A Static Analysis to Detect Coupling and Conflicts in Declarative Specifications

by

Ian R. Schechter

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science
and
Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

© Ian R. Schechter, MM. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 22, 2000

Certified by

Daniel Jackson
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
A Static Analysis to Detect Coupling and Conflicts in Declarative Specifications

by

Ian R. Schechter

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 2000, in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Engineering

Abstract

Alloy is a declarative specification language especially designed for object modelling. Using Alcoa, an automatic design checking tool, models in Alloy can be automatically analyzed to yield simulations or to find bugs. Models in Alloy consist of a number of separate paragraphs representing conditions or operations to be simulated, or claims to be verified. When a given paragraph in an Alloy model is analyzed, all the model’s stated invariants and constraints on state elements are implicitly conjoined with the paragraph’s formula. This sometimes results in the inclusion of extraneous and unnecessary invariants in analyses.

This thesis presents polarity analysis, a static analysis of Alloy that can be used to accurately determine which invariants and state elements a given Alloy formula is dependent on, and prevent the inclusion of unnecessary invariants in analyses of Alloy models.

Polarity analysis is further presented as the basis of a simplification algorithm that dramatically simplifies Alloy formulas, facilitating their analysis.

The design, correctness, and implementation of this analysis are presented, along with the results of applying it to a number of various specifications.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor
To my father.
Acknowledgments

I'd like to offer my most sincere gratitude to my advisor Daniel Jackson. He offered me immeasurable amounts of assistance at every step of the way, and has really opened up my mind to a much deeper understanding of the logic and mathematics behind software. He is really the best advisor a graduate student could ever hope to have. In addition, I’d like to thank Mandana Vaziri who also offered me an incredible amount of assistance in all my academic endeavors. I also owe a great debt of thanks to Ilya Shlyakhter, along with whom I helped build Alcoa, and who has taught me a great many things. I should also acknowledge other students with whom I’ve had the pleasure of working in the Software Design Group; Sarfraz Khurshid, Uriel Schafer, Manu Sridharan, Jon Whitney, and Derek Bruening. I’d also like to thank John Chapin, and pay homage to his vision of the Rivet Virtual Machine that originally drew me into this group.

I’d like to thank the Center for Innovation in Product Development, and the National Science Foundation which funded part of my work. Similarly, I owe a great debt to Alcoa (the Aluminum Corporation of America) for their nifty name :-).

On a personal note, I’d like to thank my dear wife Sharyn for making my life at MIT livable, and for all her love and support. Lastly, but most importantly, I’d like to thank my beloved parents, David and Andrea, to whom I owe everything, and for whom I have no words that can express my gratitude.
## Contents

1 Introduction ................................................. 9
   1.1 Overview ........................................... 9
   1.2 Introduction to Alloy .................................. 10
   1.3 Introduction to Polarity Analysis ......................... 11
       1.3.1 Problem to Be Solved .......................... 11
       1.3.2 Background on Polarity Analysis ................. 12

2 Structure and Semantics of Alloy ..................... 15
   2.1 Structure of Alloy Models ........................... 15
   2.2 Implicit Conjunctions Of Relevant Formulas ........... 17
   2.3 Semantics of Alloy Models and Instances ............ 20

3 Design of Polarity Analysis .......................... 23
   3.1 Overview ........................................... 23
   3.2 Formal Presentation of the Polarity Analysis .......... 23
       3.2.1 Formal Definition of the Polarity Analysis over Alloy 23
       3.2.2 Soundness of the Polarity Analysis ................ 30
3.2.3 Benefit of the Polarity Analysis 49

3.3 Runtime Analysis 51

4 Implementation of the Polarity Analysis 53

4.1 Overview 53

4.2 Alcoa’s Architecture and Initial Implementation 54

4.2.1 Sharing Analysis 55

4.2.2 Conflict Analysis 56

4.3 Extended Implementation with Propagation for Formula Simplification 57

4.3.1 Regular Formulas 58

4.3.2 DNF Clauses 58

4.4 Extended Implementation with Propagation for Eliminating Dependencies 61

5 Results 62

5.1 Overview 62

5.2 Performance on Eliminating Dependencies 62

5.3 Performance on Simplifying Clauses 66

6 Related Work 70

7 Conclusion 72

A List of Functions 74
List of Figures

3-1 Polarity Lattice ..................................................... 24

4-1 Alcoa Object Model .................................................. 55
# List of Tables

1.1 Metavalues of Components ........................................ 14

2.1 Alloy’s Syntax and Type Rules ..................................... 16

2.2 Semantic Functions and Definitions .............................. 21

5.1 Number of Dependencies Detected by Sharing and Conflict Analysis .......................... 64

5.2 Number of Transitive Dependencies Detected by Sharing and Conflict Analysis .................. 65

5.3 Percentage Reduction in Formula Size ............................. 68

5.4 Polarity Based Simplification Results With Multiple DNF clauses ............................ 69
Chapter 1

Introduction

1.1 Overview

Robust software is difficult to design. This is a well known problem, which none have yet solved to universal satisfaction. Different approaches and methodologies abound, each of which addresses the problems of software design in its own way.

One approach used to facilitate the design of software is the use of formal specification languages. Formal specification languages allow for the precise mathematical description of software systems. Systems described in formal specification languages can be analyzed and formally reasoned about. This formal reasoning can allow flaws to be identified in a design, or can lead to a proof of correctness for a design. This thesis will focus on one formal specification language: Alloy.

Alloy is a declarative specification language which has been especially designed for object modeling. Specifications, or models, in Alloy can be automatically analyzed to obtain simulations and counterexamples. In this thesis, we will present a static analysis of Alloy called
polarity analysis. The information gained from polarity analysis can be used to reduce the number of detected dependencies between specification elements, and to dramatically simplify specifications. By eliminating unnecessary dependencies and simplifying formulas, polarity analysis enables more relevant and faster automatic analyses of specifications.

1.2 Introduction to Alloy

Alloy, a “lightweight, precise, and tractable notation for object modeling” is presented in [6, 9]. An in depth description of Alloy’s syntax and semantics follows in Chapter 2. It is a first order relational logic, based upon a simple calculus of sets and relations. In its use of sets and relations for modeling systems, Alloy approximates a first order subset of Z [11], with some extra power such as the ability to take the transitive closure of relations.

The key distinction between Alloy and Z, VDM, and many other modeling languages, is that Alloy is tractable. Alloy models can be automatically analyzed to find examples of states, simulations of state transitions, and counterexamples to assertions about models. Since Alloy is undecidable, automatic analysis of Alloy only considers all possible configurations of an Alloy model with a fixed number of elements or atoms in each domain. Alcoa [9] is a tool that automatically analyzes Alloy models, by compiling Alloy formulas into boolean constraints and using constraint satisfaction solvers to solve these boolean constraints. A solution to these boolean constraints is mapped back into Alloy’s language of sets and relations, yielding an assignment of elements to domains, sets, and relations that satisfies the given Alloy formula. In automatically finding satisfying assignments to sets and relations in relational formulas, Alcoa is the successor to Nitpick [7], a tool that automatically analyzed
specifications in NP [4], Alloy’s predecessor.

Even though analysis of Alloy formulas only considers the possible state configurations for a fixed number of elements, or “scope”, the space of possible configurations, even for small scopes, is nevertheless tremendous. Any technique capable of reducing the size of Alloy formulas, enabling Alcoa to produce smaller and simpler boolean constraints more easily solved by constraint satisfaction solvers, is well worth exploration. Simplification of Alloy formulas also enables larger, formerly intractable models, to be analyzed. Therefore, simplification of Alloy formulas is the aim of several optimizations done on Alloy models, and is one of the motivations behind polarity analysis.

1.3 Introduction to Polarity Analysis

1.3.1 Problem to Be Solved

Our primary goal in designing and implementing polarity analysis was to detect independence between Alloy formulas. Essentially, we wanted to be able to look at several Alloy formulas, and statically determine whether the formulas depended on one another. In particular, we wanted to be able to look at formulas that mentioned the same exact components, and determine that they were independent, or that an analysis of one did not have to include an analysis of the other.
1.3.2 Background on Polarity Analysis

The key to our approach to the problem of detecting dependence among formulas was rooted in an insight of Daniel Jackson’s regarding the behavior of domains and Alloy formulas. The basic observation Jackson made was that different satisfying assignments to components of Alloy formulas fell into broad categories that could be characterized statically.

Consider a simple Alloy model:

```alloy
model polarities {
  domain {D}
  state {s : D, t : D}   //The sets s and t are both within the domain D
  inv C1 {s in t}       //The set s is a subset of the set t
  inv C2 {some t}       //The set t is not empty
}
```

This model starts with a domain paragraph that declares a domain \( D \), and a state paragraph that declares the sets \( s \) and \( t \). The two declarations in the state paragraph can be thought of as state formulas that constrain the elements of the model’s state.

The declaration “\( s : D \)” can be translated into the Alloy formula \( s \in D \), which constrains the set \( s \) to be a subset of the domain \( D \). We’ll refer to this state formula for convenience as \( S_1 \). Similarly, the declaration “\( t : D \)” can be translated into the Alloy formula \( t \in D \), which constrains the set \( t \) to be a subset of the domain \( D \). We’ll refer to this state formula as \( S_2 \).

In addition to the domain and state paragraphs, this model contains two invariants. Each invariant in this model constrains the domain \( D \), and the sets \( s \) and \( t \) in different ways. When we analyze each invariant, by the definition of Alloy, all the other invariants are implicitly
included. However, it’s not always desirable to have to include all the other invariants.

For instance, let’s say that a user wanted a simulation of C2. Alcoa would have to decide which formulas to include in its simulation. The formulas that could be conjoined with C2 are the state formulas S1 (s in D) and S2 (t in D) and the invariant C1 (s in t).

One simple way for Alcoa to figure out what should be conjoined to an analysis of C2 would be to look at what components are mentioned in each formula, and include all formulas that mention components that might affect the components that C2 mentions.

C2 only mentions t. C1 and S2 both mention t, but S2 mentions D, and C1 mentions s. Thus, s and D might affect the value of t, and all formulas mentioning D and s must be included in our analysis. Using this strategy, we will conjoin all of the formulas above together.

This result seems counterintuitive. The user asked for an example where the set t was not empty; why should the value of s be included such an example? Intuitively, at least, C1 and S1 shouldn’t be included in an analysis of C2, despite the fact that our informal “mentioning analysis” indicated that they should be.

This is the exact problem that polarity analysis addresses, through the observation that we can obtain information about classes of satisfying assignments just by superficial examination.

For instance, any satisfying assignment to C1 will still be a satisfying assignment to C1 if we add atoms to the value of t in the assignment. This is true because the set s is a subset of the set t, and it must also be a subset of all supersets of t. Similarly, if we remove atoms from the value of s in a satisfying assignment, we will still have a valid assignment; since s is a subset of t, any subset of s is also a subset of t.
We can generalize these observations by associating a metavalue with each Alloy component in a formula. If we can take away atoms from the value of a component in any satisfying assignment to a formula and still arrive at a satisfying assignment to the formula, we can associate a \textit{minus} metavalue to that component. Conversely, if we can add atoms to the value of a component in any satisfying assignment to a formula and still arrive at a satisfying assignment to the formula, we can associate a \textit{plus} metavalue to that component. Using these metavalues, we can characterize the conditions in the above model:

<table>
<thead>
<tr>
<th></th>
<th>s's metavalue</th>
<th>t's metavalue</th>
<th>D's metavalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>\textit{minus}</td>
<td>--</td>
<td>\textit{plus}</td>
</tr>
<tr>
<td>S2</td>
<td>--</td>
<td>\textit{minus}</td>
<td>\textit{plus}</td>
</tr>
<tr>
<td>C1</td>
<td>\textit{minus}</td>
<td>\textit{plus}</td>
<td>--</td>
</tr>
<tr>
<td>C2</td>
<td>--</td>
<td>\textit{plus}</td>
<td>--</td>
</tr>
</tbody>
</table>

Some formulas' metavalues seem to contradict with the metavalues of other formulas, whereas some formulas don't seem to conflict at all. These contradictions and conflicts among entries in this table, we will show, actually reflect dependence and independence of the corresponding formulas. In our example, we can clearly see that C2 only conflicts with S2. We will show that this fact can be used to prove that for an analysis of C2 only S2 needs to be conjoined with C2; S1, and C1 can be ignored. This is a much better result than what we obtained by just looking at what sets were mentioned, and didn't require significantly more analysis. In fact, the metavalues of components, which we'll refer to as \textit{polarities}, can be used to gain a wide variety of information about specification elements, and their relationships with other specification elements.
Chapter 2

Structure and Semantics of Alloy

2.1 Structure of Alloy Models

The syntax and typing rules of Alloy are presented in Table 2.1 (based on material from [6]). Alloy models are structured in paragraphs. The domains of an Alloy model, from which the sets and relations of the model are drawn, are enumerated in a domain paragraph.

The state components of an Alloy model, consisting of the sets, relations, and functions of the model are declared in the state paragraph. Each declaration of a state component is actually translated into a formula describing the constraints on the component. These formulas derived from state declarations are called state formulas and are associated within the state paragraph with the state components they constrain. The core of an Alloy model consists of its domain and state paragraphs.

In addition to the domain and state paragraphs, there are condition, invariant, definition, operation, and assertion paragraphs. These paragraphs each consist of a conjunctive Alloy
### Table 2.1: Alloy’s Syntax and Type Rules

<table>
<thead>
<tr>
<th>Syntax of Alloy</th>
<th>Type Rules</th>
</tr>
</thead>
</table>
| **model** ::= model name { (domains | state | cons) * } | $E \vdash a : S$, $E \vdash b : S$  
| 
| **domains** ::= domain {type* } | $E \vdash \text{in } b$  
| **state** ::= state {var : typexpr} | $E, v : T \rightarrow \text{Unit} \vdash f$  
| **typexpr** ::= type | $E \vdash \text{all } v : T | f$  
|  
| **cons** ::= (cond | assert | inv | op) | $E \vdash a : S \rightarrow T$, $E \vdash b : S \rightarrow T$  
| **name** ((var : typexpr) *)? {formula *} | $E \vdash a+b : S \rightarrow T$  
| **formula** ::= expr in expr | $E \vdash a : S \rightarrow T$, $E \vdash b : S \rightarrow U$  
|  
| **expr** ::= expr + expr | $E \vdash a : S \rightarrow T$  
|  
| **Var** ::= var | $E \vdash +a : S \rightarrow T$  

Formula constraining state components and arguments. *Condition* paragraphs express a condition of the model’s state that may or may not be true. *Invariant* paragraphs express conditions constraining the model’s state that must be true of the model’s state at all times. *Definition* paragraphs describe how derived components are constrained by other components of the model’s state. *Operation* paragraphs describe a state transition of the model. *Assertion* paragraphs express conditions on a model’s state that are posited to be true of every legal configuration of the model’s state.

The purpose of analyzing state, definition, condition, and invariant paragraphs is to find an example of the model’s state configured according to the constraints embodied in the paragraph. Assertion paragraphs, however, are analyzed so as to find counterexamples to
the stated assertion.

Analysis of an Alloy formula consists of exhaustively searching through the space of possible values within a fixed scope for every state component and variable mentioned, to find an assignment of values that will cause the formula to be true. Such an assignment is referred to as a satisfying assignment. The details of the process by which Alcoa tries to find satisfying assignments have been discussed at length in other sources [9], and will only be briefly summarized here. In short, Alcoa compiles the relational formulas it is analyzing from Alloy into a series of boolean constraints. Alcoa then uses constraint satisfaction (SAT) solvers to solve these constraints. The solution to the boolean constraints is then mapped back into an assignment of values to the components in the relational formulas.

2.2 Implicit Conjunctions Of Relevant Formulas

When analyzing any paragraph, all invariants, definitions, and state formulas are implicitly included. Deciding which invariant, state, and definition paragraphs must be conjoined to a given paragraph to have a legal analysis of it is one of the main challenges presented by Alloy’s modular design. As an example of some of the issues presented by implicit inclusions of invariants, we propose a hypothetical example:

Let us consider two teams of software developers working on a chess game. The first team, Team A, is working on the graphical rendering engine that will render the chess pieces and the board for display in a window. Here’s a look at a partial specification Team A has been working on:
The second team, Team B, is working on the game engine, and modeling the rules of chess and different strategies. Here is their partial specification:

```plaintext
//Partial model of game engine for chess game
model chess {
    //Piece is the domain of different chess pieces
    //Position is the domain of different positions on a chess board
    //Int is the domain of integers
    domain {Piece, Position, Int}

    state {
        //The different divisions of pieces...
        partition Black, White : Piece
        partition King, Queen, Castle, Bishop, Knight, Pawn : Piece

        //Each piece has a location on the chess board
        location : Piece -> Position !

        //This invariant ensures that all the pieces are at different positions
        inv OnePiecePerSquare {
            all p1, p2 : Piece - p1 | p2.location.column != p2.location.column ||
        }
    }
}
```

18
At some point in the design process, these two teams finish their specifications and integrate them into one total specification of the chess game. This complete specification has all the domains from both partial specifications: Piece, Coordinate, Position, and Int. Moreover, the complete specification has all the invariants, operations, and conditions of the partial models. Thus, it contains OnePiecePerSquare, MeshesDoNotIntersect, and Move.

Once in possession of the total specification, the two teams wish to perform analyses of different paragraphs of the specification. Since the two teams worked separately, the invariants, operations, and conditions largely apply either to the rendering subsystem part of the model, or the game engine part of the model, but not to both. Thus, when one of the software designers uses Alcoa to see an example of Move, the designer is expecting to see assignments for those state elements relevant to the game engine part of the model. In analyzing Move, a user is specifically not expecting to see the coordinate meshes of the different pieces in the generated example. In fact, the user analyzing a high level operation like Move would certainly not want to wait for Alcoa to find an assignment for the very complex graphical rendering part of the model. Intuitively, information about the coordinate meshes is seemingly irrelevant to the high level game engine operations like Move.

Thus, the user analyzing Move would probably not want Alcoa to conjoin the formula of
Move with the formula of the invariant \texttt{MeshesDoNotIntersect}. Even though \texttt{MeshesDoNotIntersect} is an invariant on the same model that \texttt{Move} describes; in some sense \texttt{Move} is "independent" of \texttt{MeshesDoNotIntersect}. At the same time, a user would certainly want analysis of \texttt{Move} to include the invariant \texttt{OnePiecePerSquare}. A move that results in putting a chess piece on an already occupied square should certainly not be considered legal by Alcoa. Thus, \texttt{Move} is "dependent" on \texttt{OnePiecePerSquare}. From this example, we see that the user will expect Alcoa to determine which invariants and state formulas are relevant to their given analysis.

Determining which invariants and state formulas to include in an analysis is a particularly difficult problem in an example like the one above. A naive scheme we could use when analyzing a given invariant would be to conjoin all invariants that mention components that are mentioned in the invariant. Such a scheme would fail to work on our example, since the domain \texttt{Piece} is bound to be mentioned in almost every invariant.

In short, we need a more sophisticated way of determining which invariants must be included in a given analysis. We have shown intuitively that when analyzing a given paragraph Alcoa should include the formulas of paragraphs that the given paragraph is dependent on, and exclude the formulas of paragraphs that the given formula is independent of. Statically determining which paragraphs are dependent on which other paragraphs is the original goal of polarity analysis, and the subject of much of the rest of this thesis.

### 2.3 Semantics of Alloy Models and Instances

Before presenting the semantics of Alloy specifications under polarity analysis, we must first define the regular, concrete semantics of Alloy. For the purposes of this work, we will
restrict our discussion to Alloy's kernel language, the core calculus of Alloy to which the rest of the language easily reduces [6]. The semantics of Alloy's kernel language are presented in Table 2.2 (these semantics have already been presented elsewhere [6, 9])

Table 2.2: Semantic Functions and Definitions

<table>
<thead>
<tr>
<th>$M$</th>
<th>$F$</th>
<th>$GL$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M[a in b]$</td>
<td>$e$</td>
<td>$X[a] e \subseteq X[b] e$</td>
</tr>
<tr>
<td>$M[F]$</td>
<td>$e$</td>
<td>$\neg M[F] e$</td>
</tr>
<tr>
<td>$M[all v : t</td>
<td>F]$</td>
<td>$e$</td>
</tr>
<tr>
<td>$M[some v : t</td>
<td>F]$</td>
<td>$e$</td>
</tr>
<tr>
<td>$X[a + b]$</td>
<td>$e$</td>
<td>$X[a] e \cup X[b] e$</td>
</tr>
<tr>
<td>$X[a&amp;&amp;b]$</td>
<td>$e$</td>
<td>$X[a] e \cap X[b] e$</td>
</tr>
<tr>
<td>$X[a - b]$</td>
<td>$e$</td>
<td>$X[a] e \setminus X[b] e$</td>
</tr>
<tr>
<td>$X[a.b]$</td>
<td>$e$</td>
<td>${(x, unit) \exists y. (y, unit) \in X[a] e \land (y, x) \in X[b] e}$</td>
</tr>
<tr>
<td>$X[\sim a]$</td>
<td>$e$</td>
<td>${(x, y) \mid (x, y) \in X[a] e}$</td>
</tr>
<tr>
<td>$X[\sim a]$</td>
<td>$e$</td>
<td>${(x, y) \mid (x, y) \in \text{transitive closure of } X[a] e}$</td>
</tr>
<tr>
<td>$X[[v : t</td>
<td>F]]$</td>
<td>$e$</td>
</tr>
<tr>
<td>$X[v]$</td>
<td>$e$</td>
<td>$e(v)$</td>
</tr>
<tr>
<td>$X[a[v]]$</td>
<td>$e$</td>
<td>$X[a] e \cup {w</td>
</tr>
</tbody>
</table>

Interpreting Alloy formulas and expressions requires assignments. Assignments map from Alloy types, variables, and components to values. In our scheme, values consist of mappings from atoms to atoms, or from atoms to values, where atoms are the indistinguishable objects that make up our semantic universe. As in Alloy’s type system [6], sets are denoted as mappings from atoms to unit, a specific atom set aside to denote sets and scalars. Relations are simply mappings from atoms to atoms.
In the context of a given assignment, an expression's meaning is a value, and a formula’s meaning is a boolean. For example, the value of the expression $a + b$ in the context of the assignment $\{ a \mapsto (atom_1, unit), b \mapsto (atom_2, unit) \}$ would be $\{(atom_1, unit), (atom_2, unit)\}$. Similarly, $a$ in $b$ given the same assignment $\{ (a \mapsto (atom_1, unit)), b \mapsto (atom_2, unit) \}$ would have the semantic meaning of false. An assignment to the components mentioned in a formula that gives the formula the semantic meaning of true is a satisfying assignment. Table 2.2 completely describes the semantic functions $M$ and $X$ that give expressions and formulas their semantics.

Given an Alloy formula, Alcoa’s task is to try to find an assignment that satisfies the formula, making it true, or find that no such assignment exists in the fixed scope of analysis. We’ll refer to this as the concrete interpretation of an Alloy formula, denoted by the function $S$:

$$S : \text{Formula} \rightarrow \mathcal{P}\text{Assignment}$$

$$S[f] = \{ a \mid a \in \text{Assignment} \land M[f] a \}$$

Formally, Alcoa is a tool that partially implements the function $S$. Polarity analysis will allow Alcoa to approximate $S$ with a different interpretation of formulas. The rest of this thesis will focus on deriving, defining, and analyzing this approximation.
Chapter 3

Design of Polarity Analysis

3.1 Overview

In designing polarity analysis we will pursue two main goals. First, we will develop a technique to statically determine if two Alloy formulas are independent. A precise definition of independence in the context of Alloy formulas follows in Section 3.2.2, but at an intuitive level we wish to determine that in analyzing one Alloy formula, we do not have to consider another Alloy formula with which the given one is implicitly conjoined. Then, we’ll show how to simplify Alloy formulas using substitution of constant values for state components.

3.2 Formal Presentation of the Polarity Analysis

3.2.1 Formal Definition of the Polarity Analysis over Alloy

In Section 2.3, we fully defined the concrete domain of Alloy formulas, as well as the semantics of concrete interpretation of Alloy formulas. Now, we will formally define an alternative
interpretation framework for Alloy specifications: the polarity environment. A polarity environment will map alloy components to polarities. Polarities are abstract values that we will map to components in Alloy formulas. Formally, Polarities are defined as members of the set of polarities:

\[ P \subseteq \{ \top, +, -, * \} \]

Moreover, \( P \) is structured as a lattice:

\[
\begin{array}{c}
\top \\
+ \\
\bullet \\
- \\
\ast \\
\end{array}
\]

Figure 3-1: Polarity Lattice

We define \( \sqcup \) to be a commutative operator on this lattice that yield the “least upper-bound” polarity approximation:

\[
\begin{align*}
+ \sqcup + & = + \\
+ \sqcup - & = * \\
- \sqcup - & = - \\
\forall_p . p \sqcup * & = * \\
\forall_p . \top \sqcup p & = p
\end{align*}
\]

A polarity environment is a structure in which polarities of different components are stored.
We can treat polarity environments as functions from components to polarities:

\[ PEnv : C \rightarrow P \]

where \( C \) is the set of state components of an Alloy model.

We’ll also overload our join operator to define it over polarity environments as follows:

\[ \sqcup : PEnv \rightarrow PEnv \rightarrow PEnv \]

\[ \forall p_1, p_2 \in PEnv, \forall c \in C (p_1 \sqcup p_2)(c) = (p_1(c) \sqcup p_2(c)) \]

Also, we have functions that negate polarities and polarity environments:

\( Neg : PEnv \rightarrow PEnv \) and \( NegPol : P \rightarrow P \) such that

\[
\text{Neg}[p] = \begin{cases} 
+ & \text{if } p = - \\
- & \text{if } p = + \\
\top & \text{if } p = \top \\
* & \text{if } p = *
\end{cases}
\]

\[ \text{Neg}[pe] = \{ c \mapsto \text{Neg}[p] \mid c \mapsto p \in pe \} \]

We will consider components unmentioned in a given polarity environment to always be implicitly mapped to \( \top \) No component mentioned in a formula is mapped to \( \top \); thus, \( \top \) won’t appear in polarity environments derived from actual formulas. Next, we must define a function that maps from an Alloy formula to this new domain of polarity environments. Our mapping function will be composed of the semantic functions \( PE \) and \( PF \) that provide
a mapping of Alloy expressions and formulas to polarity environments. Here’s $PE$:

\[
\begin{align*}
PE : & \text{AlloyExpression} \rightarrow \text{PEnv} \\
PE[c] & = \{c \mapsto +\} \\
PE[Exp_1 + Exp_2] & = PE[Exp_1] \cup PE[Exp_2] \\
PE[Exp_1 - Exp_2] & = PE[Exp_1] \cup \neg PE[Exp_2] \\
PE[Exp_1 \& \ Exp_2] & = PE[Exp_1] \cup PE[Exp_2] \\
PE[\{v : t|F\}] & = PF[F]
\end{align*}
\]

The polarities in an expression’s polarity environment are set to maximize the number of atoms in the concrete value of the expression. Thus, an expression’s polarity environment denotes how increasing or decreasing values of individual components can increase the value of the whole expression. For every component with polarity $+$ in the polarity environment of an expression, adding atoms to the value of that component will cause there to be more atoms in the value of expression. Conversely, for every component mapped to the polarity $-$ in an expression, adding atoms to the value of that component will decrease the number of atoms in the value of the expression. We can formalize this notion as follows:

Assuming the existence of the function:

\[
E[exp] = \text{set of all possible assignments of values to the components mentioned in the expression } exp
\]

we assert the following properties:

For all expressions $e$, and all components $c$ of $e$
This can be shown for each case of $PE$. For instance, consider the expression $a - b$ whose polarity environment is $\{a \mapsto +, b \mapsto -\}$. This polarity environment means that by adding elements to the set $a$, and by subtracting elements from the set $b$, we can make the expression $a - b$ have at least as many atoms as it originally had. A formal proof follows in Section 3.2.2.

The next part of our mapping is the semantic function $PF$ which provides a mapping of Alloy formulas:

$$PF : AlloyFormula \rightarrow PEnv$$

$$PF[Exp_1 \text{ in } Exp_2] = Neg[PE[Exp_1]] \cup PE[Exp_2]$$

$$PF[Form_1 \text{ & Form}_2] = PF[Form_1] \cup PF[Form_2]$$

$$PF[\text{not } Form_1] = Neg[PF[Form_1]]$$

$$PF[\text{all } v : D \mid Form_1] = PF[Form_1]$$

The meaning of the polarity environment of a formula is that the polarities of the components in the formula are set to “maximize” the truth of the formula. By this, we mean that polarities indicate how the values of components is an assignment can be changed so as to insure that if the original assignment satisfied the formula, the changed assignment will satisfy the formula. More formally, we claim that the following properties are true of polarity environments of formulas:

Assuming the existence of the function:
F[form] = set of all possible assignments of values to the components mentioned in the formula form

For all formulas f, and all components c of f

\[ PF[f](c) = + \rightarrow \forall m \in F[f], \forall v \rightarrow m(c). M[[f]]m \rightarrow M[[f]]m[c \rightarrow v] \]  \hspace{1cm} (P_3) 

\[ PF[f](c) = - \rightarrow \forall m \in F[f], \forall v \rightarrow m(c). M[[f]]m[c \rightarrow v] \rightarrow M[[f]]m \]  \hspace{1cm} (P_4) 

As an example of these properties, consider the formula s in T whose polarity environment is \{s \rightarrow -, T \rightarrow +\}. If there is an assignment of atoms to s and T, and we add atoms to T, there will still be an assignment. Conversely, if we add atoms to s, there might no longer be an assignment. A formal proof of these claims follows in Section 3.2.2.

Once we have created polarity environments and established their properties, we can start to use polarity environments to accomplish higher level tasks. As mentioned earlier, one of the key benefits that polarity environments offer is that they allow us to do a partial analysis of Alloy formulas that yields a partial assignment of values to components. We will formalize this notion by defining a function A that maps from polarity environments to partial assignments of values to components.

First, however, we need to define some useful constant values: Un, the universal relation, and {} the empty relation. Let us also choose to interpret Un as referring to the universal set when assigned to a set component and let {} refer to the empty set when assigned to a set component. This nuance could be easily achieved by using two different constants, but for the sake of uniformity, we'll just overload Un and {}.
Additionally, we have to decide how to deal with components whose polarities are *. In order to give these components a mapped Value, we’ll first have to extend our notion of Value to include an unknown Value, or \( \bot_{Value} \). In doing this, we extend our set Value into a partially ordered domain \( Value_{\bot} \):

\[
Value_{\bot} : \text{Value} \cup \bot_{Value}
\]

and with ordering property that:

\[
\forall v \in Value \cdot \bot_{Value} \sqsubseteq v
\]

The following order rule over Assignment follows from this partial order over \( Value \):

\[
\forall a_1, a_2 \in Assignment \cdot (\forall c \in (\text{dom } a_1 \cup \text{dom } a_2) \cdot a_1(c) \sqsubseteq a_2(c)) \iff a_1 \sqsubseteq a_2
\]

We can now fully define our partial assignment function \( A \) as follows:

\[
A : PEnv \rightarrow Assignment
\]

\[
A(p : PEnv) = \{c \mapsto \{\} \mid (c \mapsto -) \in p\} \cup \{c \mapsto \text{Un} \mid (c \mapsto +) \in p\} \\
\cup \{c \mapsto \bot_{Value} \mid (c \mapsto *) \in p\}
\]

For all components in a polarity environment mapped to + or -, \( A \) yields an assignment to \( \text{Un} \) or \( \{\} \). All components mapped to a polarity of * can’t be assigned to a useful value by \( A \), and are thus assigned to \( \bot_{Value} \).

It is our contention that \( A \) will provide a valid approximation to a legal satisfying assignment of an Alloy formula. Thus, as we’ll show, we can use \( A \), and the properties of polarity environments, to perform simplification of Alloy formulas, and to statically determine if two Alloy formulas are independent.
3.2.2 Soundness of the Polarity Analysis

Core Properties of Polarity Environments

Now that we have fully defined polarity analysis, we wish to prove its soundness. We’ll start by proving the basic properties of polarity environments introduced in Section 3.2.1.

Assuming the existence of the functions:

\[ E[exp] = \text{set of all possible assignments of values to the components mentioned in the expression } exp \]

\[ F[form] = \text{set of all possible assignments of values to the components mentioned in the formula } form \]

We assert the following properties:

For all expressions \( e \), and all components \( c \in e \)

\[ PE[e](c) = + \quad \forall m \in E[e], \forall v \supset m(c). X[e] m[c \mapsto v] \supseteq X[e] m \]  \hspace{1cm} (P_1)

\[ PE[e](c) = - \quad \forall m \in E[e], \forall v \supset m(c). X[e] m[c \mapsto v] \subseteq X[e] m \]  \hspace{1cm} (P_2)

For all formulas \( f \), and all components \( c \in f \)

\[ PF[f](c) = + \quad \forall m \in F[f], \forall v \supset m(c). M[f] m \rightarrow M[f] m[c \mapsto v] \]  \hspace{1cm} (P_3)

\[ PF[f](c) = - \quad \forall m \in F[f], \forall v \supset m(c). M[f] m[c \mapsto v] \rightarrow M[f] m \]  \hspace{1cm} (P_4)

We’ll prove \( P_1, P_2, P_3, \) and \( P_4 \) simultaneously by structural induction over Alloy expressions and formulas. For the sake of simplicity, we’ll consider the Alloy kernel language without functions or set comprehensions. Set comprehensions can be desugared into regular formulas.
with added state components. Functions are just generalizations of relations, and the argument we employ to prove that the properties of polarity environments hold over relational image, will apply equally as well to function application.

Thus, our proof of properties \( P_1, P_2, P_3, \) and \( P_4 \) will be over the following structures:

\[
\text{exp ::= \( c \mid \text{exp} + \text{exp} \mid \text{exp} \& \text{exp} \mid \text{exp} - \text{exp} \mid \text{exp} \cdot \text{exp} \mid \sim \text{exp} \mid + \text{exp} \)}
\]

\[
\text{form ::= \text{exp in exp} \mid \text{form \&\& form} \mid \text{not form} \mid \text{all } v : \text{DBAR form}}
\]

**Induction over all possible expressions:**

**Case 1. \( \text{exp ::= const} \) (where const is a constant)**
In this case, since we know that \( PE[\text{exp}] \) only maps const and \( PE[\text{exp}](\text{const}) = + \), we are only concerned with proving \( P_1 \), since the other properties are vacuously true.

Let \( m \) be an arbitrary assignment to \( \text{exp} \) and let \( v \) be an arbitrary superset of \( m(c) \):

We know that \( X[c] \ m = m(c) \), and similarly \( X[c] \ m[c \mapsto v] = v \)
Therefore \( X[c] m[c \mapsto v] \supseteq X[c] m \)
Thus, \( P_1 \) holds for this case.

**Case 2. \( \text{exp ::= exp}_1 + \text{exp}_2 \)**
By the inductive hypothesis, we know that \( P_1 \) and \( P_2 \) hold for \( \text{exp}_1 \) and \( \text{exp}_2 \).
We also know that \( PE[\text{exp}_1 + \text{exp}_2] = PE[\text{exp}_1] \cup PE[\text{exp}_2] \)

**Proof of \( P_1 \)**
Given the definition of \( \cup \),
\( PE[\text{exp}_1 + \text{exp}_2](c) \) will only map to + in the following cases:

**Case 2a. \( PE[\text{exp}_1](c) = + \) and \( PE[\text{exp}_2](c) \in \{+, \top \} \)**
Let \( m \) be an arbitrary assignment to \( \text{exp} \) and let \( v \) be an arbitrary superset of \( m(c) \).
Since \( PE[\text{exp}_1](c) = + \), and \( P_1 \) holds for \( \text{exp}_1 \),
\( X[\text{exp}_1] m[c \mapsto v] \supseteq X[\text{exp}_1] m \).

Similarly, if \( PE[\text{exp}_2](c) = + \) then since by the inductive hypothesis \( P_1 \) holds for \( \text{exp}_2 \),
\( X[\text{exp}_2] m[c \mapsto v] \supseteq X[\text{exp}_2] m \)
if however \( PE[\text{exp}_2](c) = \top \) then \( c \) is unmentioned in \( \text{exp}_2 \) and
Thus we are guaranteed that
\[ X[\text{exp}_1][c \mapsto v] \cup X[\text{exp}_2][c \mapsto v] \supseteq X[\text{exp}_1]m \cup X[\text{exp}_2]m \]
From the definition of \( X \) we can reduce this to
\[ X[\text{exp}_1 + \text{exp}_2][c \mapsto v] \supseteq X[\text{exp}_1 + \text{exp}_2]m \]
which proves that \( P_1 \) holds for this case.

**Case 2b.** \( PE[\text{exp}_1](c) \in \{+, \top\} \) and \( PE[\text{exp}_2] = + \)
This is exactly symmetric to case 2a.

**Proof of \( P_2 \)**

Given the definition of \( \sqcup \)
\( PE[\text{exp}_1 + \text{exp}_2](c) \) will only map to \( \bot \) in the following cases:

**Case 2c.** \( PE[\text{exp}_1](c) = - \) and \( PE[\text{exp}_2] \in \{-, \top\} \)
Let \( m \) be an arbitrary assignment to \( \text{exp} \) and let \( v \) be an arbitrary super-set of \( m(c) \):
Since \( PE[\text{exp}_1](c) = - \), and \( P_2 \) holds for \( \text{exp}_1 \)
\[ X[\text{exp}_1][c \mapsto v] \subseteq X[\text{exp}_1]m. \]

Similarly, if \( PE[\text{exp}_2](c) = - \) then by the inductive hypothesis,

since \( P_2 \) holds for \( \text{exp}_2 \),
\[ X[\text{exp}_2][c \mapsto v] \subseteq X[\text{exp}_2]m. \]
if however \( PE[\text{exp}_2](c) = \top \) then
\( c \) is unmentioned in \( \text{exp}_2 \) and
\[ X[\text{exp}_2][c \mapsto v] = X[\text{exp}_2]m. \]

Thus we are guaranteed that
\[ X[\text{exp}_1][c \mapsto v] \cup X[\text{exp}_2][c \mapsto v] \subseteq X[\text{exp}_1]m \cup X[\text{exp}_2]m \]
From the definition of \( X \) we can reduce this to
\[ X[\text{exp}_1 + \text{exp}_2][c \mapsto v] \subseteq X[\text{exp}_1 + \text{exp}_2]m \]
which proves that \( P_2 \) holds for this case.

**Case 2d.** \( PE[\text{exp}_1](c) \in \{-, \top\} \) and \( PE[\text{exp}_2] = - \)
This is exactly symmetric to case 2c.

**Case 3.** \( \text{exp} := \text{exp}_1 - \text{exp}_2 \)
We know that \( P_1 \) and \( P_2 \) hold for \( \text{exp}_1 \) and \( \text{exp}_2 \).
We also know that \( PE[\text{exp}_1 - \text{exp}_2] = PE[\text{exp}_1] \sqcup Neg[PE[\text{exp}_2]] \)

**Proof of \( P_1 \)**

Given the definition of the \( \sqcup \), and \( Neg \) which just flips polarities:

\( PE[\text{exp}_1 - \text{exp}_2](c) \) will only map to + in one of the following cases:

**Case 3a.** \( PE[\text{exp}_1](c) = + \) and \( PE[\text{exp}_2] \in \{-, \top\} \)

Let \( m \) be an arbitrary assignment to \( \text{exp} \) and let \( v \) be an arbitrary super-set of \( m(c) \):

In this case, since \( PE[\text{exp}_1](c) = + \), and \( P_1 \) holds for \( \text{exp}_1 \) We know that \( X[\text{exp}_1]m[c \mapsto v] \supseteq X[\text{exp}_1]m \)

If \( PE[\text{exp}_2](c) = - \) then by the inductive hypothesis

\( X[\text{exp}_2]m[c \mapsto v] \supseteq X[\text{exp}_2]m \).

If however \( PE[\text{exp}_2](c) = \top \) then

\( c \) is unmentioned in \( \text{exp}_2 \) and

\( X[\text{exp}_2]m[c \mapsto v] = X[\text{exp}_2]m \).

Thus we are guaranteed that

\( X[\text{exp}_1]m[c \mapsto v] \setminus X[\text{exp}_2]m[c \mapsto v] \supseteq X[\text{exp}_1]m \setminus X[\text{exp}_2]m \)

From the definition of \( X \) we can reduce this to

\( X[\text{exp}_1 - \text{exp}_2]m[c \mapsto v] \supseteq X[\text{exp}_1 - \text{exp}_2]m \)

which proves that \( P_1 \) holds for this case.

**Case 3b.** \( PE[\text{exp}_1](c) \in \{+, \top\} \) and \( PE[\text{exp}_2] = - \) This is symmetric to case 3a.

**Proof of \( P_2 \)**

Given the definition of \( \sqcup \), and the \( Neg \) op which just flips polarities:

\( PE[\text{exp}_1 - \text{exp}_2](c) \) will only map to - in the following cases:

**Case 3c.** \( PE[\text{exp}_1](c) = - \) and \( PE[\text{exp}_2] \in \{+, \top\} \).

Let \( m \) be an arbitrary assignment to \( \text{exp} \) and let \( v \) be an arbitrary super-set of \( m(c) \):

In this case, since \( PE[\text{exp}_1](c) = - \), and \( P_2 \) holds for \( \text{exp}_1 \), we know that \( X[\text{exp}_1]m[c \mapsto v] \subseteq X[\text{exp}_1]m \).

if \( PE[\text{exp}_2](c) = + \) then by the inductive hypothesis we know

\( X[\text{exp}_2]m[c \mapsto v] \supseteq X[\text{exp}_2]m \).

if however \( PE[\text{exp}_2](c) = \top \) then

33
c is unmentioned in $exp_2$ and
\[ X[exp_2]m[c \mapsto v] = X[exp_2]m. \]

Thus, we are guaranteed that:
\[ X[exp_1]m[c \mapsto v] \setminus X[exp_2]m[c \mapsto v] \subseteq X[exp_1]m \setminus X[exp_2]m \]

From the definition of \(X\) we can reduce this to
\[ X[exp_1 - exp_2]m[c \mapsto v] \subseteq X[exp_1 - exp_2]m \]
which proves that \(P_2\) holds for this case.

**Case 3d.** \(PE[exp_1](c) \in \{-, \top\}\) and \(PE[exp_2] = +\)
This is symmetric to case 3c

**Case 4.** \(exp ::= exp_1 & exp_2\)

By the inductive hypothesis, we know that \(P_1\) and \(P_2\) hold for \(exp_1\) and \(exp_2\).
We also know that \(PE[exp_1 \& exp_2] = PE[exp_1] \sqcup PE[exp_2]\)

**Proof of \(P_1\)**

Given the definition of \(\sqcup\)
\(PE[exp_1 \& exp_2](c)\) will only map to + in the following cases:

**Case 4a.** \(PE[exp_1](c) = +\) and \(PE[exp_2] \in \{+, \top\}\)

Let \(m\) be an arbitrary assignment to \(exp\) and let \(v\) be an arbitrary super-set of \(m(c)\)
since \(PE[exp_1](c) = +\), and \(P_1\) holds for \(exp_1\)
\[ X[exp_1]m[c \mapsto v] \supseteq X[exp_1]m \]

Similarly, if \(PE[exp_2](c) = +\) then since by the inductive hypothesis
\(P_1\) holds for \(exp_2\),
\[ X[exp_2]m[c \mapsto v] \supseteq X[exp_2]m \]
if however \(PE[exp_2](c) = \top\) then \(c\) is unmentioned in \(exp_2\) and
\[ X[exp_2]m[c \mapsto v] = X[exp_2]m. \]

Thus we are guaranteed that
\[ X[exp_1]m[c \mapsto v] \cap X[exp_2]m[c \mapsto v] \supseteq X[exp_1]m \cap X[exp_2]m \]
From the definition of \(X\) we can reduce this to
\[ X[exp_1 \& exp_2]m[c \mapsto v] \supseteq X[exp_1 \& exp_2]m \]
which proves that \(P_1\) holds for this case.

**Case 4b.** \(PE[exp_1](c) \in \{+, \top\}\) and \(PE[exp_2] = +\)
This is exactly symmetric to case 4a.
Proof of $P_2$
Given the definition of $\sqcup$

$PE[\text{exp}_1 \& \text{exp}_2](c)$ will only map to $-$ in the following cases:

Case 4c. $PE[\text{exp}_1](c) = -$ and $PE[\text{exp}_2] \in \{-, T\}$
Let $m$ be an arbitrary assignment to $\text{exp}$ and let $v$ be an arbitrary superset of $m(c)$:

since $PE[\text{exp}_1](c) = -$ and $P_2$ holds for $\text{exp}_1$

$X[\text{exp}_1]m[c \mapsto v] \subseteq X[\text{exp}_1]m$.

Similarly, if $PE[\text{exp}_2](c) = -$ then by the inductive hypothesis,

since $P_2$ holds for $\text{exp}_2$,

$X[\text{exp}_2]m[c \mapsto v] \subseteq X[\text{exp}_2]m$.

if however $PE[\text{exp}_2](c) = T$ then $c$ is unmentioned in $\text{exp}_2$ and

$X[\text{exp}_2]m[c \mapsto v] = X[\text{exp}_2]m$.

Thus we are guaranteed that

$X[\text{exp}_1]m[c \mapsto v] \cap X[\text{exp}_2]m[c \mapsto v] \subseteq X[\text{exp}_1]m \cap X[\text{exp}_2]m$.

From the definition of $X$ we can reduce this to

$X[\text{exp}_1 \& \text{exp}_2]m[c \mapsto v] \subseteq X[\text{exp}_1 \& \text{exp}_2]m$.

which proves that $P_2$ holds for this case.

Case 4d. $PE[\text{exp}_1](c) \in \{-, T\}$ and $PE[\text{exp}_2] = -$.
This is exactly symmetric to case 4c.

Case 5. $\text{exp} ::= \text{exp}_1 . \text{exp}_2$

By the inductive hypothesis, we know that $P_1$ and $P_2$ hold for $\text{exp}_1$ and $\text{exp}_2$.

We also know that $PE[\text{exp}_1 \cdot \text{exp}_2] = PE[\text{exp}_1] \sqcup PE[\text{exp}_2]$.

For the purpose of this case, let's first define a function:

$\text{relValue} : \text{Value} \rightarrow \text{Value} \rightarrow \text{Value}$

$\text{relValue}[v_1, v_2] = \{(x, \text{unit}) \exists y \cdot (y, \text{unit}) \in v_1 \land (y, x) \in v_2\}$

It follows from the definition of $\text{Value}$ that $\text{relValue}$ has the following property we'll call $RV$:

$\forall v'_1 \supseteq v_1, \forall v'_2 \supseteq v_2 : \text{relValue}[v'_1, v'_2] \supseteq \text{relValue}[v_1, v_2]$.

The proof of this property is simple; if a pair $(atom_1, \text{unit})$ exists in a Value $v$, and a pair $(atom_1, atom_2)$ exists in a Value $w$, and those mappings must exist in all supersets of $v$ and $w$.

Proof of $P_1$

Given the definition of $\sqcup$

$PE[\text{exp}_1, \text{exp}_2](c)$ will only map to $+$ in the following cases:
Case 5a. $PE[exp_1](c) = +$ and $PE[exp_2] \in \{+, \top\}$
Let $m$ be an arbitrary assignment to $exp$ and let $v$ be an arbitrary superset of $m(c)$
since $PE[exp_1](c) = +$, and $P_1$ holds for $exp_1$
$X[exp_1]m[c \mapsto v] \supseteq X[exp_1]m$

Similarly, if $PE[exp_2](c) = +$ then by the inductive hypothesis
$P_1$ holds for $exp_2$,
$X[exp_2]m[c \mapsto v] \supseteq X[exp_2]m$
if however $PE[exp_2](c) = \top$ then $c$ is unmentioned in $exp_2$ and
$X[exp_2]m[c \mapsto v] = X[exp_2]m$.
Thus we are guaranteed that
$X[exp_1]m[c \mapsto v] \supseteq X[exp_1]m \land X[exp_2]m[c \mapsto v] \supseteq X[exp_2]m$
From property $RV$, this means that:
$relValue[X[exp_1]m[c \mapsto v],X[exp_2]m[c \mapsto v]] \supseteq relValue[X[exp_1]m,X[exp_2]m]$

From the definition of $X$ we can reduce this to
$X[exp_1,exp_2]m[c \mapsto v] \supseteq X[exp_1,exp_2]m$
which proves that $P_1$ holds for this case.

Case 5b. $PE[exp_1](c) \in \{+, \top\}$ and $PE[exp_2] = +$
This is exactly symmetric to case 5a.

Proof of $P_2$
Given the definition of $\sqcup$
$PE[exp_1,exp_2](c)$ will only map to $-$ in the following cases:

Case 5c. $PE[exp_1](c) = -$ and $PE[exp_2] \in \{-, \top\}$
Let $m$ be an arbitrary assignment to $exp$ and let $v$ be an arbitrary superset of $m(c)$:
since $PE[exp_1](c) = -$, and $P_2$ holds for $exp_1$
$X[exp_1]m[c \mapsto v] \subseteq X[exp_1]m$.
Similarly, if $PE[exp_2](c) = -$ then by the inductive hypothesis,
since $P_2$ holds for $exp_2$,
$X[exp_2]m[c \mapsto v] \subseteq X[exp_2]m$.
if however $PE[exp_2](c) = \top$ then
$c$ is unmentioned in $exp_2$ and
Thus we are guaranteed that
\[ X[\mathit{exp}_1]m[c \mapsto v] \subseteq X[\mathit{exp}_1]m \land X[\mathit{exp}_2]m[c \mapsto v] \subseteq X[\mathit{exp}_2]m \]
From property \(RV\), this means that:

\[
\text{relValue}(X[\mathit{exp}_1]m[c \mapsto v], X[\mathit{exp}_2]m[c \mapsto v]) 
\subseteq \text{relValue}(X[\mathit{exp}_1]m, X[\mathit{exp}_2]m)
\]

From the definition of \(X\) we can reduce this to
\[ X[\mathit{exp}_1, \mathit{exp}_2]m[c \mapsto v] \subseteq X[\mathit{exp}_1, \mathit{exp}_2]m \]
which proves that \(P_2\) holds for this case.

**Case 5d.** \(PE[\mathit{exp}_1](c) \in \{-, \top\} \land PE[\mathit{exp}_2] = -\)
This is exactly symmetric to case 5c.

**Case 6.** \(\mathit{exp} ::= \sim \mathit{exp}_1\)
By the inductive hypothesis, we know that \(P_1\) and \(P_2\) hold for \(\mathit{exp}_1\).
We also know that \(PE[\mathit{exp}] = PE[\mathit{exp}_1]\)

Let's define a function:
\[
\text{transposeValue} : \text{Value} \rightarrow \text{Value}
\]
\[
\text{transposeValue}[V] = (x, y) \mid (y, x) \in V
\]

We assert that \(\text{transposeValue}\) has the following property, we'll call \((TV)\):
\[
\forall v'. v'. \text{transposeValue}[v'] \supseteq \text{transposeValue}[v]
\]
This property is true because \(\text{transposeValue}\) just flips the pairings in a \(\text{Value}\), not adding or taking away pairings. Thus, all pairings in \(\text{transposeValue}[v]\) will be in \(\text{transposeValue}[v']\) for all supersets \(v'\) of \(v\).

**Proof of \(P_1\)**
\(PE[\sim \mathit{exp}_1](c)\) will only map to \(+\) if:

**Case 6a.** \(PE[\mathit{exp}_1](c) = +:\)

Let \(m\) be an arbitrary assignment to \(\mathit{exp}\) and let \(v\) be an arbitrary superset of \(m(c)\)

since \(PE[\mathit{exp}_1](c) = +\), and \(P_1\) holds for \(\mathit{exp}_1\)

\[ X[\mathit{exp}_1]m[c \mapsto v] \supseteq X[\mathit{exp}_1]m \]

From \(TV\), we therefore know that:
\[
\text{transposeValue}[X[\mathit{exp}_1]m[c \mapsto v]] \supseteq \text{transposeValue}[X[\mathit{exp}_1]m]
\]

From the definition of \(X\) we can reduce this to
\[ X[\sim \mathit{exp}_1]m[c \mapsto v] \supseteq X[\sim \mathit{exp}_1]m \]
which proves that \(P_1\) holds for this case.
Proof of $P_2$
$PE[exp_1](c)$ will only map to $-$ in the following case:

Case 6b. $PE[exp_1](c) = -$ 

Let $m$ be an arbitrary assignment to $exp$ and let $v$ be an arbitrary super-set of $m(c)$:

since $PE[exp_1](c) = -$, and $P_2$ holds for $exp_1$

$X[exp_1][m[c \mapsto v]] \subseteq X[exp_1][m]$.

From $TV$, we therefore know that:

$transposeValue[X[exp_1][m[c \mapsto v]]] \subseteq transposeValue[X[exp_1][m]]$

From the definition of $X$ we can reduce this to

$X[\sim exp_1][m[c \mapsto v]] \subseteq X[\sim exp_1][m]$

which proves that $P_2$ holds for this case.

Case 7. $exp ::= +exp_1$

By the inductive hypothesis, we know that $P_1$ and $P_2$ hold for $exp_1$

We also know that $PE[exp] = PE[exp_1]$

We observe the following property of transitive closure of relations:

$\forall (r,s) \in \{(r_0,r_1)|X[r_0] \supset X[r_1]|a)(X[r]a \supset X[+s]a$ 

This property is true because a relation’s superset’s transitive closure will always be a superset of the relation’s transitive closure.

Proof of $P_1$

$PE[+exp_1](c)$ will only map to $+$ if:

Case 7a. $PE[exp_1](c) = +$:

Let $m$ be an arbitrary assignment to $exp$ and let $v$ be an arbitrary super-set of $m(c)$

since $PE[exp_1](c) = +$, and $P_1$ holds for $exp_1$

We know that $X[exp_1][m[c \mapsto v]] \supseteq X[exp_1][m]$

From the nature of transitive closure we can assert:

$X[+exp_1][m[c \mapsto v]] \supseteq X[+exp_1][m]$

which proves that $P_1$ holds for this case.

Proof of $P_2$

$PE[exp_1](c)$ will only map to $-$ in the following case:

Case 7b. $PE[exp_1](c) = -$
Let \( m \) be an arbitrary assignment to \( \text{exp} \) and let \( v \) be an arbitrary super-set of \( m(c) \):

since \( PE[\text{exp}_1](c) = - \), and \( P_2 \) holds for \( \text{exp}_1 \)

\[ X[\text{exp}_1]m[c \mapsto v] \subseteq X[\text{exp}_1]m. \]

From the nature of transitive closure, we can assert:

\[ X[+\text{exp}_1]m[c \mapsto v] \subseteq X[+\text{exp}_1]m \]

which proves that \( P_2 \) holds for this case.

\[ \square \]

**Induction over all possible formulas**

**Case 1. form ::= \text{exp}_1 \text{ in } \text{exp}_2**

From the definition of \( PE \)

We know that \( PF[\text{exp}_1 \text{ in } \text{exp}_2] = Neg[PE[\text{exp}_1]] \sqcup PE[\text{exp}_2] \)

**Proof of \( P_3 \)**

Assume \( PF[\text{exp}_1 \text{ in } \text{exp}_2](c) = + \)

Given definition of \( \sqcup \), this will only happen when

**Case 1a. \( PE[\text{exp}_1](c) = - \) \&\& \( PE[\text{exp}_2](c) \in \{+, \top\} \)**

Let \( m \) be an arbitrary assignment to \( \text{exp} \) and let \( v \) be an arbitrary super-set of \( m(c) \)

We know that \( X[\text{exp}_1]m[c \mapsto v] \subseteq X[\text{exp}_1]m \)

Similarly, we know that

if \( PE[\text{exp}_2](c) = + \) then by the inductive hypothesis \( P_1 \) holds for \( \text{exp}_2 \) and

\[ X[\text{exp}_2]m[c \mapsto v] \supseteq X[\text{exp}_2]m \]

if however \( PE[\text{exp}_2] = \top \)

\[ X[\text{exp}_2]m[c \mapsto v] = X[\text{exp}_2]m \]

Thus, since \( X[\text{exp}_2]m[c \mapsto v] \supseteq X[\text{exp}_2]m \)

and \( X[\text{exp}_1]m[c \mapsto v] \subseteq X[\text{exp}_1]m \)

we can always say that:

\[ X[\text{exp}_1]m \subseteq X[\text{exp}_2]m \rightarrow X[\text{exp}_1]m[c \mapsto v] \subseteq X[\text{exp}_2]m[c \mapsto v] \]

which means:

\( M[\text{exp}_1 \text{ in } \text{exp}_2]m \rightarrow M[\text{exp}_1 \text{ in } \text{exp}_2]m[c \mapsto v] \)

thus, \( P_3 \) applies in this case

**Case 1b. \( PE[\text{exp}_1](c) \in \{-, \top\} \) \&\& \( PE[\text{exp}_2](c) = + \) is symmetric to case 1a.**

\[ \square \]
Proof of $P_4$
Assume $PF[\text{exp}_1 \text{ in } \text{exp}_2](c) = -$
Given definition of $\sqcup$, this will only happen in the following cases:

Case 1c. $PE[\text{exp}_1] = + \land \land PE[\text{exp}_2] \in \{-, \top\}$
Let $m$ be an arbitrary assignment to $\text{exp}$ and let $v$ be an arbitrary super-set of $m(c)$
By the inductive hypothesis we know that $P_1$ holds for $\text{exp}_1$ and $\text{exp}_1 \in m(c)$
Similarly we know that

if $PE[\text{exp}_2] = -$ then by the inductive hypothesis and $P_2$
$X[\text{exp}_2]m[c \mapsto v] \subseteq X[\text{exp}_2]m$
if $PE[\text{exp}_2] = \top$
$X[\text{exp}_2]m[c \mapsto v] = X[\text{exp}_2]m$

Thus, since
$X[\text{exp}_2]m[c \mapsto v] \subseteq X[\text{exp}_2]m$
and $X[\text{exp}_1]m[c \mapsto v] \supseteq X[\text{exp}_1]m$
we can always say that:
$X[\text{exp}_1]m[c \mapsto v] \subseteq X[\text{exp}_2]m[c \mapsto v] \rightarrow X[\text{exp}_1]m \subseteq X[\text{exp}_2]m$
which means:
$M[\text{exp}_1 \text{ in } \text{exp}_2]m[c \mapsto v] \rightarrow M[\text{exp}_1 \text{ in } \text{exp}_2]m$
thus, $P_4$ applies in this case

Case 1d. $PE[\text{exp}_1](c) \in \{+, \top\} \land \land PE[\text{exp}_2](c) = -$ is symmetric to case 1c.

Case 2. $\text{form} ::= \text{form}_1 \land \land \text{form}_2$
From the definition of $PF$
We know that $PF[[\text{form}_1 \land \land \text{form}_2]] = PF[[\text{form}_1]] \sqcup PF[[\text{form}_2]]$

Proof of $P_3$
Assume $PF[[\text{form}_1 \land \land \text{form}_2]](c) = +$
Given definition of $\sqcup$, this will only happen when

Case 2a. $PF[[\text{form}_1]](c) = + \land \land PF[[\text{form}_2]](c) \in \{+, \top\}$
Let $m$ be an arbitrary assignment to $\text{form}$ and let $v$ be an arbitrary super-set of $m(c)$
We know that $M[[\text{form}_1]]m \rightarrow M[[\text{form}_1]]m[c \mapsto v]$ Similarly, we know that
If $PF[[\text{form}_2]](c) = +$ then by the inductive hypothesis $P_3$ holds for $\text{form}_2$ and
\[ M[form_2]m \rightarrow M[form_2]m[c \mapsto v] \]

if however \( PF[form_2](c) = \top \) then \( c \) is unmentioned in \( form_2 \) and

\[ M[form_2]m \leftrightarrow M[form_2]m[c \mapsto v] \]

Thus, since \( M[form_2]m \rightarrow M[form_2]m[c \mapsto v] \)
and \( M[form_1]m \rightarrow M[form_1]m[c \mapsto v] \)
we can always say that:

\[ M[form_1]m \land M[form_2]m \rightarrow M[form_1]m[c \mapsto v] \land M[form_2]m[c \mapsto v] \]

which means:

\[ M[form_1 \land form_2]m \rightarrow M[form_1 \land form_2]m[c \mapsto v] \]

thus, \( P_3 \) applies in this case

Case 2b. \( PF[form_1](c) \in \{+, \top\} \) \& \& \( PF[form_2](c) = + \) is symmetric to case 2a.

\[ \square \]

Proof of \( P_4 \)
Assume \( PF[form_1 \land form_2](c) = - \)
Given definition of \( \sqcup \), this will only hapPF\( n \) in the following cases:

Case 2c. \( PF[form_1](c) = - \) \& \& \( PF[form_2](c) \in \{-, \top\} \)
Let \( m \) be an arbitrary assignment to \( form \) and let \( v \) be an arbitrary su-
perset of \( m(c) \)
By the inductive hypothesis we know that \( P_4 \) holds for \( form_1 \) and
\n\[ M[form_1]m[c \mapsto v] \rightarrow M[form_1]m \]

Similarly we know that

if \( PF[form_2](c) = - \) then by the inductive hypothesis and \( P_4 \)
\[ M[form_2]m[c \mapsto v] \rightarrow M[form_2]m \] and

if \( PF[form_2](c) = \top \)
\[ M[form_2]m[c \mapsto v] \leftrightarrow M[form_2]m \]

Thus, since
\[ M[form_2]m[c \mapsto v] \rightarrow M[form_2]m \]
and\( M[form_1]m[c \mapsto v] \rightarrow M[form_1]m \)
we can always say that:
\[ M[form_1]m[c \mapsto v] \land M[form_2]m[c \mapsto v] \rightarrow M[form_1]m \land M[form_2]m \]
which means:
\[ M[form_1 \land form_2]m[c \mapsto v] \rightarrow M[form_1 \land form_2]m \]
thus, \( P_4 \) applies in this case

Case 2d. \( PF[form_1](c) \in \{-, \top\} \) \& \& \( PF[form_2](c) = - \) is symmetric to case 2c.

\[ \square \]
Case 3. form ::= not form

From the definition of PF
We know that PF[not form] = Neg[PF[form]]

Proof of P₃
Assume PF[not form] = +
Given definition of Neg, this will only happen when

Case 3a. PF[form](c) = −
Let m be an arbitrary assignment to form and let v be an arbitrary superset of m(c)

By the inductive hypothesis we know that P₄ holds for form₁ and
We know that M[form₁]m[c → v] → M[form₁]m
which is equivalent to:
−M[form₁]m[c → v] → −M[form₁]m[c → v]
which, from the definition of M means that:
M[not form₁]m[c → v] → M[not form₁]m[c → v]
thus, P₃ applies in this case

Proof of P₄
Assume PF[not form] = −
Given definition of Neg, this will only happen in the following case:

Case 3b. PF[form](c) = +
Let m be an arbitrary assignment to form and let v be an arbitrary superset of m(c)
By the inductive hypothesis we know that P₃ holds for form₁ and
M[form₁]m[c → v] → M[form₁]m[c → v]
which is equivalent to:
−M[form₁]m[c → v] → −M[form₁]m
which, from the definition of M means that:
M[not form₁]m[c → v] → −M[not form₁]m
Thus, P₄ applies in this case

Case 4. form ::= all v : t BAR form

From the definition of PF
We know that PF[all v : t BAR form] = PF[form]

Proof of P₃
Assume PF[all v : t BAR form](c) = +
This will only happen when
Case 4a. \( PF[form_1](c) = + \)
Let \( m \) be an arbitrary assignment to \( form \) and let \( w \) be an arbitrary superset of \( m(c) \)

By the inductive hypothesis we know that \( P_3 \) holds for \( form_1 \) and
We know that \( M[[form_1]]m \rightarrow M[[form_1]]m(m(c)/w) \)
Since this holds for any arbitrary \( m \) we can be guaranteed that this will hold for any assignment \( m \oplus v \mapsto x \) where \( v \neq c \).

More formally, it follows that:

\[
\forall m \in F[form_1], \forall w \supseteq m(c), \forall v \neq c, \forall x \in \text{Value} \\
M[[form_1]](m \oplus v \mapsto x) \rightarrow M[[form_1]](m(m(c)/w) \oplus v \mapsto x)
\]

From this we can say that for our arbitrary \( m \) and \( w \), and for some type \( t \), and some variable \( v \in F, v \neq c \):

\[
\wedge \{M[F](m \oplus v \mapsto x) \mid (x, \text{unit}) \in m(t)\} \rightarrow \\
\wedge \{M[F](m(m(c)/w) \oplus v \mapsto x) \mid (x, \text{unit}) \in m(t)\}
\]

From the definition of \( M \), this means that:

\[
M[[\forall v : t \text{ BAR } form_1|m \rightarrow M[[\forall v : t \text{ BAR } form_1]]m(m(c)/w)
\]

Thus, \( P_3 \) holds in this case.

\[\blacksquare\]

Proof of \( P_4 \)
Assume \( PF[[\forall v : t \text{ BAR } form_1]](c) = - \)
This will only happen when

Case 4b. \( PF[[form_1]](c) = - \)
Let \( m \) be an arbitrary assignment to \( form \) and let \( w \) be an arbitrary superset of \( m(c) \)

By the inductive hypothesis we know that \( P_4 \) holds for \( form_1 \) and
We know that \( M[[form_1]]m(m(c)/w) \rightarrow M[[form_1]]m \)
Since this holds for any arbitrary \( m \) we can be guaranteed that this will hold for any assignment \( m \oplus v \mapsto x \) where \( v \neq c \).

More formally, it follows that:

\[
\forall m \in F[form_1], \forall w \supseteq m(c), \forall v \neq c, \forall x \in \text{Value} \\
M[[form_1]](m(m(c)/w) \oplus v \mapsto x) \rightarrow M[[form_1]](m \oplus v \mapsto x)
\]
From this we can say that for our arbitrary \( m \) and \( w \), and for some type \( t \), and some variable \( v \in F, v \neq c \):

\[
\land \{ M[F](m(m(c)/w) \oplus v \mapsto x)|(x, \text{unit}) \in m(t) \} \rightarrow \\
\land \{ M[F](m \oplus v \mapsto x)|(x, \text{unit}) \in m(t) \}
\]

From the definition of \( M \), this means that:

\( M[\forall v : t \text{ BAR form}_1]m(m(c)/w) \rightarrow M[\forall v : t \text{ BAR form}_1]m \)

Thus, \( P_4 \) applies in this case.

**Resulting and Related Properties**

Now that we have shown polarity environments have the properties attributed to them above, we will prove a few useful corollaries of these properties.

Firstly, we wish to prove that the partial assignment based on polarity environments, derived from \( A \), is part of a valid assignment. More formally, we wish to assert that

**Claim 1.**

\[
\forall f \in \text{Formulas} \cdot (S[f] \neq \{\}) \rightarrow \exists a \in S[f] \cdot A(PF[f]) \subseteq a
\]

**Proof of Claim 1**

Let \( f \) be an arbitrary Alloy formula. We are concerned with the case that a satisfying assignment exists to \( f \), so let \( a \) be an arbitrary satisfying assignment to \( f \), and let \( pf = PF[f] \). For every component \( c \in \text{dom } pf \):

**Case 1.** if \( pf(c) = + \) we know from property \( P_3 \), that:

\[
\forall v \supseteq a(c). M[f]a \rightarrow M[f][a(a(c)/v)]
\]

We know by definition that \( Un \supseteq a(c) \). Thus:
Since, by construction, \( a \in S[f] \) we know that \( M[f]a \) is true and that means that \( M[f]a(a(c)/Un) \) is also true. Thus, since \( A(pf)(c) = Un \), we can say that \( A(pf) \) is equal to some legal concrete assignment at component \( c \), namely \( a(a(c)/Un) \).

Case 2. if \( pf(c) = - \) we know from property \( P_4 \), that:

\[
\forall m \in Assignment, \forall v \subseteq m(c). M[f]m[c \mapsto v] \rightarrow M[f]m
\]

Now we can observe a general principle of ordering. If we know that for some function \( f \):

\[
\forall v' \supseteq v. f(v') \rightarrow f(v)
\]

We can always pick an arbitrary \( V \) and \( V' \) such that \( V' \supseteq V \) and \( f(V') \rightarrow f(V) \).

If we find a \( V'' \), such that \( V \supseteq V'' \), then we know that \( f(V) \rightarrow f(V'') \). In fact, this is true for all \( V'' \subseteq V \), and we can conclude more generally that

\[
\forall v' \subseteq v. f(v) \rightarrow f(v')
\]

Applied to \( P_4 \) in our example, this means that:

\[
\forall m \in Assignment, \forall v \subseteq m(c). M[f]m \rightarrow M[f]m[c \mapsto v]
\]

and for our assignment \( a \):

\[
\forall v \subseteq a(c). M[f]a \rightarrow M[f]a(a(c)/v)
\]

We know by definition that \( \{\} \subseteq a(c) \). Thus:

\[
M[f]a \rightarrow M[f]a(a(c)/\{\})
\]

Since \( a \in S[f] \) we know that \( M[f]a \) is true and that means that \( M[f]a(a(c)/\{\}) \) is also true. Thus, since \( A(pf)(c) = \{\} \), we can say that \( A(pf) \) is equal to some
legal assignment at component $c$, namely $a(a(c)/\{\})$.

**Case 3.** if $pf(c) = *$ that $A(pf)(c) = \bot_{\text{Value}}$ and thus we know that for any arbitrary satisfying assignment $a$ to $f$, $A(pf) \sqsubseteq a$

Thus, if there is a satisfying assignment for a formula $f$, for all mentioned components $c \in f$, for all possible polarities of $c$, we have shown that there is some satisfying assignment $a$ such that $A(pf) \sqsubseteq a$.

\[\blacksquare\]

As mentioned in the introduction, the primary purposes of polarity analysis is to detect formula dependence. Namely for two formulas $f_1$ and $f_2$, we wish to detect the following property of two formulas:

\[\text{independent} : \text{Formula} \rightarrow \text{Formula} \rightarrow \text{Boolean}\]

\[\text{independent}[f_1, f_2 : \text{Formulas}] =\]

\[\left( \exists a_1, a_2 \in \text{Assignment} . \left( M[[f_1]] a_1 \land M[[f_2]] a_2 \right) \right) \rightarrow \left( \exists a_3 \in \text{Assignment} . M[[f_1 \land f_2]] a_3 \right)\]

The repercussions of this property are immense. If we are concerned with satisfying one formula $f_1$ conjoined to another formula $f_2$, with which it has no polarity conflicts, we can solve $f_1$ independently of $f_2$, and be assured that if $f_2$ is satisfiable, then our solution to $f_1$ will mean that there’s a solution to $f_2$. Specifically, in Alloy models, where invariants and state formulas are implicit conjoined to every analysis, we can use this property to eliminate an analysis’s dependence upon an imported formula, and perform the analysis
without considering the formula. Actually determining that two formulas are independent, from the definition of independent involves searching the space of possible assignments to find an \( a_1, a_2, \) and \( a_3 \) that satisfy the definition of independent. In Alcoa's case, where we wish to use the fact that two formulas are independent to simplify a formula before searching for an assignment, it would be ridiculous to search for an assignment so as to determine the independence of formulas.

Polarity analysis, however, gives us an inexpensive and easy, if incomplete, way of determining that two formulas are independent. Specifically, we maintain that if the polarity environments of two formulas don't conflict, then we can be sure that the formulas are independent. In order to prove this, we have to formalize what we mean by conflicting polarity environments. We start by defining the function:

\[
\text{conflict} : PEnv \rightarrow PEnv \rightarrow Boolean
\]

\[
\text{conflict}[p_1, p_2 : PEnv] = \exists_{c \in dom \ p_1 \cap dom \ p_2} \ (p_1 \cup p_2)(c) = *
\]

We wish to prove that conflict allows us to show that two formulas satisfy independent:

Claim 2.

\[
\forall f_1, f_2 \in \text{Formulas} . \ \neg \text{conflict}[PF[f_1], PF[f_2]] \rightarrow \text{independent}[f_1, f_2]
\]

Proof of Claim 2.

Let \( f_1, \) and \( f_2, \) be two arbitrary formulas.

We are interested in the case where

\[
\neg \text{conflict}[PF[f_1], PF[f_2]] \ \text{and} \ \exists_{a_1, a_2 \in Assignment} . \ (M[f_1] a_1 \ \land \ M[f_2] a_2)
\]
as Claim 2 is otherwise vacuously true.

We wish to show \( \exists a_3 \in \text{Assignment} . M[f_1 \land f_2] a_3 \) From the definition of conflict, we know that

\[
\forall c \in \text{dom } P(F[f_1] \cup P(F[f_2]))(c) \in \{+, -\}
\]

For these components we can create a new partial assignment \( a_3 \):

\[
a_3(c) = \begin{cases} 
\{\} & \text{if } (P(F[f_1] \cup P(F[f_2]))(c) = - \\
Un & \text{if } (P(F[f_1] \cup P(F[f_2]))(c) = + 
\end{cases}
\]

Now, if we apply this partial assignment to \( f_1 \land f_2 \), substituting \( \{\} \) for all components with a \(-\) polarity, and \( Un \) for all components with a \(+\) polarity, we'll be left with a formula where the only mentioned components have a polarity of \(*\). From the definition of conflict, we also know that these components are all mentioned exclusively in only one of the two formulas. Thus, we can express this altered form of \( f_1 \land f_2 \) as:

\( f'_1 \land f'_2 \)

where \( f'_1 \) only has components mentioned in \( f_1 \) and \( f'_2 \) only has components mentioned in \( f_2 \).

An assignment to \( f'_1 \land f'_2 \) will exist iff an assignment to \( f'_1 \) exists and an assignment to \( f'_2 \) exists. \( f'_1 \), however, is just \( f_1 \) with components replaced with constants derived from their polarities. Thus, since an assignment exists to \( f_1 \), the same assignment will satisfy \( f'_1 \) for all the remaining components. The same argument holds for \( f'_2 \). Thus, we can find a total assignment that will satisfy \( f_1 \land f_2 \):
\[ a_3(c) = \begin{cases} 
\emptyset & \text{if } (PF[f_1] \cup PF[f_2])(c) = - \\
Un & \text{if } (PF[f_1] \cup PF[f_2])(c) = + \\
v & \text{if } c \in (\text{dom}\,PF[f_1] - \text{dom}\,PF[f_2]) \land PF[f_1](c) = * \land a_1(c) = v \\
w & \text{if } c \in (\text{dom}\,PF[f_2] - \text{dom}\,PF[f_1]) \land PF[f_2](c) = * \land a_2(c) = w 
\end{cases} \]

Thus: \( \neg \text{conflict}[PF[f_1], PF[f_2]] \rightarrow \text{independent}[f_1, f_2] \)

3.2.3 Benefit of the Polarity Analysis

Now that we have established the soundness of polarity analysis, we will discuss its benefits. Polarity analysis has several different uses and is a powerful tool for analyzing Alloy specifications in several ways.

Polarity analysis was originally designed to detect dependencies between formulas in Alloy specifications. As demonstrated above, polarity analysis allows us to determine that two formulas are independent even if they share variables. This offers a tremendous advantage over simple sharing analysis. Since many specifications have disjoint paragraphs built on top of the same domains, different invariants and conditions will often mention the same basic domains and types, without necessarily conflicting. As explained before, when Alcoa analyzes any paragraph it must gather together the invariants and state formulas, which are implicitly included in every paragraph, to conjoin with the paragraphs formula. Using polarity analysis, Alcoa can determine that the analyzed formula is completely dependent.
from a given invariant or state formula, and then avoid conjoining it to the analyzed formula. By thus eliminating unnecessary invariants Alcoa is able to perform analyses that are more efficient, since the analyzed formula doesn’t include the eliminated invariants. Moreover, by not including state formulas that are irrelevant to the analysis of a paragraph, Alcoa can provide a resulting assignment that doesn’t include assignments to unmentioned state components. Thus, polarity analysis allows Alcoa to return an assignment to the slice of state relevant to a particular paragraph.

There is an additional peculiar aspect of this functionality that can be attributed to our definition of the independence in Section 3.2.2. Our definition takes the form of an implication: if there are separate satisfying assignments to two independent formulas, then there is a satisfying assignment to the conjunction of the two formulas. If there is no satisfying assignment to one of the formulas, then the right side of the implication is false, and the implication is vacuously true. In using the conflict function to detect independence, we don’t separately check to make sure that satisfactions of the tested formulas exist. Thus, it is possible for our method to determine that two formulas, \( f_1 \) and \( f_2 \) are independent, when one of them is unsatisfiable, and thus no solution exists to \( f_1 \land f_2 \). What our independence test does guarantee, is that if the conjunction of two independent formulas is unsatisfiable, then this unsatisfiability can be traced to one of the formulas, and isn’t the result of a conflict between the formulas. This guarantee, if somewhat complicated, is very powerful in determining conflicts between formulas and constraints.

Another use of polarity analysis is for more straightforward simplification. In claims 1 and 2 in Section 3.2.2, we showed that if a formula is satisfiable, then we can set fixed values for some of its components for which we are guaranteed to find a satisfaction if
one exists. Components mapped to + in a formula’s polarity environment can be set to \( Un \), and components mapped to - can be set to \( {} \), and we can be guaranteed to find a satisfying assignment to the formula if it is satisfiable. With this knowledge, we can actually substitute for the components in the formula with their respective constant values. After this substitution, we can then reduce the formula using simple constant folding and arrive at a smaller formula that we can then analyze again. Thus, we can iteratively compute polarities, substitute constants for components with + or - polarities, simplify the resulting formula and repeat. This technique reduces the formula size considerably and reduces the the number of components that need assignments. For some very simple and loosely constrained formulas and specifications, this simple technique can actually provide a complete assignment of values to components, where every component gets a value of \( Un \) or \( {} \).

### 3.3 Runtime Analysis

One of the most attractive features of polarity analysis is its efficiency. Polarity analysis can be performed from the bottom up on a Alloy formula in \( O(n) \) time where \( n \) is the size of the formula. Polarity environment based substitution and simplification of an Alloy formula can be also performed from the bottom up in linear time. When we perform the repeated polarity analysis, we would encounter the worst case if we could only substitute for one constant at a time. In this case the simplification would take \( O(cn) \) time where \( c \) is the number of Alloy components in the formula, and \( n \) is the formula size. This sharply contrasts with the time it would take to actually solve the Alloy formula, which is exponential in \( c \).

Detecting dependencies between different formulas is a simple matter of checking their
polarity environments for conflicts, over all the components in the polarity environments. This check will take $O(c)$ time where $c$ is the number of components in the polarity environments. When dealing with many formulas, we are interested not only in direct dependencies, but in transitive dependencies. If formula $f_1$ depends on $f_2$, and $f_2$ depends on $f_3$, then we must consider $f_1$ as depending on $f_3$. If we have $m$ formulas, it will take $O(n)$ time to create a polarity environment from each formula, where $n$ is the size of the formula, and $O(c)O(m^2)$ time to detect direct dependencies, where $c$ is the number of Alloy components mentioned in the formulas, and checking for conflicts takes $O(c)$ time. Thus checking for direct dependences will take $O(n)O(m) + (O(c)O(m^2))$ time. Once we have the direct dependency graph, find any transitive dependencies is a matter of taking its transitive closure. Taking the transitive closure of this graph will take $\theta(m^3)$ time [CLR 26.2]. Thus, calculating the transitive dependency graph among formulas, which can be used to determine which formulas have to be imported in a given analysis, takes $O(n)O(m) + (O(c)O(m^2)) + \theta(m^3)$ time. Since we introduce a separate state formula for every component, we can reduce this to $O(n)O(m) + O(m^3)$ time. Thus, in cubic time we are able to figure out formula dependencies and imports. This is far superior to the brute force method of trying to find assignments that indicate independence of formulas; each pairwise attempt to determine independence would be exponential in the number of mentioned components.
Chapter 4

Implementation of the Polarity Analysis

4.1 Overview

Polarity analysis was implemented in two phases within the context of Alcoa’s compiler infrastructure. Initially, polarity analysis was done strictly in one pass for the purpose of detecting inter-paragraph dependencies. Later, functionality was added to allow constant substitution, formula simplification, and iterative propagating polarity analysis in formulas. Since polarity analysis was implemented inside of Alcoa’s compiler infrastructure, a brief overview of Alcoa’s architecture will be necessary to explain the overall implementation of polarity analysis.
4.2 Alcoa’s Architecture and Initial Implementation

We implemented Alcoa in Java during the spring and summer of 1999. An object model of the core of our design is depicted in Figure 4-1. The main classes that make up the core of the design are the intermediate representation classes, that encapsulate the intermediate representation of a compiled Alcoa model. At a higher level of modularity, there are classes for each type of paragraph. InvSchema, DefSchema, AssertSchema, CondSchema, and StateSchema, are all subclasses of FormulaSchema which encapsulates the functionality common to all paragraphs. Common to all FormulaSchema objects is the formula field which holds a reference to the Alloy formula contained in each paragraph. This formula is actually a Formula object at the root of an intermediate level abstract syntax tree representing the formulas and expressions in the paragraph. The classes in the Formula and Expr class hierarchies represent the different types of the formulas and expressions in the Alloy intermediate representation. This intermediate representation (IR) infrastructure is used to perform simplifications of Alloy formulas, and ultimately to translate the Alloy formulas into boolean formulas that are then solved with a constraint satisfaction solver.

Apart from the IR classes themselves, there is a Dependency class, whose task it is to detect dependencies between the different paragraphs. It was in the context of this Dependency class and its helpers that the polarity analysis was implemented. At the time of its creation, every FormulaSchema object registers its formula with the Dependency object. This is done by wrapping the formula in a delegate, or Porter object, and invoking the Dependency object’s register method. Most paragraphs only register one porter. Invariants and definitions, however, whose formulas might have to be conjoined to a given formula in
their primed state, register two porters, one for their regular formula, and one for the primed version of their formula. **StateSchemas** have a separate porter for every state formula.

When a new porter is constructed for a formula, it extracts a polarity environment from the formula. This is accomplished with a simple visitor over the formula tree that implements the $PF$ and $PE$ functions. When this is finished, every porter has a polarity environment that maps from Alloy components to polarities. Thus, by the time a porter is passed to the **Dependency** object it already has a polarity environment associated with it.

### 4.2.1 Sharing Analysis

As mentioned above, our initial implementation of the dependency infrastructure focused on determining which paragraphs had to be imported, or conjoined to the formula being
analyzed, for a given analysis. Once all the porters had been created and registered with the Dependency object, the task of determining which paragraphs' porters were dependent on which other paragraphs' porters remained. Our first attempt at determining whether or not two porters were independent relied on a traditional sharing analysis. If the polarity environments for two porters shared any common constant, we would count that as a dependency. We built a graph of these pairwise dependencies for every pair of porters, and then took the transitive closure of this graph. Then, when the time came to analyze a paragraph, its formula would be conjoined with the formulas of all the state, invariant and definition porters it depended on.

4.2.2 Conflict Analysis

Once we proved that conflicts between polarity environments of formulas were sufficient to assert their independence (claim 2.), we updated our dependency test between porters. Now, we only counted porters as depending on one another if their polarity environments clashed. This resulted in a considerably sparser graph of direct dependencies. The transitive closure of this graph was also somewhat sparser than the transitive closure of the sharing graph, enabling Alcoa to include fewer unnecessary state formulas and invariants in its analyses of different specification elements.
4.3 Extended Implementation with Propagation for Formula Simplification

In addition to determining the dependency graph between paragraphs, polarity analysis allows us to simplify formulas by replacing components with the constants $Un$ and $\{}$ while still guaranteeing the existence of a satisfying assignment. Once these substitutions have been made, constant folding allows for further simplifications which alter the polarity environment, allowing for repeated passes of polarity calculation substitution and simplification. The basic simplification algorithm is:

$$p := PF[f]$$

while ($\{+, -\} \cap (ran\ p) \neq \{\})$ do

for all components $c$ of $f$

if $p(c) = +$ then $f := f(c/Un)$

else if $p(c) = -$ then $f := f(c/\{\})$

$f := \text{Simplify}[f]$

$p := PF[f]$

Simplification continues until there are no + or - polarities in a formula’s polarity environment.
4.3.1 Regular Formulas

Initially, polarity-based simplification was applied to an entire conjunctive formula before it was analyzed. At the same time that polarity based simplifications were being tested on formulas, we were investigating different schemes for translating and solving Alloy formulas. In particular we were investigating the possibility of converting Alloy formulas into negated normal form (NNF), disjunctive normal form (DNF), and conjunctive normal form (CNF). Although this investigation was largely orthogonal to the initial intended use of polarity analysis, namely finding dependencies between formulas, we soon discovered that Alloy formulas in DNF were particularly amenable to polarity analysis.

4.3.2 DNF Clauses

Converting an Alloy formula into DNF involves transforming it into a top level disjunction of subformulas, or clauses. An assignment to any one of these clauses that causes it to be true will cause the whole formula to be true. Thus, each clause can be solved separately, and multiple clauses can be solved in parallel. As an example, consider a formula of the form:

\[ F \lor G \]

This formula is satisfiable if \( F \) is satisfiable or if \( G \) is satisfiable. The subproblems of finding a satisfying assignment to \( F \) and finding a satisfying assignment to \( G \) are completely independent, and can be solved separately.

In the general case, conversion to DNF allows the problem of finding a satisfying assignment to an Alloy formula to be divided easily into subproblems.

In order to convert Alloy formulas to disjunctive normal form, we first convert them to
negated normal form. This process consists of recursively applying the following rewrite rules (which are equivalences):

### NNF rewrite rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 \rightarrow f_2$</td>
<td>$\Rightarrow \neg f_1 \lor f_2$</td>
</tr>
<tr>
<td>$f_1 \leftrightarrow f_2$</td>
<td>$\Rightarrow f_1 \land f_2 \lor \neg f_1 \land \neg f_2$</td>
</tr>
<tr>
<td>$\neg \neg f$</td>
<td>$\Rightarrow f$</td>
</tr>
<tr>
<td>$\neg (f_1 \land f_2)$</td>
<td>$\Rightarrow \neg f_1 \lor \neg f_2$</td>
</tr>
<tr>
<td>$\neg (f_1 \lor f_2)$</td>
<td>$\Rightarrow \neg f_1 \land \neg f_2$</td>
</tr>
<tr>
<td>$\neg (f_1 \rightarrow f_2)$</td>
<td>$\Rightarrow f_1 \land \neg f_2$</td>
</tr>
<tr>
<td>$\neg (f_1 \leftrightarrow f_2)$</td>
<td>$\Rightarrow \neg f_1 \land f_2 \lor \neg f_1 \land f_2$</td>
</tr>
<tr>
<td>$\neg \forall_v . f$</td>
<td>$\Rightarrow \exists_v . \neg f$</td>
</tr>
<tr>
<td>$\neg \exists_v . f$</td>
<td>$\Rightarrow \forall_v . \neg f$</td>
</tr>
</tbody>
</table>

Once in NNF, a formula can be converted to DNF by pushing conjunctions in and disjunctions out using the following rewrite rules (which are also equivalences):

### DNF rewrite rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 \land (f_2 \lor f_3)$</td>
<td>$\Rightarrow (f_1 \land f_2) \lor (f_1 \land f_3)$</td>
</tr>
<tr>
<td>$(f_1 \lor f_2) \land f_3$</td>
<td>$\Rightarrow (f_1 \land f_3) \lor (f_2 \land f_3)$</td>
</tr>
</tbody>
</table>

A very interesting property of DNF clauses in Alloy is that their polarity environments often conflict with one another. These conflicts derive from the fact that DNF clauses express disjoint conditions on state.

For instance, consider the logical statement:

For instance, consider the logical statement:
if rule₁ then result₁ else result₂

which would be translated into an Alloy formula as follows:

\[ rule₁ \rightarrow result₁ \&\& \text{not } rule₁ \rightarrow result₂ \]

Following the rewrite rules above, this formula converted into DNF would be:

\[ (\text{not } rule₁ \&\& result₁) \| (\text{not } rule₁ \&\& result₂) \| (result₁ \&\& result₂) \]

In the original formula, the polarity environments of rule₁ and not rule₁ are joined, resulting in a * polarity for all components in rule₁. In the separate DNF subformulas however, this isn’t the case. Since the separate subformulas are disjoint and can be solved in parallel, we can use polarity based simplification on each subformula.

The next phase of our implementation incorporated both DNF and polarity based simplification. First we converted our Alloy formulas to DNF, using a visitor over the intermediate abstract syntax tree. Our implementation was a straightforward application of the rewrite rules above to the abstract syntax tree. To help break formulas into more DNF clauses, we added two simple transformations:

\[
\begin{align*}
\text{all } v : t \mid f₁ \&\& f₂ & \rightarrow \text{all } v : t \mid f₁ \&\& \text{all } v : t \rightarrow f₂ \\
\text{some } v : t \mid f₁ \| f₂ & \rightarrow \text{some } v : t \mid f₁ \| \text{some } v : t \rightarrow f₂
\end{align*}
\]

Once our complete formula was in DNF form, and ready to be analyzed, we applied iterative polarity based simplification (Section 4.3) to each clause separately, and then invoked the solver on the simplified clauses.
4.4 Extended Implementation with Propagation for Eliminating Dependencies

The final phase of implementation involved combining polarity analysis's two principal uses, simplification and dependency detection. The overall strategy was to consider all the formulas that could possibly be joined to a given formula being analyzed, repeatedly perform polarity based simplification over all the formulas conjoined, and recalculate the dependency graph. As components are substituted and expressions and formulas are simplified, many dependencies drop out between formulas, and implicitly conjoined formulas can be dropped.

This was implemented by taking a global polarity environment across all paragraphs, doing global substitution and simplification and iterating.
Chapter 5

Results

5.1 Overview

Quantitatively measuring the effectiveness of a static analysis like polarity analysis is somewhat tricky, since polarity analysis’s effectiveness is largely dependent upon how and in what context it is applied. In this section, we will examine polarity analysis’s effectiveness at accomplishing the two specific tasks it was designed to perform; detecting independence between Alloy formulas, and simplifying Alloy formulas.

5.2 Performance on Eliminating Dependencies

Originally, we designed polarity analysis for the purpose of detecting independence between specification elements, and thereby reducing the number of formulas that had to be imported into a given analysis. There is likely no single canonical way to quantitatively characterize interdependence between formulas in an Alloy specification. For our purposes, we will
use the convention of measuring dependence and independence that Alcoa itself uses: the *dependency graph*.

As explained in Section 4.2, Alcoa represented dependence between formulas as a dependence graph, whose nodes are paragraphs and state formulas, and whose edges denote dependence. The existence of an edge between two nodes in the dependency graph means that the formulas, or paragraphs represented by the nodes are dependent on one another. If a node representing a condition has an edge connecting it to a node representing an invariant, then in order for Alcoa to accurately analyze the condition in the context of the model, the condition’s formula must be conjoined to the invariant’s formula before analysis.

The dependency graph models *direct dependence* between formulas. Basically, two formulas are directly dependent on one another if we cannot find that they are *independent*, as defined in Section 3.2.2. Since, as we have discussed, proving that two formulas are independent is rather difficult, we employ different approximations of dependence, always erring on the side of caution. While we must detect all dependencies between formulas so as to have accurate analyses, there is no parallel requirement that we detect all independencies. If we err and find a condition erroneously dependent on an invariant, analysis of that condition will still be valid, just overconstrained. Thus, more accurate safe approximations of dependence will find fewer dependencies between formulas, whereas less accurate safe approximations will err more frequently on the side of caution and find more dependencies that might not be necessarily.

Before we implemented polarity analysis, the approximation of formula dependence that we employed was sharing analysis, in which two formulas were independent only if they didn’t mention any of the same components. As explained in 3.2.2, polarity analysis allowed us to
use conflict analysis, an approximation of dependence at least as accurate as that yielded by sharing analysis.

Our first examination of polarity analysis’s effectiveness is a comparison of the dependency graph generated from a given specification by sharing analysis, versus the dependency graph generated by conflict analysis. Shown in the following table is a comparison of the number of direct dependencies, or edges in the dependency graph, detected by sharing analysis and conflict analysis in 16 different specifications:

Table 5.1: Number of Dependencies Detected by Sharing and Conflict Analysis

<table>
<thead>
<tr>
<th>Specification</th>
<th># dependencies from sharing analysis</th>
<th># dependencies from conflict analysis</th>
<th>% difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>c2</td>
<td>111</td>
<td>83</td>
<td>25.23</td>
</tr>
<tr>
<td>click</td>
<td>216</td>
<td>185</td>
<td>14.35</td>
</tr>
<tr>
<td>con</td>
<td>449</td>
<td>339</td>
<td>24.50</td>
</tr>
<tr>
<td>counter-uml</td>
<td>1861</td>
<td>918</td>
<td>50.67</td>
</tr>
<tr>
<td>file-system</td>
<td>66</td>
<td>52</td>
<td>21.21</td>
</tr>
<tr>
<td>finder</td>
<td>152</td>
<td>117</td>
<td>23.03</td>
</tr>
<tr>
<td>graphical</td>
<td>366</td>
<td>328</td>
<td>10.38</td>
</tr>
<tr>
<td>graphs</td>
<td>56</td>
<td>49</td>
<td>12.50</td>
</tr>
<tr>
<td>merge</td>
<td>1067</td>
<td>705</td>
<td>33.93</td>
</tr>
<tr>
<td>mobile-ip</td>
<td>466</td>
<td>357</td>
<td>23.39</td>
</tr>
<tr>
<td>sectors</td>
<td>80</td>
<td>76</td>
<td>5.00</td>
</tr>
<tr>
<td>sets-and-relations</td>
<td>239</td>
<td>119</td>
<td>50.21</td>
</tr>
<tr>
<td>simulations</td>
<td>100</td>
<td>48</td>
<td>52.00</td>
</tr>
<tr>
<td>state-machine</td>
<td>55</td>
<td>47</td>
<td>14.55</td>
</tr>
<tr>
<td>uml-new</td>
<td>1800</td>
<td>958</td>
<td>46.78</td>
</tr>
<tr>
<td>www</td>
<td>216</td>
<td>148</td>
<td>31.48</td>
</tr>
</tbody>
</table>

On average, conflict analysis produces a dependency graph with 27.45 percent fewer dependencies. For some specifications, more than half of the dependencies could be eliminated.
Despite these promising results, actually taking advantage of eliminated dependencies turned out to be more difficult. When analyzing the formula of a given condition or invariant, we must import not only those formulas upon which the given formula is directly dependent, but also those upon which it is transitively dependent. Thus, it is not the direct dependency graph that really determines the degree to which we can exclude unnecessary formulas from analyses, but rather the transitive closure of that graph. Even though conflict analysis can be used to greatly reduce the number of edges in the dependency graph of a specification, we discovered that the effects on the transitive closure of the graph were significantly less pronounced:

Table 5.2: Number of Transitive Dependencies Detected by Sharing and Conflict Analysis

<table>
<thead>
<tr>
<th>Specification</th>
<th># dependencies from sharing analysis</th>
<th># dependencies from conflict analysis</th>
<th>% difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>c2</td>
<td>195</td>
<td>195</td>
<td>0.00</td>
</tr>
<tr>
<td>click</td>
<td>484</td>
<td>484</td>
<td>0.00</td>
</tr>
<tr>
<td>com</td>
<td>736</td>
<td>736</td>
<td>0.00</td>
</tr>
<tr>
<td>counter-uml</td>
<td>9416</td>
<td>9416</td>
<td>0.00</td>
</tr>
<tr>
<td>file-system</td>
<td>110</td>
<td>110</td>
<td>0.00</td>
</tr>
<tr>
<td>finder</td>
<td>206</td>
<td>206</td>
<td>0.00</td>
</tr>
<tr>
<td>graphical</td>
<td>535</td>
<td>535</td>
<td>0.00</td>
</tr>
<tr>
<td>graphs</td>
<td>63</td>
<td>63</td>
<td>0.00</td>
</tr>
<tr>
<td>merge</td>
<td>2016</td>
<td>2016</td>
<td>0.00</td>
</tr>
<tr>
<td>mobile-ip</td>
<td>852</td>
<td>852</td>
<td>0.00</td>
</tr>
<tr>
<td>sectors</td>
<td>180</td>
<td>180</td>
<td>0.00</td>
</tr>
<tr>
<td>sets-and-relations</td>
<td>440</td>
<td>119</td>
<td>72.95</td>
</tr>
<tr>
<td>simulations</td>
<td>112</td>
<td>48</td>
<td>57.14</td>
</tr>
<tr>
<td>state-machine</td>
<td>65</td>
<td>60</td>
<td>7.69</td>
</tr>
<tr>
<td>uml-new</td>
<td>9900</td>
<td>9900</td>
<td>0.00</td>
</tr>
<tr>
<td>www</td>
<td>504</td>
<td>464</td>
<td>7.94</td>
</tr>
</tbody>
</table>
Conflict analysis is not nearly as effective at reducing transitive dependencies as it is at reducing direct dependencies. Some of the specifications, like sets-and-relations and simulations are very simple specifications with intentionally disjoint and under constrained paragraphs. On these simple examples, conflict analysis is able to significantly reduce the number of transitive dependencies. Similarly, on some other interesting models, like www, conflict analysis is somewhat more effective at reducing dependencies than sharing analysis. Unfortunately, however, on many more fleshed out specifications like the uml, com and mobile-ip models, conflict analysis is no more effective than sharing analysis at reducing transitive dependencies.

These results seem to indicate that for many models, polarity analysis won’t be able to reduce the number of formulas that have to be imported for a given analysis. Nevertheless, polarity analysis’s success at reducing the number of direct dependencies between formulas might lead to new applications, such as extracting slices of large models, restructuring models for more efficient analyses, or deeper analysis of why different invariants conflict.

5.3 Performance on Simplifying Clauses

The other application of polarity analysis that we investigated was formula simplification. The algorithm for polarity based formula simplification and its theoretical justification have been presented earlier in Section 3.2.3 and Section 4.3. We performed formula simplification on 307 different Alloy formulas, each one derived from a paragraph in one of 16 specifications. These formulas are all of the exact type that Alcoa sends to be translated into boolean
constraints and solved; they consist of a core paragraph formula, conjoined with all the relevant imports. So as to increase the effectiveness of polarity analysis, we converted all of these formulas to DNF, before performing polarity analysis.

Alloy formulas in DNF (as explained in section 4.3.2) consist of a high level sum of products. When finding a satisfaction to a formula in DNF, the subformulas of the top level disjunction, which we’ll refer to as DNF clauses, can be solved in parallel. For this reason, measuring the effectiveness of formula simplification is slightly more nuanced for formulas with multiple DNF clauses, than for formulas with only one DNF clause, and we will look at these cases separately.

First, we will examine the effectiveness of polarity based simplification on formulas with only one DNF clause. 269 of our formulas, or 87.6% fall into this category, and had only one DNF clause. This top level clause is a large conjunction of subclauses. We will measure the effectiveness of simplification by measuring the percentage of these subclauses our simplification allows us to reduce to true, and thus eliminate from consideration. The subclauses are roughly of the same size, and by measuring the percentage of eliminated subclauses, we can get an accurate estimate of overall reduction in formula size.

Here is table showing the percentage reduction in formula size that polarity based simplification yielded for different formulas:
Table 5.3: Percentage Reduction in Formula Size

<table>
<thead>
<tr>
<th>Reduction in Formula Size</th>
<th># of Formulas in Range</th>
<th>% of Formulas in Range</th>
<th>Reduction in Formula Size</th>
<th># of Formulas in Range</th>
<th>% of Formulas in Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>54</td>
<td>20.07</td>
<td>46% - 50%</td>
<td>21</td>
<td>7.81</td>
</tr>
<tr>
<td>1% - 5%</td>
<td>12</td>
<td>4.46</td>
<td>51% - 55%</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>6% - 10%</td>
<td>36</td>
<td>13.38</td>
<td>56% - 60%</td>
<td>2</td>
<td>0.74</td>
</tr>
<tr>
<td>11% - 15%</td>
<td>35</td>
<td>13.01</td>
<td>61% - 65%</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>16% - 20%</td>
<td>20</td>
<td>7.43</td>
<td>66% - 70%</td>
<td>3</td>
<td>1.12</td>
</tr>
<tr>
<td>21% - 25%</td>
<td>16</td>
<td>5.95</td>
<td>71% - 75%</td>
<td>12</td>
<td>4.46</td>
</tr>
<tr>
<td>26% - 30%</td>
<td>14</td>
<td>5.20</td>
<td>76% - 80%</td>
<td>10</td>
<td>3.72</td>
</tr>
<tr>
<td>31% - 35%</td>
<td>2</td>
<td>0.74</td>
<td>81% - 85%</td>
<td>1</td>
<td>0.37</td>
</tr>
<tr>
<td>36% - 40%</td>
<td>6</td>
<td>2.23</td>
<td>86% - 90%</td>
<td>4</td>
<td>1.49</td>
</tr>
<tr>
<td>41% - 45%</td>
<td>3</td>
<td>1.12</td>
<td>91% - 95%</td>
<td>2</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>96% - 100%</td>
<td>16</td>
<td>5.95</td>
</tr>
</tbody>
</table>

Our technique is able to reduce formula size on average by 28%. While 20% of formulas can’t be reduced in size, almost a quarter of the formulas can be at least halved in size. Thus, in practice, polarity analysis can be a very useful tool on a sizable fraction of formulas, greatly simplifying their analysis.

Applying formula simplification to formulas with multiple DNF clauses yields several different results. For each formula, we simplified each of its DNF clauses separately to maximize the effects of simplification. Thus, for each formula, there is the maximum amount of simplification in any of its DNF clauses, the minimum amount of simplification in any of its DNF clauses, and the average amount of simplification in any of its DNF clauses.

Here is a table showing the results averaged over examples with multiple DNF clauses:
As these results show, our polarity based simplification is particularly effective on formulas with multiple DNF clauses. This is probably due to the observation in section 4.3.2, that DNFing tends to separate conflicting polarity environments into different clauses. Thus, by simplifying the clauses individually, we can obtain large simplifications on the individual clauses, and a significant overall reduction in formula size.
Chapter 6

Related Work

There has been a wealth of related work on Alloy and Alcoa and their predecessors NP and Nitpick. The present technique Alcoa uses to find solutions to Alloy models is described is two papers by Jackson [9] [5]. In addition, earlier techniques to solve models in NP are described elsewhere [7] [4].

The motivation of reducing the space of assignments that has to be searched, has motivated other previous work, such as research on eliminating isomorphic configurations of state from consideration in the solution process [8]. Polarity analysis also reduces the space of assignments, but is a static analysis on the source language of Alloy, and thus operates a much higher level of abstraction than other state space reduction techniques.

In the model checking world, similar motivations to those behind polarity analysis gave rise to the use of “cone of influence reduction” techniques [3] that are used by model checkers to only consider equations effecting relevant variables. The analysis employed by the “cone of influence reduction” is essentially a sharing analysis of relevant variables, similar to the sharing analysis described in Section 4.2.1. In Alloy, polarity analysis yielded a much more
accurate measure of dependence than sharing analysis.

Another technique that, like polarity analysis, involves detecting dependencies between specification elements is specification slicing [10]. This technique allows the extraction of a slice of a specification relevant to a particular schema. It is also based on sharing analysis, and runs into the problem of extracting slices that consist of the entire specification. This problem has motivated work by Richardson on dynamic slicing [2]. Dynamic slicing succeeds in extracting smaller more accurate slices from a specification, but it produces slices that are only relevant for a specific sequence of operations, and thus lacks the completeness of a static analysis like polarity analysis.

Lastly, in the area of model theory, the Lyndon Interpolation Theorem [1] lays deep theoretical foundations for the kind of dependency detection we are seeking in performing polarity analysis.
Chapter 7

Conclusion

We have established that polarity analysis is a sound analysis over the Alloy specification language that can safely and accurately detect dependencies among specification elements, and can be used to dramatically simplify Alloy formulas.

Even over a specification language like Alloy, where traditional programming language notions of variables and execution do not apply, we have shown that static analysis can be very powerful. At the same time, we have further demonstrated that static analysis of whole programs, or in our case, whole specifications is often doomed to be overly conservative. Thus, we can see the need for future research into ways of focusing static analyses like polarity analysis on subprograms or parts of specifications.

With regard to the original goal of polarity analysis, we have seen that statically detecting dependencies is a promising technique. We obtained interesting results, and succeeded in eliminating many dependencies in specifications; but many of the deep problems of how to identify independent specification elements still remain.

The nature of our static analysis is that we stored a fixed amount of information for every
constant in a formula. This fixed amount of information gave polarity analysis its provable efficiency, but seriously detracted from its accuracy. A finer grain analysis, that tracked more information about different constants and their relationships would no doubt succeed in more accurately detecting dependencies, but would not scale to large specifications, and would cease to be a quick static analysis, and become a form of formula satisfaction.

Polarity analysis’s success at formula simplification proved much more dramatic than we had originally anticipated. With the ability to halve the size of many formulas, polarity-based simplification can be integrated into solution tools like Alcoa to significantly accelerate solving.

Much work remains still to be done in this area. As mentioned above, finer grain analyses than the polarity analysis presented here might prove more accurate in detecting dependencies. In addition, many of the potential applications of polarity analysis have yet to be explored. Requirement conflict analysis, for instance, could reap much benefit from an analysis that could precisely indicate how requirements conflict. Similarly, polarity analysis could be of great use in specification slicing and restructuring. In closing, polarity analysis has proven an interesting and effective analysis over the Alloy specification language, with much potential for future work.
# Appendix A

## List of Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>$\text{Formula} \rightarrow \text{Assignment} \rightarrow \text{Boolean}$</td>
<td>meaning function for formulas</td>
</tr>
<tr>
<td>$X$</td>
<td>$\text{Expression} \rightarrow \text{Assignment} \rightarrow \text{Value}$</td>
<td>meaning function for expressions</td>
</tr>
<tr>
<td>$S$</td>
<td>$\text{Formula} \rightarrow \mathcal{P}\text{Assignment}$</td>
<td>satisfying solutions function for formulas</td>
</tr>
<tr>
<td>$\mathcal{PE}$</td>
<td>$\mathcal{C} \rightarrow \mathcal{P}$</td>
<td>a map from components to polarities</td>
</tr>
<tr>
<td>$\sqcup$</td>
<td>$\mathcal{P}\text{Assignment} \rightarrow \mathcal{P}\text{Assignment}$</td>
<td>join of polarities</td>
</tr>
<tr>
<td>$\sqcap$</td>
<td>$\mathcal{PE}\text{nv} \rightarrow \mathcal{PE}\text{nv} \rightarrow \mathcal{PE}\text{nv}$</td>
<td>join of polarity environments</td>
</tr>
<tr>
<td>$\mathcal{PE}$</td>
<td>$\mathcal{Expression} \rightarrow \mathcal{PE}$</td>
<td>polarity environment of expression</td>
</tr>
<tr>
<td>$\mathcal{PF}$</td>
<td>$\mathcal{Formula} \rightarrow \mathcal{PE}$</td>
<td>polarity environment of formula</td>
</tr>
<tr>
<td>$F$</td>
<td>$\mathcal{Formula} \rightarrow \mathcal{P}\text{Assignment}$</td>
<td>all possible assignments to components in formula</td>
</tr>
<tr>
<td>$E$</td>
<td>$\mathcal{Expression} \rightarrow \mathcal{P}\text{Assignment}$</td>
<td>all possible assignments to components in expression</td>
</tr>
<tr>
<td>$A$</td>
<td>$\mathcal{PE}\text{nv} \rightarrow \text{Assignment}$</td>
<td>polarity-based partial assignment</td>
</tr>
</tbody>
</table>
Bibliography


