<default-val>` is `1`, and `1` if `<default-val>` is `0`. The first n characters must be one of `1`, `0`, or `-`.

The value of a table's output node is determined as follows: For any `<relation>`, if for all i such that the ith character is 1 `<in-`i`>` is $TRUE$, and for all j such that the jth character is 0 `<in-`j`>` is $FALSE$, then `<out>` takes on the value indicated by the $(n+1)$th character (i.e. the opposite of `<default-val>`). If none of the `<relation>`s apply, then `<out>` takes on the value `<default-val>`.

A table's output node may either be one of the network's output nodes named in `<output-list>` or an internal node named neither in `<input-list>` nor `<output-list>`. In order that the network be deterministic, each of the network's output nodes and every other internal node used in the network must be the output of exactly one table. The network's input nodes may not be used as the output of a table.

There may be not be any cyclic dependencies among the nodes of the network. That is, if we define a depends-on relation that relates the output node of a table to all input nodes of a table, this relation may not be cyclic.

Bibliography

[1] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. *Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)*, September 2001.

[2] Daniel Jackson. Automating first-order relational logic. *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, November 2000.

[3] Ilya Shlyakhter, Manu Sridharan, Robert Seater, and Daniel Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. *6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Portofino, Italy*, May 2003.

[4] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. R Kurshan, S. Malik, A. L. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang. BLIF-MV: An interchange format for design verification and synthesis. Technical report, University of California, 1991.

[5] Minxi Gao, Jie-Hong Jiang, Yunjian Jiang, Yinghua Li, Subarna Sinha, and Robert Brayton. Mvsis. *In the Notes of the International Workshop on Logic Synthesis*, June 2001.

[6] Alan Mishchenko and Robert Brayton. Simplification of non-deterministic multi-valued networks. *IEEE/ACM International Conference on CAD, ICCAD 02*, November 2002.

[7] Hamid Savoj. *Don't cares in multi-level network optimization.* PhD thesis, University of California at Berkeley, Berkeley, CA, 1992.

[8] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Electronic Notes in Discrete Mathematics*, June 2001.