Worcester Polytechnic Institute
Digital WPI

Doctoral Dissertations (All Dissertations, All Years) Electronic Theses and Dissertations

2013-04-25

First-Order Models for Configuration Analysis

Tim Nelson
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/etd-dissertations

Repository Citation
Nelson, T. (2013). *First-Order Models for Configuration Analysis*. Retrieved from https://digitalcommons.wpi.edu/etd-dissertations/452

This dissertation is brought to you for free and open access by Digital WPI. It has been accepted for inclusion in Doctoral Dissertations (All Dissertations, All Years) by an authorized administrator of Digital WPI. For more information, please contact wpi-etd@wpi.edu.

First-Order Models for Configuration Analysis

by

Tim Nelson

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

May 2013

APPROVED:

Professor Kathi Fisler, Thesis Co-Advisor, WPI Computer Science

Professor Daniel J. Dougherty, Thesis Co-Advisor, WPI Computer Science

Professor Craig Wills, Head of Department, WPI Computer Science

Professor Joshua Guttman, WPI Computer Science

Professor Shriram Krishnamurthi, Brown University Computer Science

Abstract

Our world teems with networked devices. Their configuration exerts an ever-expanding influence on our daily lives. Yet correctly configuring systems, networks, and access-control policies is notoriously difficult, even for trained professionals. Automated static analysis techniques provide a way to both verify a configuration's correctness and explore its implications. One such approach is scenario-finding: showing concrete scenarios that illustrate potential (mis-)behavior. Scenarios even have a benefit to users without technical expertise, as concrete examples can both trigger and improve users' intuition about their system. This thesis describes a concerted research effort toward improving scenario-finding tools for configuration analysis.

We developed Margrave, a scenario-finding tool with special features designed for security policies and configurations. Margrave is not tied to any one specific policy language; rather, it provides an intermediate input language as expressive as first-order logic. This flexibility allows Margrave to reason about many different types of policy. We show Margrave in action on Cisco IOS, a common language for configuring firewalls, demonstrating that scenario-finding with Margrave is useful for debugging and validating real-world configurations.

This thesis also presents a theorem showing that, for a restricted subclass of first-order logic, if a sentence is satisfiable then there must exist a satisfying scenario no larger than a computable bound. For such sentences scenario-finding is *complete*: one can be certain that no scenarios are missed by the analysis, provided that one checks up to the computed bound. We demonstrate that many common configurations fall into this subclass and give algorithmic tests for both sentence membership and counting. We have implemented both in Margrave.

Aluminum is a tool that eliminates superfluous information in scenarios and allows users' goals to guide which scenarios are displayed. We quantitatively show that our methods of scenario-reduction and exploration are effective and quite efficient in practice. Our work on Aluminum is making its way into other scenario-finding tools.

Finally, we describe FlowLog, a language for network programming that we created with analysis in mind. We show that FlowLog can express many common network programs, yet demonstrate that automated analysis and bug-finding for FlowLog are both feasible as well as complete.

This work is dedicated to the memory of my father.

Acknowledgements

First and foremost comes my wife, Emily. Without her support – emotional, intellectual, creative, and financial – I would not have been able to complete this work.

I owe my advisors, Dan Dougherty and Kathi Fisler, along with the rest of my committee: Joshua Guttman, Shriram Krishnamurthi, and Craig Wills, enormous thanks for their advice and patience throughout my years as a PhD student. Other mentors have had a profound effect as well, most notably Dan Bach, Nancy Ryanen-Grant, Virginia Schaefer, and Stanley Selkow.

My fellow ALAS lab members, including but not limited to: Ken Breeman, Yu Feng, Evan Frenn, Theophilos Giannakopoulos, Sarah Jaffer, Chris King, Guillaume Marceau, Salman Saghafi, Curtis Taylor, Francis Usher, and Danny Yoo have been an invaluable resource and good friends.

I am blessed with a loving and understanding family: my mother, Lisa Nelson, my sister Erika and brother Cary. My father-in-law and mother-in-law, John and Linda Gibbons.

I am grateful to Joshua Guttman, Daniel Jackson, and Emina Torlak, Joe Beck, and Janice Gobert for valuable discussions about the Aluminum project. I thank John Basik, Jeff Coady, Mark Dieterich, Jason Montville and Richard Silverman for giving sysadmins' perspectives on firewall-analysis in Margrave, and Craig Wills and Matt Disney for additional feedback.

In compiling our related work, we contacted many authors with questions about their projects. We thank them for their prompt and cheerful responses and hope we have represented their work accurately; any errors are our own.

Cisco supported an early phase of our firewall-analysis project. My graduate studies received support from WPI's Institute Fellowship, the Carl and Inez Weidenmiller Endowed Fellowship, and the Graduate Assistantships in Areas of National Need program. This work was also partially supported by the NSF. I am grateful for this generous support.

There are likely dozens of others who deserve to be acknowledged here; if you have been left out, please bring your copy of this document to me in exchange for a cookie.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

System configuration is a difficult, high-risk task. Even a small misconfiguration can result in disaster, yet misconfigurations abound [OGP, Woo04]. In large enterprise systems, the problem of correct configuration is even more difficult [Woo10], and such misconfigurations are highly visible and costly. For instance, a 2011 outage in Amazon Web Services [Ama11] and a 2010 outage in Facebook [Rob10] were both caused by a configuration change. Even if the root cause of an outage is a hardware failure rather than a misconfiguration, restoring service can involve reconfiguration of backup hardware under a tight deadline. One outage can be quickly followed by another, as experienced by PayPal in 2010 [Sco10].

In the face of these dangers, the first line of defense is run-time testing of configurations. While such testing is valuable, it also has weaknesses. In the context of a network or federation of policies, on-line testing requires either extensive hardware resources or a complex virtualized infrastructure. Both of these are vulnerable to their own configuration problems. Moreover, such testing cannot account for all possible scenarios. In contrast, *static* analysis of configurations, also called "off-line" analysis, never actually deploys a configuration. Instead, configurations are treated as mathematical artifacts. Automated reasoning techniques are then used to investigate the behavior of the configuration. In many cases, static analysis *can* cover all possible scenarios, although it is limited by the mathematical model of the system used. For instance, a 2010 WordPress outage [Jus10] was caused by a cable plugged into the wrong socket; most static analyses will not catch such low-level errors. This balance of strengths and weaknesses makes static analysis a good complement to on-line testing.

Scenario-finding is one particular type of static analysis. Scenario-finding tools provide concrete instances of how a software artifact (such as a configuration) behaves. For instance, given a router configuration, a scenario-generating tool would produce an example of packets being routed by that configuration (using random or invented names and packet-headers). Scenario-finding is more than just automated simulation, however: it can be targeted to specific goals in a system's output, or be focused on particular portions of a configuration.

The concreteness of scenarios makes scenario-finding a popular technique in software engineering. Software modeling helps system designers understand the consequences of their specifications, determine missing constraints, and explore alternatives. Specification languages like UML benefit from scenario-finding to help bridge concrete and abstract representations. Scenarios are also useful for presenting counter-examples to verification tasks. This is especially helpful when studying security policies, where the concreteness helps envision attacks [NBD$^+$10, FKMT05, DGT07, ABL$^+$10]. Network modeling systems similarly use scenarios to visualize designs [BEW95, TASB07].

In short, scenarios are useful for many reasons:

- They are concrete, making it easy for users to grasp the output and map it to reality.
- They do not require logical expertise to grasp. Thus, a logic-aware modeler working with a

domain expert innocent of the joys of logic can present the output scenarios to the domain expert, who should be in a position to understand them.

- They are rigorous, in that we can ascribe a precise semantics to them. This makes it possible to employ them in formal settings.

Because of this unique combination of user-focused characteristics, many tools provide scenario-based output. High-quality scenario-finding tools for configuration analysis could drastically reduce the danger of configuration authoring and maintenance. Configuration-like artifacts, such as access-control policies and network programs, would also be productive to analyze. The goal of this work is to produce such tools and lay foundations for the improvement of existing tools. In this document, we assert and defend that:

Declarative, predicate-logic-based formalisms make suitable foundations for expressing real-world system configurations, access-control policies, and network programs, as well as for building user-focused static-analysis tools for analyzing them.

1.1 Research Questions

This document will support the above thesis by answering several concrete questions:

Configurations can be complex, often involving state, and they are expressed in a plethora of languages which differ not only by syntax, but also by semantics and domain. What logical foundation is suitable for representing configurations, policies, and network programs such that scenario-finding can be performed? To be suitable, a foundation must not only be able to represent the configurations, but be amenable to scenario-finding and other user-focused techniques.

Existing tools for scenario-finding focus on efficiently finding and presenting single scenarios (or showing that none exist), rather than on encouraging the *exploration* of the entire space of scenarios. This view neglects the main advantage of scenario-finding: leveraging users' strong intuition for examples, even in the absence of a formal specification of "correctness". *Which scenarios* ought to be presented first? How might scenario-finding tools *encourage exploration* and gradual increase in understanding? In what ways can tools that perform bounded scenario-finding *increase confidence* in their results, or make *guarantees of completeness*?

1.2 Contributions

The contributions of this thesis are manifest in the following software projects:

Margrave Margrave is a static analysis tool for off-line testing of policies and configurations. Margrave has been in development, in one form or another, for seven years. It has been applied to different domains including access-control policies and firewall configurations. Margrave differs from most other static analysis tools in that it is agnostic toward the type of configuration it analyzes. It is not only (say) a "firewall analysis tool". Instead, the tool provides a general policy framework on which support for different domains and input languages can be built. Unsurprisingly, Margrave is a scenario-finder: given a configuration and a query, Margrave produces a concrete set of scenarios that show how the configuration can behave in the situations the query describes. Margrave's user interface is designed to encourage an iterative exploration of scenarios by way of an interactive prompt. Users can view the results of a query about one policy, then load another and ask "How would these interact," then refine that query to focus on a particular rule, all without "re-compiling" and leaving their existing thread of analysis and exploration. Chapter 3 presents the Margrave tool itself, while Chapter 6 shows Margrave in action by analyzing Cisco router configurations.

Finally, Margrave can often *automatically* compute bounds under which its scenario-finding is complete, that is, no scenarios will ever be "missed". We accomplish this via a theorem stating, for a class of formulas that often appears in policy analysis, that whenever there exists a scenario for the policy's behavior, there must exist a "small" scenario under a bound that can be easily computed from the policy and query. In fact, this result is not tied to the policy-analysis domain: *any* first-order logic formula benefits, provided it falls into an easily determined syntactic class. Chapter 4 discusses the proof of this theorem and presents concrete and efficient algorithms for both detecting whether the formula falls into our class and computing sufficient bounds for such formulas.

Aluminum The Alloy Analyzer [Jac12] is a pre-existing popular scenario-finder. Alloy has been used in hundreds of academic papers in software engineering, security, networking, and other domains. Alloy allows users to specify systems and properties in relational logic with transitive closure. Although Alloy can also be used for verification tasks, it focuses more on finding and displaying scenarios. Alloy lets users see examples of how their system behaves, even in the absence of formal goals, but only searches for scenarios up to a user-provided size bound.

Our project, Aluminum, is an extension to the Alloy Analyzer that displays only *minimal* scenarios: those which contain nothing unnecessary. Since they lack superfluous information, minimal scenarios are more succinct than others, which we believe facilitates user understanding. Moreover, removing non-minimal scenarios from consideration greatly reduces the number of scenarios users must wade through in their exploration.

Aluminum does more than just minimize its output; it also shows what new facts can be consistently added to each minimal scenario. For instance, on an operating-system model, Aluminum might say that it is possible to add a new file to a filesystem or make someone that file's owner. The tool also allows users to *add* a consistent truth and examine the consequences of that addition. In short, Aluminum gives users the ability to explore scenarios in a disciplined manner, rather than just asking for yet another. We discuss our notion of minimality and the Aluminum itself in depth in Chapter 5.

FlowLog The more expressive a configuration language, the more difficult it is to analyze statically. Configuration languages for traditional network hardware are often fairly limited and hence easy to analyze. However, fully programmable *software-defined* networks, wherein a central controller directs the global action of the network, change the nature of the beast: software-defined network configurations can be far more powerful and flexible. This flexibility results in more opportunities for mis-configuration and greatly complicates analyses. Thus the networking field is now faced with a classic programming language design trade-off: that between analyzability and expressive power. Chapter 7 presents FlowLog, a declarative, stateful policy language for programming network controllers. FlowLog is both expressive enough to be used for real-world network programming and restrictive enough to facilitate rich, tractable model-checking and scenario-finding.

Roadmap

This document is organized so as to showcase each of these projects separately. Due to the multi-pronged nature of this research, we discuss related work as appropriate in each chapter. Chapter 8 closes with discussion and a list of future research directions.

Chapter 2

Foundations: Order-Sorted Predicate Logic

The foundational setting of this thesis is order-sorted predicate logic. Since many of the concepts are used throughout, this chapter serves as a primer on the necessary background. The definitions and results in this chapter are either directly from Goguen and Meseguer's work [GM92] or they are the obvious extensions required to handle relations as well as functions.

Notation We use $\langle\rangle$ for the empty sequence. If $(\mathcal{S}, \leq)$ is an ordering we extend $\leq$ to words in $\mathcal{S}^*$ and then to products, pointwise.

Signatures An *order-sorted signature* is a triple $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ where $(\mathcal{S}, \leq)$ is a finite poset of sorts and Σ is an indexed family of symbols, the vocabulary, comprising

- $\{\Sigma_w \mid w \in \mathcal{S}^*\}$, an $\mathcal{S}^*$-sorted family of *relation symbols,* and
- $\{\Sigma_{w,A} \mid w \in \mathcal{S}^*\ , A \in \mathcal{S}\}$, an $(\mathcal{S}^* \times \mathcal{S})$-sorted family of *function symbols.*

We assume that the Σ_w and $\Sigma_{w,A}$ are pairwise disjoint.

For readers familiar with the Alloy Analyzer tool, we stress that an order-sorted signature is not the same as a signature in Alloy. Here, a signature denotes a language of discourse: available sorts, functions, etc. along with an ordering on the sorts. In this way, an Alloy signature is a sort or a predicate within an overall order-sorted signature.

Formalizing relations and functions through words—rather than through tuples — of sorts simplifies certain definitions and eases capturing overloaded function symbols (as [GM92] does). The restriction to finite vocabulary is reasonable given the fact that our main concern here is the creation of software tools and (in Chapter 4) finite-model theorems. Our work assumes function symbols are not overloaded (through the disjointness condition on the $\Sigma_{w,A}$); this is consistent with Margrave and Alloy. Most of the results herein generalize to handle overloading, including our finite model theorem (Theorem 4.4.1). The one exception is our term-counting algorithm (Theorem 4.4.2), which relies on the lack of overloading to compute precise bounds; with overloading, our algorithm only promises upper bounds on the sort-sizes.

When $R \in \Sigma_w$ we say that w is the *arity* of R. When $f \in \Sigma_{w,A}$ we say that w is the *arity* of f and A is the *result sort* of f. If $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ and $\mathcal{L}' = (\mathcal{S}, \leq, \Sigma')$ are such that for each w and A, $\Sigma_w \subseteq \Sigma'_w$ and $\Sigma_{w,A} \subseteq \Sigma'_{w,A}$ we say that $\mathcal{L}'$ is an *expansion* of $\mathcal{L}$, and that $\mathcal{L}$ is a *reduct* of $\mathcal{L}'$.

Following standard usage, a function symbol $a \in \Sigma_{\langle\rangle,A}$, taking no arguments, is referred to as a "constant" of sort A, and in concrete syntax we write simply a instead of $a()$.

The *connected components* of an ordering $(\mathcal{S}, \leq)$ are the equivalence classes for the equivalence relation generated by $\leq$. A signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ is *coherent* if each pair of sorts in the same connected component has an upper bound. Henceforth we assume that our signatures are coherent. See [GM92] for an extended discussion of the importance of coherence. Note: in [GM92] the notion of coherence also requires that signatures be *regular*, a technical condition that is trivially satisfied in the absence of overloading.

The set of *formulas* is defined inductively by closing the set of atomic formulas under the propositional operators $\wedge$, $\vee$, and $\neg$ and the quantifiers $\exists$ and $\forall$. We will indicate quantification over a sorted variable $x \in X_A$ by $\exists x^A$ or $\forall x^A$ (where X_A is the set of variables of sort A). The notions of free and bound variable are standard; let $FV(\phi)$ denote the set of free variables of formula ϕ. A *sentence* is a formula with no free variable occurrences.

Models Fix a signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. An $\mathcal{L}$-*model* $\mathbb{M}$ comprises (i) an $\mathcal{S}$-sorted family $\{\mathbb{M}_A \mid A \in S\}$ of sets, the *universe* of $\mathbb{M}$, such that $A \leq A'$ implies $\mathbb{M}_A \subseteq \mathbb{M}_{A'}$, (ii) for each $R \in \Sigma_w$ a relation $R^{\mathbb{M}_w} \subseteq \mathbb{M}_w$, and (iii) for each $f \in \Sigma_{w,A}$ a function $f^{\mathbb{M}_{w,A}} : \mathbb{M}_w \to \mathbb{M}_A$.

Skolemization, the process of eliminating existential quantifiers in favor of function symbols, will be important in Chapter 4. As a consequence we need to be attentive to the ways that the language over which our models are defined can shift. If $\mathbb{M}$ is a model for $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ and $\mathcal{L}'$ is an expansion of $\mathcal{L}$ then an *expansion* of $\mathbb{M}$ to $\mathcal{L}'$ is a model of $\mathcal{L}'$ with the same universe as $\mathbb{M}$ which agrees with $\mathbb{M}$ on the symbols in Σ.

Definition 2.0.1. *An environment η over a model $\mathbb{M}$ is an $\mathcal{S}$-indexed family of partial functions $\{\eta_A : X_A \to \mathbb{M}_A \mid A \in \mathcal{S}\}$ such that $\eta_A = (\eta_{A'})\restriction_{X_A}$ (the restriction to X_A) whenever $A \leq A'$. As usual the notation $\eta[x^A \mapsto e]$ refers to the environment agreeing with η except that it maps variable $x \in X_A$ to e. An environment η can be extended to terms in the usual way.*

Definition 2.0.2 (Truth in a model)**.** *Let $\mathbb{M}$ be a model, ϕ a formula, and η an environment such that $FV(\phi) \subseteq dom(\eta)$. The relation $\mathbb{M} \models_\eta \phi$ is defined by induction over ϕ as follows.*

- *If $\phi \equiv P(t_1, ..., t_n)$ then $\mathbb{M} \models_\eta \phi$ if and only if $P^{\mathbb{M}}(\eta(t_1), ..., \eta(t_n))$ holds.*
- *If $\phi \equiv t_1 = t_2$ then $\mathbb{M} \models_\eta \phi$ if and only if $\eta(t_1) = \eta(t_2)$ holds.*
- *If $\phi \equiv \neg\alpha$ then $\mathbb{M} \models_\eta \phi$ if and only if $\mathbb{M} \not\models_\eta \alpha$.*
- *If $\phi \equiv \alpha \wedge \beta$ then $\mathbb{M} \models_\eta \phi$ iff $\mathbb{M} \models_\eta \alpha$ and $\mathbb{M} \models_\eta \beta$.*
- *If $\phi \equiv \alpha \vee \beta$ then $\mathbb{M} \models_\eta \phi$ iff $\mathbb{M} \models_\eta \alpha$ or $\mathbb{M} \models_\eta \beta$.*
- *If $\phi \equiv \exists\, x^A \alpha$ then $\mathbb{M} \models_\eta \phi$ iff there is some element e in $\mathbb{M}_A$ such that $\mathbb{M} \models_{\eta[x \mapsto e]} \alpha$.*
- *If $\phi \equiv \forall\, x^A \alpha$ then $\mathbb{M} \models_\eta \phi$ iff for each element e in $\mathbb{M}_A$ it holds that $\mathbb{M} \models_{\eta[x \mapsto e]} \alpha$.*

Homomorphism A *homomorphism* $h : \mathbb{M} \to \mathbb{N}$ between models $\mathbb{M}$ and $\mathbb{N}$ is an $\mathcal{S}$-sorted family of functions $\{h_A : \mathbb{M}_A \to \mathbb{N}_A \mid A \in \mathcal{S}\}$ satisfying the following conditions (suppressing sort information for readability): (i) $A \leq A'$ implies $h_A = (h_{A'}) \restriction_{\mathbb{M}_A}$ (ii) $h(f^{\mathbb{M}}(a_1, \ldots, a_n)) = f^{\mathbb{N}}(h(a_1), \ldots, h(a_n))$, and (iii) $R^{\mathbb{M}}(h(a_1, \ldots, a_n))$ implies $R^{\mathbb{N}}(h(a_1), \ldots, h(a_n))$.

The Term Model When the set of relation symbols in $\mathcal{L}$ is empty then the set of ground terms forms the universe of a model for $\mathcal{L}$, the *term algebra* [GM92]. We may view this as a model for an arbitrary order-sorted signature, as follows.

Fix $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. The family $\{\mathcal{T}^{\mathcal{L}}_A \mid A \in \mathcal{S}\}$ of *ground terms* over $\mathcal{L}$ is the $\subseteq$-least family such that (i) $\mathcal{T}^{\mathcal{L}}_A \subseteq \mathcal{T}^{\mathcal{L}}_{A'}$ whenever $A \leq A'$ and (ii) if $f \in \Sigma_{w,A}$ with $w = A_1 \ldots A_n$ and for each i, $t_i \in \mathcal{T}^{\mathcal{L}}_{A_i}$ then $f(t_1, \ldots, t_n) \in \mathcal{T}^{\mathcal{L}}_A$. The ground terms determine a model $\mathcal{T}^{\mathcal{L}}$ of $\mathcal{L}$, the *term model*, by taking the interpretation of each $f \in \Sigma_{\langle A_1 \ldots A_n \rangle, A}$ to be the function taking each tuple

$(t_1, \ldots, t_n) \in (\mathcal{T}^{\mathcal{L}}_{A_1} \times \cdots \times \mathcal{T}^{\mathcal{L}}_{A_n})$ to the term $f(t_1, \ldots, t_n)$, and taking the interpretation of each relation symbol to be the empty relation.

It may be that one would like to specify that certain pairs of sorts are to be disjoint (*e.g.* Alloy, with "extends"). For simplicity we do not consider such a mechanism in our notion of signature, since we can get the same effect by explicitly writing disjointness sentences and conjoining them with any sentences under consideration (as long as they are in the same connected component!). For future reference we note that such sentences involve no existential quantifiers.

Theorem 2.0.1. *Suppose $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ is a signature such that Σ has no relation symbols. Then for any model $\mathbb{M}$ of $\mathcal{L}$ there is a unique homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathbb{M}$ (i.e., $\mathcal{T}^{\mathcal{L}}$ is initial).*

Proof. Initiality of $\mathcal{T}^{\mathcal{L}}$ in the category of *algebras* was shown by Goguen and Meseguer [GM92]. Now, given an $\mathcal{L}$-model $\mathbb{M}$, we let $\mathbb{M}'$ be the reduct of $\mathbb{M}$ to the language $\mathcal{L}'$ obtained by removing the relation symbols. So $\mathbb{M}'$ is a $\mathcal{L}'$-algebra so that Goguen and Meseguer's theorem applies. But the unique algebra homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathbb{M}'$ is itself a $\mathcal{L}$-homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathbb{M}$, simply because each $\mathcal{T}^{\mathcal{L}}$-relation is empty, and the result follows. $\square$

Definition 2.0.3 (Open terms)**.** *Fix a signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. Let X be an $\mathcal{S}$-sorted set $\{X_A \mid A \in \mathcal{S}\}$ of variables, with the X_A mutually disjoint and disjoint from Σ. The set $\mathcal{T}^{\mathcal{L}}_X$ of* (open) terms over *X is, intuitively, obtained by adjoining the variables in X_A to the term model at type A. Formally we proceed as follows [GM92]. Define Σ^X to be the family of symbols with ${\Sigma^X}_{\langle\rangle,A} = \Sigma_{\langle\rangle,A} \cup X_A$ and ${\Sigma^X}_{w,A} = \Sigma_{w,A}$ for $w \neq \langle\rangle$. Then $\mathcal{L}^X$ is the signature $(\mathcal{S}, \leq, \Sigma^X)$. Then the family $\mathcal{T}^{\mathcal{L}}_X$ of open terms over X is the family $\mathcal{T}^{\mathcal{L}^X}$ that is the terms of $\mathcal{T}^{\mathcal{L}}_X$ are the ground terms over $\mathcal{L}^X$.*

As noted by Goguen and Meseguer [GM92] the fact that the open term algebra is initial in the category of $\mathcal{L}^X$-algebras entails the fact that this algebra is *free* over the generators X in the category of $\mathcal{L}$-algebras. In particular, if $\eta : X \to \mathbb{M}$ is an $\mathcal{S}$-sorted assignment of values in $\mathbb{M}$ to variables from X then there is a canonical way to extend η to map $\mathcal{T}^{\mathcal{L}}_X$ to $\mathbb{M}$.

Aside: On reduction to unsorted logic

It is not unusual for treatments of many-sorted logic to "encode" sorts as unary predicates and to view many-sorted logic as a particular syntactic discipline over standard one-sorted logic. For example one can reasonably view the sorted quantification $\exists x^A \,.\, \phi$ as shorthand for $\exists x \,.\, (A(x) \wedge \phi)$. This is the traditional approach taken in mathematical logic [End72]. To handle subsorting it is natural to introduce *coercion functions:* to capture $A \leq B$ in the order-sorted setting we can add to the signature a function symbol $c_{AB} : A \to B$ in the flat setting. There are some natural axioms imposed on the coercion functions: that they are injective, that they compose properly to reflect transitivity, and so forth. Goguen and Meseguer [GM92] prove a Reduction Theorem, showing that there is an equivalence of categories between order-sorted algebra and flat many-sorted algebra, once the latter is equipped with coercion functions and the conditional equations expressing the axioms.

This theorem lifts, of course, to our setting of order-sorted *logic.* But, for a variety of reasons, we resist such an encoding into unsorted logic. First, it would make counting ground terms, our central concern in Chapter 4, more difficult. This is because the ground terms would involve the coercion functions and we would have to count *modulo* the axioms governing them, otherwise we would overestimate the size of the true set of ground terms in the original signature. Second, introducing coercion functions would clutter the treatment of results such as Herbrand's Theorem. Finally, we want our results to provide clear information to users of tools—like both Margrave and Alloy—that are explicitly order-sorted, and an encoding into another formalism would be an obstacle to these users. So we prefer to work with order-sorted logic directly.

Chapter 3

The Margrave Tool

Margrave is a flexible, powerful tool for policy analysis that has been in development for seven years. The original version of Margrave appeared in a 2005 paper by Fisler, et al. [FKMT05], and performed verification and change-impact analysis on policies written in a core fragment of XACML. We have since re-written the original tool to reason about more general policies, and have successfully applied it to Cisco IOS router configurations (more on this in Chapter 6), representing both packet filtering ACLs and static routing as Margrave policies. We use "Margrave" to refer to the current state of the tool, not the original 2005 version.

Margrave's flexibility comes from thinking about policy analysis from an end-user's perspective. The questions that users wish to ask about policies obviously affect modeling decisions, but so does our form of answer. Margrave's core paradigm is *scenario finding*: when a user poses a query, Margrave produces a (usually exhaustive) set of scenarios that witness the queried behavior. Whether a user is interested in the impact of changes or how one rule can override another, scenarios concretize a policy's behavior. Margrave also allows queries to be built incrementally, with new queries refining the results from previous ones.

Margrave's power comes from choosing an appropriate model. Embracing both scenario-finding and multi-level policy-reasoning leads us to model policies in first-order logic. Many analysis tools (especially in the firewall-analysis domain, which we discuss in Chapter 6) use propositional models for which analysis questions are decidable and efficient. In general, one cannot compute an exhaustive and finite set of scenarios witnessing first-order logic formulas. Fortunately, the formulas corresponding to many common policy queries do yield such sets. Margrave identifies such cases automatically, thus providing exhaustive analysis for richer policies and queries than other tools; we discuss the details of computing this exhaustive yet finite set in Chapter 4.

Before delving into the details of Margrave, we first discuss the motivation of the tool: what do we mean when we say "policy analysis" in the first place?

3.1 Motivation: Policies and Configurations

Policies have many different forms and purposes. The use-cases below sketch the space that we are interested in modeling.

Networking: Packet Filters Packet filters (often referred to as "access-control lists" or ACLs) are simple access-control policies used in firewalls. Rules either `permit` or `deny` packets. The first rule that applies to a packet dictates whether that packet is forwarded or dropped by the firewall. While this style of policy occurs elsewhere, our main concern in this example is ACLs in the firewall domain, written in languages such as Cisco IOS, Juniper's JunOS, *iptables* scripts, etc. The following *iptables* script serves as a packet filter. Each line in the script is a command-line invocation. The `-A` switch instructs the firewall to append a new rule, in this case to the `INPUT` policy, which applies to incoming packets. `-p tcp` says that the rule only matches if the protocol used is TCP. `-s` adds a constraint on the source address of incoming traffic; `--dport` similarly constrains the destination TCP port. Finally, `-j DROP` and `-j ACCEPT` give the decision to render should a packet match the rule. Rules are interpreted in the order in which they appear.

```
/sbin/iptables -A INPUT -p tcp -s 10.1.2.3 --dport 80 -j DROP
/sbin/iptables -A INPUT -p tcp -s 10.1.2.0/24 --dport 80 -j ACCEPT
```

Given (for example) the packet header `<10.1.2.3, 192.235.10.11, 12345, 80, TCP>`, this filter will `drop` the packet. In some firewall configuration languages, packet filters can be more complex, e.g. by the addition of `log` rules, which always apply if matched, irrespective of the rule order:

```
/sbin/iptables -A INPUT -p tcp -s 10.10.10.0/24 --dport 80 -j ACCEPT
/sbin/iptables -A INPUT -p tcp -s 10.1.2.3 --dport 80 -j DROP
/sbin/iptables -A INPUT -p tcp -s 10.1.2.3 --dport 80 -j LOG
```

This policy will both `drop` and `log` the packet header `<10.1.2.3, 192.235.10.11, 12345, 80>`.

Networking: Static and Policy-Based Routing While routers can make use of *dynamic routing* algorithms[1], their configurations frequently contain *static routing* instructions for well-known or local destinations. For example, this Cisco IOS command:

```
ip route 0.0.0.0 0.0.0.0 Serial0
```

which directs all traffic out the `Serial0` interface. Cisco (and other vendors) also provides a more flexible policy-based routing feature:

```
route-map all-employee-machines permit 10
match ip address 10
set ip next-hop 10.10.10.1
```

This snippet says that all traffic matching `access-list` 10 (itself a packet filtering policy!) ought to be forwarded to address `10.10.10.1`. Policy-based routing allows fairly sophisticated routing policies to be expressed in the configuration itself, without utilizing dynamic methods.

Networking: Stateful Firewalls While routing policies and packet filtering can interact in interesting ways, we must also consider *state*. Firewalls are no longer "dumb" packet filters; they are aware of the traffic flowing to and fro across their interfaces, and their configuration often takes this fact into account. For instance, the IOS policy:

```
interface fe0
ip access-group fromoutside in

interface Vlan1
ip access-group frominside in

ip access-list extended frominside
  permit tcp any any reflect returntcp

ip access-list extended fromoutside
  evaluate returntcp
  deny tcp any any
```

allows only *return* traffic from the outside world. No access is allowed via the `fe0` interface that is not necessary to service a flow initiated from the `serial0` interface. The bolded commands in the example cause such flows to be monitored via `returntcp`, a temporary state table.

[1]Such as Open Shortest Path First (OSPF), Border Gateway Protocol (BGP), etc.

Access Control Access-control policies are a more general version of packet-filtering firewalls. Broadly, they describe whether a *subject* entity is allowed to take a certain *action* on a *resource*. XACML [OAS05] is an example of an access-control policy language in wide use.

In XACML, a policy consists of rules, each of which has a *target*, a *condition*, and a *decision*. If both the target and condition match, the rule matches and renders its *decision*; conflicts are resolved by means of a *rule-combination* setting (for instance, favoring the first applicable rule). The *target* element a rule governs whether that rule is considered applicable to a request; the *condition* element dictates whether the rule actually fires for the request.

An XACML policy should only render one decision. Conflicts between rules must sometimes be resolved. XACML provides several conflict-resolution algorithms, one of which is the "first applicable" algorithm seen in packet-filtering firewalls. Also of interest are "permit-overrides" and "deny-overrides", which state that rules yielding permit (respectively, deny) always take precedence.

XACML also supports *policy sets*, which themselves contain a collection of policies and (smaller) policy sets. This feature allows policies to be created by separate entities and then consulted separately, depending on the nature of the request. Like rules, policies and policy sets have *target* elements that advertise their domains of applicability.

XACML is an XML language. An example rule in XACML is:

```
<Rule RuleId="myrulename" Effect="Permit">
<Target>
  <Subjects>
    <AnySubject/>
  </Subjects>
  <Resources>
    <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue
      DataType="http://www.w3.org/2001/XMLSchema#string">Tim's Thesis</AttributeValue>
      <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
                                   AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>

  </Resources>
  <Actions>
    <AnyAction/>
  </Actions>
</Target>
</Rule>
```

This rule allows any subject to take any action on resources with `id` equal to "`Tim's Thesis`".

Federations of Heterogeneous Policies In the wild, policies interact to produce the overall behavior of a system. A network is the canonical example of policy federation: multiple firewalls may make access and routing decisions on a packet as it traverses the network. As the packet travels, it may be modified by operations such as NAT. Moreover, when a packet containing (say) a database request reaches its destination, new policies come into play, deciding whether or not to honor the request. This interaction between policies is especially powerful in the cloud, since the federation may be distributed between different principals. Similarly, a single physical device may embody a large number of policies. A physical firewall houses packet filters, routing policies, NAT policies, etc. and is a federation in itself.

3.2 Motivation: Analysis

We support all of the above policy types with a general, unified model for policies, which we can then use to help policy authors better understand the implications of what they write. Of course, that goal raises the question of just what kind of "understanding" we plan to promote.

Margrave performs off-line, static analysis of policies. Policy analysis questions can take many forms. We enumerate some common question types below. For each type, we include an example in Margrave's current query language.

"What if?" The most simple, and most natural, question about a policy's behavior is: *Given a concrete request, what will the policy do?*

Some examples of "What if?" queries are:

1. "Will an IP packet with the header (10.1.1.2, 192.168.2.3, 20000, 80, tcp) be allowed through the packet filter?"
2. "Will patient A be permitted to access patient B's medical records — provided the database is in state C?"

```
Example Margrave Query for (2)
let Q[r: Record] be
  recordFor(r, 'patientB) and
  AccessPolicy.permit('patientA, 'access, r) and
  exists d: Doctor | assignedDoctor(d, 'patientB);
```

This analysis is amenable to ordinary run-time testing: simply load the policy and send the request! However, that approach requires that the policy already be deployed – either to a production environment (which would be unwise for an un-tested policy) or a sandboxed testing environment (which may be expensive or impractical in some cases). Margrave's static analysis is therefore useful even for such simple, "testable" questions.

"When can...?" More generally, we might ask: *Under what circumstances can the policy yield a certain decision?*

This question is more abstract than the last. Rather than inquiring about the fate of a single request, we are interested in the results for all possible requests. Instead of a yes or no answer, we expect something more substantial: a set of *scenarios* that describe when the behavior of interest can occur.

Some examples of "When can ...?" queries are:

1. "Which IP packets from `web.cs.wpi.edu` will be allowed through the firewall?"
2. "Can a student ever read someone else's grades?"

```
Example Margrave Query for (2)
let Q[s: Subject, r: Resource] be
  student(s) and grade(r) and
  exists s2: Subject | (not s1 = s2) and
    gradeFor(s2, r) and student(s2) and
    AccessPolicy.permit(s, 'read, r)
```

While, in theory, run-time testing may be able to answer this type of question, it is often impractical. Testing a firewall with every possible TCP packet, to determine which pass successfully, would take an unacceptable amount of time.

"What did I change?" After making an edit to a policy a natural question is: "What did this change affect?"

Answering this question requires going beyond just the syntactic alterations in the policy text. Rather, we are interested in how the policy's behavior changed as a result of the edit. We call this *change-impact analysis*.

Some examples of "What did I change?" queries are:

1. "Which IP packets will now be allowed through the firewall?"

2. "Of those newly permitted packets, do any involve the accounting or human-resources networks?"

Example Margrave Query for (2)

```
let Q[s: Subject, a:Action, r: Resource] be
    not OldAccessPolicy.permit(s, a, r) and
    NewAccessPolicy.permit(s, a, r) and
    (inHRNet(s) or inAccountingNet(s))
```

Change-impact analysis is one instance in the broader class of *property-free verification.* In traditional property verification, one has two artifacts to compare: a set of formally expressed properties, and a system specification. In property-free verification, other artifacts are compared; in the case of change-impact analysis, the artifacts are the old and the new policy. Property-free verification can be helpful when there are no formally articulated properties to verify, or when attempting to elucidate these properties.

Indeed, policy authors often do not know what properties they wish to verify until those properties are violated — with embarrassing or disastrous results. Change-impact analysis can reveal questions that the policy author did not realize they should ask.

"Why did that happen?" Confronted with unexpected policy behavior, a policy author would like to know *which portion(s) of the policy* is responsible for the behavior. In a single policy, this may amount to discovering the rule that fired for a request. In a federation, it also involves computing the chain of custody for the request as it traversed the federation.

Some examples of "Why did that happen?" queries are:

1. "Which filtering rule is responsible for dropping my boss' web packets?"
2. "Which firewalls handled my e-mail packets on the way out of the corporate network?"

Example Margrave Query for (1)

```
let Q[p: Packet] be true;
show realized PacketFilter.rule1_applies(p),
              PacketFilter.rule2_applies(p),
              ...;
```

More complex questions Policy authors may also have broader questions about the behavior of their policies. They may wish to feed the results of one query into a second or aggregate query results. They may also seek domain-specific phenomena that require constructing queries in the context of a larger application. To support such questions, a simple query language is insufficient; advanced users require Margrave to be intelligently integrated with a programming language.

Some examples of domain-specific questions are:

1. "Which roles, when added to a set of roles, increase privileges vs. the original set?"
2. "Which of my firewall rules never apply? For each of those rules, which prior rules collude to cause the inapplicability?"

A fragment of a Racket program using the Margrave API might resemble:

Example Margrave Script for (1)

```
  (define fmla-sexpr `(and ([Mypol permit] s a r)
                          (Write a)
                          ,@(cover-roles-qry aroleset)))
  (m-let thisid
        '([s Subject]
          [a Action]
          [r Resource])
        fmla-sexpr)

  (define result (m-is-poss? thisid))
```

This fragment shows how a Margrave query can be constructed as an expression in the Racket [FP10] programming language, and how query results can be obtained and manipulated.

Having seen concrete examples of our notion of "policy" and "query", we can now discuss the current state of the tool in detail.

3.3 The Tool

Margrave comprises a front-end read-eval-print loop (REPL) written in Racket [FP10] and a backend written in Java. The front-end handles parsing (of queries, commands, policies, and vocabularies) and output presentation. Users write policies and issue queries in the Racket front-end. The actual analysis and scenario generation occurs in the Java backend, which uses a tool called Kodkod [TJ07] that produces solutions to first-order formulas using SAT-solving. Those solutions are then returned to the front-end and presented to the user. Figure 3.1 shows Margrave in action.

In Margrave, there are two key objects: the *policies* to be analyzed and the *queries* that describe analysis.

3.3.1 Policies

A Margrave policy describes a list of rules that dictate a *decision* for *requests* to the policy, along with conflict-resolution settings for when different rules apply to the same request. For error-checking purposes, policies contain type declarations for all variables used. Margrave's policy language is inspired by XACML [OAS05], but has a more readable format. Figure 3.2 gives an example Margrave policy.

Line 1 declares the policy's vocabulary name, which we discuss in the next paragraph. Variables used by a policy must be declared (lines 2 – 5) along with their type. The rules on line 7 – 10 say that faculty members can always write grades, students can always read their own grades, and nothing else is allowed. The `RComb` expression on line 11 indicates that permit decisions should always override deny decisions; we discuss such conflict-resolution options later.

Vocabularies Every policy in Margrave uses a single *vocabulary*, which gives the signature (types, predicates, functions, etc.) that the policy uses. For instance, a vocabuluary for firewalls will define a notion of IP address, predicates for state tables, and possibly functions to describe NAT. Such a vocabulary can be shared by many different firewall policies. A sample vocabulary (for the school policy in Figure 3.3) can be found in Figure 3.3. The first line declares the name of the vocabulary, which is referenced in policies. Lines 2 – 5 declare several types; some are subtypes of others (e.g. `Read` and `Write` are subtypes of `Action`). Lines 6 – 10 declare the predicates: similar to types but not constrained to be unary, and unused by the type checker.

Formalism of Policies and Vocabularies Relations are a natural representation for the conditions in policy rules, the association of decisions with requests, and interactions between policies. Accordingly, we use relational structures as the foundation of our notion of policy. Margrave uses order-sorted predicate logic (Chapter 2), and a Margrave *vocabulary* defines an order-sorted signature, with the constraint that functions may not be overloaded. As usual, we view *constants* as function symbols of 0 arguments.

Nothing in the definition of an order-sorted signature, and thus nothing in Margrave vocabularies, is specific to security, firewalls, or access-control; this is intentional. Margrave is a general tool for reasoning about rule-based specifications. When instantiated for a policy, a vocabulary would contain relations for the policy's decisions (such as permit and deny), each of whose type ranges over the form of request for that policy (e.g., a subject-action-resource triple in access-control or packet contents for a firewall). Concepts such as "Action", "Resource", or "IP Address" would constitute the sorts.

Policies are formed of rules that define some relations in Σ in terms of others. Specifically:

Definition 3.3.1. *Given a vocabulary Σ, a* policy P against Σ *is a set of rules of the form*

$$Dec(\vec{u}) :\!-\, L_1(\vec{u_1}) \ldots, L_n(\vec{u_n})$$

where Dec is a relation name in Σ that does not occur in the right-hand side of any rule, each $L_i(\vec{u_i})$ is an atomic or negated atomic formula over Σ, and $\vec{u}$ and each $\vec{u_i}$ is a vector of variables and constants.

This is a semi-positive datalog program [AHV95]. It is fruitful to view the semantics of such policies from the logic programming/database perspective.

We may view the facts about the system and the environment as a database over the relations in the policy vocabulary, and view the policy as a database query whose answer predicates are the decision predicates. Thus, applying the policy to an environment is just running the policy as a query on the database that is the environment. The *view* computed by this query determines a relation for each decision-predicate: these are the "database tables" defining permit, deny, etc.

From this point of view it is entirely reasonable to imagine allowing recursion in the policy, by allowing decision-predicates on the right-hand sides of rules, though this would require a more careful handling of negation. There are thus no fundamental obstacles to supporting recursion, but the current implementation of Margrave does not support it. Although recursion is not a target of this thesis, support for recursion is one motivation for investigating potential new approaches to scenario-computation.

Policy Completion Margrave policies differ from Datalog in another important way. Datalog programs take a set of ground facts and compute IDB tables from those facts. For instance:

```
permit(s, r) :- administrator(s), file(r).
permit(s, r) :- ownerOf(s, r).
administrator(Alice).
employee(Bob).
ownerOf(Bob, Bobcar).
file(Billing).
```

results in a `permit` IDB that includes $[Alice, Billing]$ and $[Bob, Bobcar]$. Margrave needs to do more – it must allow information to flow in the other direction. Suppose a query asks for scenarios in which $permit(s, r)$ holds. Then Margrave must produce scenarios that describe bindings for s and r. Thus for each decision, the implementation uses a *bi*-implication to encode the fact that a decision can only be rendered because of some rule's action. For the above, it would use:

$$permit(s, r) \iff (administrator(s) \wedge file(r)) \vee ownerOf(s, r)$$

rather than only the "if" direction, as in the Datalog program. This approach is similar to the completion semantics for Negation as Failure [Cla78] in Prolog.

Conflict Resolution The Datalog IDB viewpoint leads to a complication in the policy setting. In the Datalog perspective, a request may be contained in the relations for multiple decisions. While there are no *foundational* complications with this, in practice some decisions cannot be allowed to apply concurrently. For example, in most access-control contexts, it is nonsensical for a request to be both permitted and denied. Such a situation is a conflict that the policy must resolve. Margrave's policy language echoes XACML and others in providing conflict-resolution (also known as rule-combination) settings. Currently supported rule-combination algorithms are:

- **First-applicable**: Rules may apply in the order they are given; the first rule that matches a request will apply.

- **Overrides** $(D_1, ..., D_n)$: Takes an ordered list of decisions as an argument; conflicts are resolved by favoring the highest-priority decision.
- **No-op**: Potential conflicts will be ignored; multiple decisions can apply.

At present, only one algorithm may be provided per policy. Recent work by Giannakopoulos [Gia12] will allow more complex conflict-resolution, but is not currently implemented.

3.3.2 Queries

By way of example, we will examine the "What if?" query from that section. (In the current language, preceding an identifier with a single-quote tells Margrave that it is a constant symbol, rather than a variable name. All text on a line after a double-slash is ignored by Margrave.)

```
// Will patient A be permitted to access patient B's medical
// records --- provided a doctor has been assigned to patient B?

let Q[r: Record] be
  recordFor(r, 'patientB) and
  AccessPolicy.permit('patientA, 'access, r) and
  exists d: Doctor | assignedDoctor(d, 'patientB);
```

The `let` keyword binds the name `Q` to the formula, which can include both universal and existential quantifiers, as well as the usual boolean operators of first-order logic. The variable `r` is free. The existential quantifier on `d` is required: if there were no quantifier and `d` were not listed among the typed free variables in the declaration of `Q`, the formula would be ill-formed and rejected by Margrave. The explicit types are used in error-checking.

The `let` construct lets users compute sets of scenarios, but does not provide a way to view them. The query language therefore includes a set of commands for getting information about sets of scenarios. Each `let` statement initializes an iterator over the corresponding set of scenarios. Commands for showing scenarios utilize these iterators, which are stateful across REPL commands. Subsection 3.3.4 details the list of available commands and their semantics; first we give the semantics of queries within the REPL.

3.3.3 Semantics of Queries

Our semantics needs to support several features: formulas are bound to names that can be reused in subsequent queries, formulas may reference the decisions of multiple policies, and commands support iteration over the scenarios resulting from a query. This section describes the formal infrastructure that enables these actions. Intuitively, at any point during REPL execution, the system has a current vocabulary (indicating the predicates that formulas can reference), a current environment mapping formula names to formulas, and a store of the current state of each formula's iterator. Formally:

Definition 3.3.2. *The* state *of the* REPL *consists of the current vocabulary Σ, an environment Γ that maps formula names to Σ-formulas, and a store σ that maps bound formula names to an iterator over the set of scenarios for that formula.*

The rest of this section discusses how each construct in the query language depends on and updates the state of the REPL. When the REPL is started afresh, all three components are initialized to empty.

Binding and Interpreting Formulas A formula declaration is of the form

$$\textbf{let } f[x_1 : A_1, \ldots x_k : B_k] \textbf{ be } \alpha$$

where α is an ordinary first-order logic formula over the formula names bound in the environment and the functions and relations defined in the current vocabulary. Quantifiers annotate their variables with types. The set of free variables of α must be a subset of $\{x_1, \ldots, x_k\}$.

Semantically, the **let** statement first type checks the formula (for consistent use of other formulas in Γ), then

- Sends α and the current environment to the query interpreter, which returns the set S of scenarios for α.
- Defines an iterator $\mathcal{I}_f$ over S and binds f to $\mathcal{I}_f$ in the iterator store.
- Extends the environment with the mapping of f to α (including the type declaration).

Our notion of type checking is standard. The details of interpretation and formal definition of scenarios follow.

Interpretation and Scenarios

Definition 3.3.3. *Given a vocabulary $\Sigma = (\mathcal{S}, \leq, \Sigma, ar)$, a model $\mathbb{M}$ of Σ comprises*

- *A* universe *of elements for each type in $\mathcal{S}$. If $A \leq A'$, then the universe of A is contained in the universe of A'. We allow universes to be empty.*
- *For each predicate R with type signature $A_1 \times \cdots \times A_k$, a set of tuples $\langle e_1, \ldots, e_k \rangle$ such that each e_i is in the universe corresponding to A_i.*
- *For each function symbol f with type signature $A_1 \times \cdots \times A_k \to A$, a function from tuples over the corresponding universes of each A_i to the universe of A.*

Given a formula ϕ over Σ, a scenario for *ϕ is a pair $(\mathbb{M}, \eta)$ where $\mathbb{M}$ is a Σ-model and η is a type-respecting map from free variables in ϕ to $\mathbb{M}$ such that $\mathbb{M}, \eta \vDash \alpha$ in the standard semantics of first-order logic.*

Given a formula α, vocabulary Σ, and formula environment Γ, we interpret α by inlining each reference to a formula f in α with $\Gamma(f)$, inlining each reference to a formula identifier with the formula corresponding to that name and computing the models of α under Σ. Note that inlining formula bindings is safe as we do not allow rebinding of the same formula name, thus eliminating the distinction between static and dynamic binding semantics.

If a type symbol, relation symbol, or function symbol in the current REPL vocabulary $\Sigma$is not mentioned in a query Q and the policies it references, Margrave will omit that symbol from the scenarios it produces for Q. This omission is only for optimization: if a formula does not contain any occurance of a relation, the value of that relation in a model $\mathbb{M}$ will have no effect on the formula's truth in $\mathbb{M}$.

3.3.4 Commands

After a **let** construct has been used to bind a query to a name, additional commands are used to manipulate the scenarios for a query. Each command in our query language operates in the context of the current vocabulary, environment, and store (although in fact not all of these "commands" affect the store).

- **show** f: returns the next scenario in $\mathcal{I}_f$.

- **show-all** f: returns the set of all scenarios for $\Gamma(f)$.
- **reset** f: updates the binding of f in the store to an iterator over all scenarios for $\Gamma(f)$ (which can be obtained by re-interpreting the formula bound to f in the environment).
- **count** f: returns the cardinality of the set of scenarios for $\Gamma(f)$. This value is implementation specific, as some model finders work with bounds on the universe sizes, while others remove isomorphic scenarios.
- **isposs?** f: returns true iff the set of scenarios for $\Gamma(f)$ is not empty.
- **show realized** C: returns the set of those $c \in C$ such that $\Gamma(f) \wedge c$ is satisfiable.
- **show realized** C **for cases** D**:** for each $d \in D$, returns the set of those $c \in C$ such that $\Gamma(f) \wedge d \wedge c$ is satisfiable.
- **load-policy** $P = filename$: concrete syntax defined in the Margrave documentation). Adds a typed relation $P.D$ to the vocabulary for every relation D appearing on the left side of a rule within p. If the vocabulary specified in p has not already been loaded, merge p's vocabulary with the extended current vocabulary to form the updated vocabulary. (It is possible that the vocabularies conflict, so that a merge is not well-defined: the command fails in this case.)

3.4 Implementation Details

Margrave's backend must produce sets of solutions to first-order logic formulas. We currently use a tool called Kodkod [TJ07] that produces solutions to first-order formulas using SAT solving.[2] SAT solvers handle propositional formulas. Kodkod bridges the gap from first-order to propositional formulas by asking users for a finite universe-size; under a finite universe-size, first-order formulas translate easily to propositional ones. For instance, consider the original first-order sentence:

$$\forall x\ host(x) \quad \Longrightarrow \quad \exists y\ (router(y) \wedge CanAccess(x,y))$$

Assume a universe of size 2 with elements A and B. Expand the $\forall$-formula with a conjunction over each of A and B for x:

$$\begin{aligned} host(A) \quad &\Longrightarrow \quad \exists y\ (router(y) \wedge CanAccess(A,y)) \wedge \\ host(B) \quad &\Longrightarrow \quad \exists y\ (router(y) \wedge CanAccess(B,y)) \end{aligned}$$

Next, expand each $\exists$-formula with a disjunction over each of A and B for y:

$$\begin{aligned} host(A) \quad \Longrightarrow \quad & (router(A) \wedge CanAccess(A,A)) \vee \\ & (router(B) \wedge CanAccess(A,B)) \wedge \\ host(B) \quad \Longrightarrow \quad & (router(A) \wedge CanAccess(B,A)) \vee \\ & (router(B) \wedge CanAccess(B,B)) \end{aligned}$$

Replace each remaining formula with a propositional variable (e.g., $router(A)$ becomes p_2):

$$\begin{aligned} p_1 \quad \Longrightarrow \quad & (p_2 \wedge p_3) \vee \\ & (p_4 \wedge p_5) \wedge \\ p_6 \quad \Longrightarrow \quad & (p_2 \wedge p_7) \vee \\ & (p_4 \wedge p_8) \end{aligned}$$

[2]Within Kodkod, we use a SAT-solver called SAT4J [BP10].

Every solution produced using a bounded size is legitimate (in logical terms, our analysis is *sound*). However, analysis will miss solutions that require a universe larger than the given size (in logical terms, it is not *complete*). Fortunately, for many policies and queries, Margrave is able to automatically compute a size to use such that completeness is preserved (see Chapter 4.)

Regardless of whether Margrave can automatically produce a universe size, each query may be enhanced with a user-provided size.

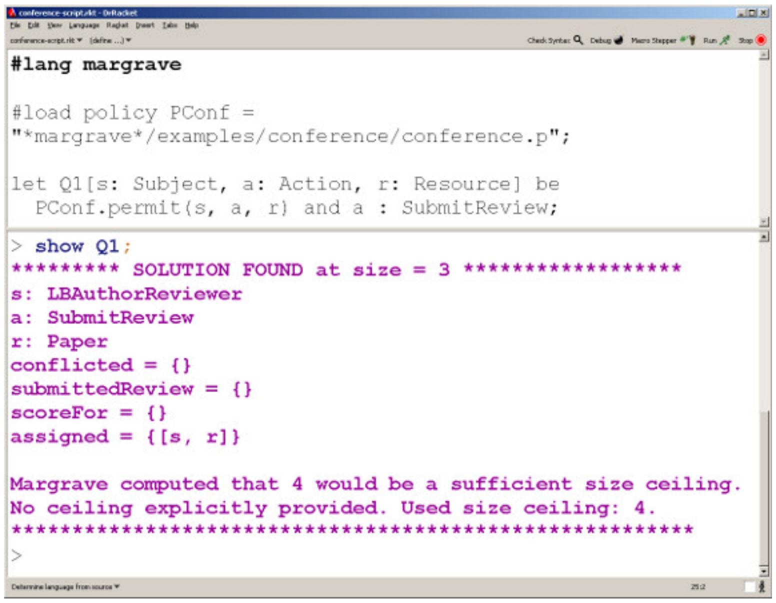
Overall, we believe sacrificing exhaustiveness for the expressive power of first-order logic in policies and queries is worthwhile, especially given the large number of practical queries that can be checked exhaustively.

3.5 Conclusion

Margrave is constantly being developed. The latest public release can be found at:

www.margrave-tool.org

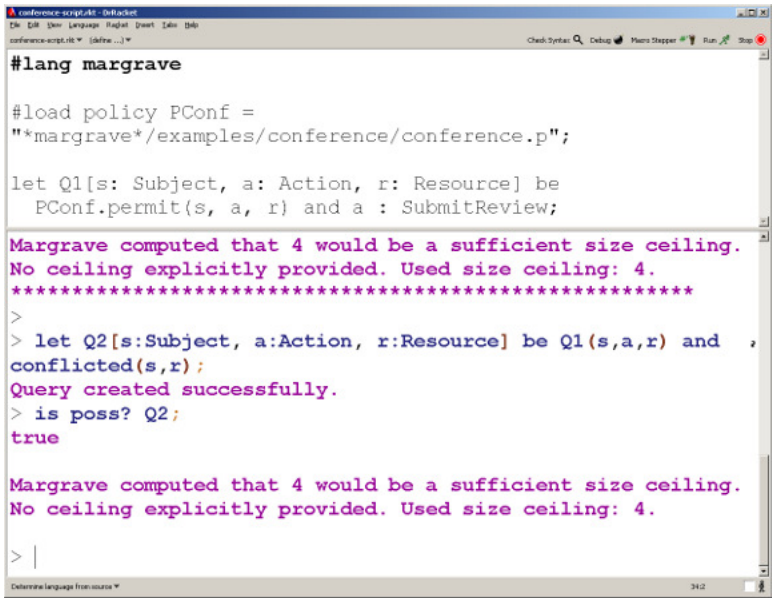
We discuss many potential future enhancements to Margrave in Chapter 8.



```
#lang margrave

#load policy PConf =
"*margrave*/examples/conference/conference.p";

let Q1[s: Subject, a: Action, r: Resource] be
  PConf.permit(s, a, r) and a : SubmitReview;
```

```
> show Q1;
********* SOLUTION FOUND at size = 3 ******************
s: LBAuthorReviewer
a: SubmitReview
r: Paper
conflicted = {}
submittedReview = {}
scoreFor = {}
assigned = {[s, r]}

Margrave computed that 4 would be a sufficient size ceiling.
No ceiling explicitly provided. Used size ceiling: 4.
*********************************************************
>
```

(a) Running a Query



```
#lang margrave

#load policy PConf =
"*margrave*/examples/conference/conference.p";

let Q1[s: Subject, a: Action, r: Resource] be
  PConf.permit(s, a, r) and a : SubmitReview;
```

```
Margrave computed that 4 would be a sufficient size ceiling.
No ceiling explicitly provided. Used size ceiling: 4.
*********************************************************
>
> let Q2[s:Subject, a:Action, r:Resource] be Q1(s,a,r) and
conflicted(s,r);
Query created successfully.
> is poss? Q2;
true

Margrave computed that 4 would be a sufficient size ceiling.
No ceiling explicitly provided. Used size ceiling: 4.

> |
```

(b) Refining a Previous Query

Figure 3.1: The Margrave Tool

```
(Policy uses SchoolVocab
        (Variables
         (Variable s Subject)
         (Variable a Action)
         (Variable r Resource))
        (Rules
            (ProfessorsCanWrite = (permit s a r) :- (isFaculty s) (Write a) (Grade r))
          (StudentsCanReadOwn = (permit s a r) :-
             (isStudent s) (Read a) (Grade r) (gradeBelongsTo r s))
          (DenyOtherwise = (deny s a r) :- true))
        (RComb (fa permit deny)))
```

Figure 3.2: A Margrave policy specification

```
(Vocab SchoolVocab
             (Types
              (Type Subject)
              (Type Action > Read Write)
              (Type Resource > Grade))
             (Predicates
              (Predicate isStudent Subject)
              (Predicate isFaculty Subject)
              (Predicate isAdministrator Subject)
              (Predicate gradeBelongsTo Grade Subject)))
```

Figure 3.3: A Margrave vocabulary

Chapter 4

Completeness Guarantees for Scenario-Finding

The undecidability of first-order logic poses a challenge to using first-order scenario-finding tools for verification: analysis performed under bounds may not be complete. While incompleteness is unavoidable for some classes of formulas, there are also classes for which analysis is complete under domains of finite size. Margrave, like other similar tools, requires finite domain-size bounds. In the general case, it obtains these bounds from the user, and does not help users determine whether their bounds suffice for completeness. Ideally, tools like Margrave or Alloy would provide such feedback or, better still, compute sufficient bounds automatically when possible.

Sufficient-bounds results are long-established for classical first-order logic. Margrave's logic, however, is different in ways that impact computing bounds. Margrave vocabularies yield first-order logic with *sorts*: the class of many-sorted first-order logic formulas with sufficient bounds properly includes that for the unsorted case. Existing results on sufficient bounds for many-sorted logic, however, make assumptions that are not valid for many Margrave policies or Alloy specifications: both tools allow sorts to be empty, and also allow sorts to overlap. These features, which are critical for modeling realistic systems, require an extended theory of bounds-computation. This work[1] presents the theory and algorithms for computing sufficient bounds for a substantial class of first-order formulas.

We actively use these results within the current version of Margrave, which was introduced in Chapter 3. One of our standard policy examples—from a deployed conference-paper manager—requires the results we present here. Margrave uses the presented algorithms to compute how many papers are required for complete reasoning. In other examples, Margrave computes sufficient bounds on *some* sorts (even when others cannot be bounded). This can help a user decide how to allocate the computational resources of model finding. Margrave is built upon Kodkod [TJ07], the backend model-finder for the Alloy Analyzer and thus these results are of interest to both tools.

4.1 Overview of Results

The Bernays-Schönfinkel-Ramsey class, sometimes called "Effectively Propositional Logic" (EPL), comprises the set of first-order sentences of the form

$$\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m \,.\, \phi$$

where ϕ is quantifier-free and has no function symbols. The satisfiability problem for this class is decidable: Bernays and Schönfinkel [BS28] and Ramsey [Ram30] showed that such a sentence has

[1]A preliminary version [NDFK12] of this work appeared at the International Conference on Abstract State Machines, Alloy, B, and Z.

a model if and only if it has a model of size bounded by n plus the number of constants in ϕ. When such a *finite model property* holds, satisfiability-testing reduces to exhaustive search for a model within bounded domains. Furthermore, the search need only consider models whose elements are constants. In effect, satisfiability for these formulas reduces to propositional satisfiability.

The EPL results assume that all variables quantify over the same domain. Alloy uses a *sorted* first-order logic, in which values come from several domains (Alloy signatures) and each variable is associated with a particular domain. Sorts provide additional information that model finders and theorem provers can exploit in the search for models [Jer88, HRCS02, dMB08b, LS04]. More strikingly, the class of sentences with the finite model property is richer in a sorted framework [Harpt, FG03, ARS10]. The following simple example illustrates the interplay between sorts and bounds for completeness. Consider the class of unsorted sentences of the form

$$\forall y_1 \exists x \forall y_2 \,.\, \phi.$$

Satisfiability is undecidable for this prefix class [BGG97]. In contrast, this sorted verion

$$\sigma \equiv \forall y_1^A \exists x^B \forall y_2^A \,.\, \phi \tag{4.1}$$

is better behaved. Suppose that ϕ contains constants, say n_A constants of sort A and n_B of sort B, but no function symbols. If we were to postulate that sort A is a subsort of sort B, then if σ has any models at all then it has a model whose size at sort A is bounded by n_A and whose size at sort B is bounded by $(2n_A + n_B)$. On the other hand, if our signature declared that B were a subsort of A, then some σ would only have infinite models. In considering subsort relationships, this example illustrates *order-sorted* logic, in which there is a partial order on the sorts rather than an assumption that all sorts are disjoint. We give a formal treatment of this example below, as Example 3.

Tools using order-sorted logic that allow empty sorts demand new methods for computing sufficient bounds. To illustrate why, we first consider a standard approach to establishing the finite-model property. Let σ be a sentence in unsorted first-order logic.

1. By Skolemization, there is a universal sentence σ_{sk} equi-satisfiable with σ. The language of σ_{sk} is richer than that of σ, since constants and function symbols have been introduced on behalf of existential quantifiers of σ.

2. Any potential model $\mathbb{M}$ for σ_{sk} has a *Skolem hull* [CK73] consisting of the interpretation in the model of the ground terms of the language. The set of ground terms is called the *Herbrand universe*. The Skolem hull forms a submodel of $\mathbb{M}$ in which every element is named by a term in the language.

3. A fundamental classical theorem is that the truth of universal sentences is preserved under submodel. Thus, if the signature of σ_{sk} has only finitely many terms, that is, if the Herbrand universe is finite, then σ has the finite-model property.

When the language has only a single sort, the only way to guarantee that the Herbrand universe is finite is to have no function symbols (other than constants). In that setting, the sentences whose Skolemization produces no function symbols comprise the EPL class. The many-sorted setting is more lenient. Consider for example a sentence σ whose Skolemization leads to a language with simply a constant a of sort A, a function f of sort $A \to B$. Then the only ground terms that can be constructed are a and $f(a)$ (terms such as $f(f(a))$ are not well-sorted). This suggests—correctly—that a richer classes of finite-model results are available.

But there are technical obstacles to generalizing the above argument. In particular,

- When empty sorts are allowed, the Skolem form of σ is not equi-satisfiable with σ. For example the sentence $(\forall y^A . y = y) \vee (\exists x^B . x \neq x)$ is true in models where the sort B is empty. Skolemization, with a new constant b of sort B, yields the sentence $(\forall y^A . y = y) \vee (b \neq b)$ which is is unsatisfiable. Section 4.2 addresses this issue formally.

- When sorts are not assumed to be disjoint (the order-sorted setting), not every element in the Skolem hull of a model is named by a term. Indeed the Skolem hull of $\mathbb{M}$ can be infinite even when a finite submodel of $\mathbb{M}$ *does* exist. Example 2 in Section 4.2 illustrates this case.

Contributions We adapt the approaches in the standard argument to accommodate ordered sorts and empty sorts. In doing so, it enables automatic bounds computation for additional Alloy formulas through the following contributions:

- We identify (Definition 4.3.2) a syntactically-determined class of sentences extending EPL, comprising *Order-Sorted Effectively Propositional Logic (order-sorted effectively-propositional logic)*, for which the Finite Model Property holds (Theorem 4.3.2, Section 4.3).
- We present a linear-time algorithm (Corollary 4.4.1) for membership in order-sorted effectively-propositional logic. We present a cubic-time algorithm (Theorem 4.4.2) for computing an upper bound on the size of models required for testing satisfiability. It is interesting to note that the bound itself can be exponential in the size of the sentence (Section 4.4), even though it can be computed in polynomial time.

We view identification of the order-sorted effectively-propositional logicclass as a contribution to a taxonomy of decidability classes in order-sorted logic. In the presence of possibly-empty sorts, sentences do not always have equivalent prenex-normal forms, so we cannot attempt a decidability classification in terms of quantifier prefix as in [BGG97]. As Section 4.3 shows, our decidability criterion is based entirely on the signature of the Skolemization of the given formula. This signature can be viewed as a generalization of the idea of quantifier prefix, as it implicitly records the pattern of nesting between universal and existential quantification.

An Example in Margrave

The PLT Scheme application Continue [KHM+07] automates many conference-management tasks. Margrave has been helpful in developing and analyzing Continue; here we hint at some of the ways that the algorithms given in this chapter have improved some of these analyses.

As shown in Chapter 3, we represent the access-control policy of a conference in Margrave as a first-order theory, over a language representing facts about the world and access-control decisions such as *permit* and *deny*. For example, a certain policy rule might say "The conference administrator can advance the conference out of the `bidding` phase if every reviewer has bid on some paper." This rule gives rise to the sentence

$$\begin{aligned} \mathit{permit}(s, \mathit{advancePhase}, \mathit{conference}) \longleftarrow\ & \mathit{Admin}(s) \wedge (\mathit{phase} = \mathit{Bidding}) \wedge \\ & \forall u^{User} \exists p^{Paper} \,.\, \mathit{bidOn}(u,p)). \end{aligned}$$

The set of such policy rules, together with assumed facts about the application domain, comprise a background theory for analysis. Now suppose one wants to verify the property: "The conference chair can modify user passwords." It suffices to determine that there are no models of *the negation of* the formula

$$\forall r^{User} \,.\, \mathit{permit}(\mathit{chair}, \mathit{modifyPassword}, r)$$

together with the background theory. The question, of course, is determining an upper bound on the scope of the search. The fact that the formula being explored by the user is purely existential is of little help by itself, since the entire policy theory is part of the satisfiability query. Besides rules such as the permit rule quoted above, the language also includes such function symbols as $\mathit{paperPhase} : \mathit{Paper} \to \mathit{PaperPhase}$ and $\mathit{decision} : \mathit{Paper} \to \mathit{Decision}$.

In the absence of sensitivity to sorts this theory would not submit to a finite-model discipline. But in fact, for the Continue theory and the associated query above Margrave automatically computes sufficient bounds:

```
Conference:7  PaperPhase:12 Object:14 ConferencePhase:10
Action:16 User:9 Resource:6 univ:59 Paper:6
```

Without the support of the finite-model algorithms herein, the user would—à la Alloy—have to instruct the tool to restrict attention to a finite search space that was presumably arrived at in an *ad-hoc* manner.

4.2 Skolemization

Lemma 4.2.1. *Every formula is logically equivalent to a formula in negation normal form.*

Proof. As for standard one-sorted logic. DeMorgan's laws for pushing negations below $\wedge$ and $\vee$, and the equivalences between $\neg\exists x^A\alpha$ and $\forall x^A\neg\alpha$ all hold, even in the presence of empty sorts. $\square$

A formula is in *negation-normal* form if the negation sign is applied only to atomic formulas. As for standard one-sorted logic, DeMorgan's laws for pushing negations below $\wedge$ and $\vee$, and the equivalences between $\neg\exists x^A\alpha$ and $\forall x^A\neg\alpha$ all hold, even in the presence of empty sorts. So every formula is logically equivalent to a formula in negation normal form. But the fact that models can have empty sorts changes the rules for how quantifiers may be moved within a formula. In particular the passage between $((\exists x^A\alpha)\vee\beta)$ and $\exists x^A(\alpha\vee\beta)$ (when x is not free in β) does not hold if A can be empty (and of course the dual equivalence involving $\forall$ fails as well) and so we cannot in general percolate quantifiers to the front of a formula. So we cannot restrict our attention to formulas in prenex normal form, but we will always pass to negation-normal form.

Definition 4.2.1 (Skolemization)**.** *Let ϕ be a negation-normal form formula over signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$; the result of a* Skolemization-step *of ϕ is any formula ϕ' that can be obtained as follows. If $\exists x^A.\psi(x^A, x_1^{A_1}, \ldots, x_n^{A_n})$ is a subformula occurrence of ϕ that is not in the scope of an existential quantifier, let f be a function symbol not in Σ, and let ϕ' be the result of replacing the occurrence of $\exists x^A.\psi(x, x_1, \ldots, x_n)$ by $\psi(f(x_1, \ldots, x_n), x_1, \ldots, x_n)$. Note that ϕ' is a formula in an expanded signature obtained by adding f to $\Sigma_{\langle A_1,\ldots,A_n\rangle,A}$.*

A Skolemization *of a formula ϕ is a sentence with no existential quantifiers, obtained from ϕ by a sequence of such steps.*

It is not hard to see that any two Skolemizations of a sentence will differ only in the names of the new function symbols used. We do not need this result here and so will not prove it. But in order to unambiguously speak of *the* Skolemization of a sentence σ let us agree that we will eliminate existential formulas from left-to-right and use a canonical well-ordering of the universe of potential vocabulary symbols. With this understanding, if σ is a sentence over $\mathcal{L}$ we we will speak of "the Skolemization" of σ, and denote it σ_{sk}.

Lemma 4.2.2. *For any σ we have $\sigma_{sk} \models \sigma$.*

Proof. Note that the signature of σ_{sk} is an expansion of the signature of σ so the entailment claim makes sense. It suffices to show that the result of a single Skolem-step on σ entails σ; this is very easy to see from the definition. $\square$

In contrast to the classical case we do not have the fact that "σ satisfiable implies σ_{sk} satisfiable." That holds in one-sorted logic because we can always expand a model of σ to properly interpret the Skolem functions and make σ_{sk} true, but this expansion is not always possible in the presence of empty sorts.

Example 1. *Let σ be $(\exists x^A\ .\ (x = x) \vee \exists y^B\ .\ (y = y)\ ) \wedge (\forall z^A\ .\ (z \neq z))$. Then σ is satisfiable but its Skolemization $((a = a) \vee (b = b)) \wedge (\forall z^A\ .\ (z \neq z))$ is not.*

The phenomenon in Example 1 is essentially the only thing that can go wrong: models can be expanded to interpret Skolem functions if we do not existentially quantify over empty sorts. This points the way to recovering a weak version of the classical equi-satisfiability result which will be good enough for our present purposes.

Definition 4.2.2. *A model $\mathbb{M}$ is* safe for *formula ϕ if for every occurrence of a subformula $\exists x^A \, . \, \alpha$ in ϕ we have $\mathbb{M}_A \neq \emptyset$.*

Lemma 4.2.3. *If $\mathbb{M} \models \sigma$ and $\mathbb{M}$ is safe for σ then there is an expansion $\mathbb{M}^*$ of $\mathbb{M}$ to the signature of σ_{sk} such that $\mathbb{M}^* \models \sigma_{sk}$.*

Proof. It suffices to show that the corresponding result holds for a single Skolem-step on σ entails σ, so suppose σ and σ' are as in Definition 4.2.1. The argument is just as for classical one-sorted logic: we can expand the model $\mathbb{M}$ to interpret the new function symbol f precisely because $\mathbb{M}$ satisfies the the original σ, but we must know that A is non-empty in case the truth of $\exists x^A.\sigma(x, x_1, \ldots, x_n)$ is needed for this. □

Definition 4.2.3. *Let ϕ be a formula. An* approximation *of ϕ is a formula obtained by replacing some (zero or more) subformulas $\exists x^A \, . \, \alpha$ of ϕ by $\perp$ (a propositional constant, interpreted as falsehood in every model)*

It is not hard to see that if $\sigma_\perp$ is an approximation of σ then $\sigma_\perp \models \sigma$.

Lemma 4.2.4. *If $\sigma_\perp$ is an approximation of σ then $\sigma_\perp \models \sigma$.*

Proof. We prove by induction over arbitrary formulas ϕ and approximations $\phi_\perp$, and for arbitrary models $\mathbb{M}$ and environments η, if $\mathbb{M} \models_\eta \phi_\perp$ then $\mathbb{M} \models_\eta \phi$. Suppose $\mathbb{M} \models_\eta \phi_\perp$.

- ϕ is a literal: then $\phi_\perp = \phi$. So certainly $\mathbb{M} \models_\eta \phi$.
- $\phi \equiv \alpha \vee \beta$: then $\phi_\perp = \alpha_\perp \vee \beta_\perp$, where $\alpha_\perp$ and $\beta_\perp$, are approximations of α and β. Then $\mathbb{M} \models_\eta \alpha_\perp$ or $\mathbb{M} \models_\eta \beta_\perp$ so by induction $\mathbb{M} \models_\eta \alpha$ or $\mathbb{M} \models_\eta \beta$, as desired.
- $\phi \equiv \alpha \wedge \beta$: then $\phi_\perp = \alpha_\perp \wedge \beta_\perp$, where $\alpha_\perp$ and $\beta_\perp$, are approximations of α and β. Then $\mathbb{M} \models_\eta \alpha_\perp$ and $\mathbb{M} \models_\eta \beta_\perp$ so by induction $\mathbb{M} \models_\eta \alpha$ and $\mathbb{M} \models_\eta \beta$, as desired.
- $\phi \equiv \forall x^A \alpha$: then $\phi_\perp = \forall x^A \, . \, \alpha_\perp$, where $\alpha_\perp$ is an approximation of α. For every $e \in \mathbb{M}_A$ we have $\mathbb{M} \models_{\eta[x^A \mapsto e]} \alpha_\perp$, so by induction at each such e we have $\mathbb{M} \models_{\eta[x^A \mapsto e]} \alpha$, so $\mathbb{M} \models_\eta \forall x^A \, . \, \alpha$.
- $\phi \equiv \exists x^A \alpha$: then either $\phi_\perp = \exists x^A.\alpha_\perp$ or $\phi_\perp = \perp$. In the former case we have, for some $e \in \mathbb{M}_A$, $\mathbb{M} \models_{\eta[x^A \mapsto e]} \alpha_\perp$, so by induction $\mathbb{M} \models_{\eta[x^A \mapsto e]} \alpha$, so $\mathbb{M} \models_\eta \exists x^A \, . \, \alpha$. The latter case cannot arise under the hypothesis that $\mathbb{M} \models_\eta \phi_\perp$.

□

Lemma 4.2.5. *If $\mathbb{M} \models \sigma$ then there is an approximation $(\sigma_\perp^{\mathbb{M}})$ of σ such that $\mathbb{M} \models \sigma_\perp^{\mathbb{M}}$ and $\mathbb{M}$ is safe for $\sigma_\perp^{\mathbb{M}}$.*

Proof. The sentence $\sigma_\perp^{\mathbb{M}}$ is obtained by replacing $\exists x^A \, . \, \alpha$ by $\perp$ precisely when $\mathbb{M}_A = \emptyset$.

Formally we inductively define approximations $\phi_\perp^{\mathbb{M}}$ for arbitrary formulas ϕ as follows.

- ϕ is a literal: then $\phi_\perp^{\mathbb{M}} = \phi$.
- $\phi \equiv \exists x^A \alpha$ and $\mathbb{M}_A = \emptyset$: then $\phi_\perp^{\mathbb{M}} = \perp$
- $\phi \equiv \exists x^A \alpha$ and $\mathbb{M}_A \neq \emptyset$: then $\phi_\perp^{\mathbb{M}} = \exists x^A.\alpha_\perp^{\mathbb{M}}$.
- $\phi \equiv \forall x^A \alpha$: then $\phi_\perp^{\mathbb{M}} = \forall x^A \, . \, \alpha_\perp^{\mathbb{M}}$.

- $\phi \equiv \alpha \vee \beta$: then $\phi_\perp^{\mathbb{M}} = \alpha_\perp^{\mathbb{M}} \vee \beta_\perp^{\mathbb{M}}$.
- $\phi \equiv \alpha \wedge \beta$: then $\phi_\perp^{\mathbb{M}} = \alpha_\perp^{\mathbb{M}} \wedge \beta_\perp^{\mathbb{M}}$.

It is clear from the construction that $\mathbb{M}$ is safe for $\phi_\perp^{\mathbb{M}}$. We now claim that for an arbitrary environment η, if $\mathbb{M} \models_\eta \phi$ then $\mathbb{M} \models_\eta \phi_\perp^{\mathbb{M}}$. But this is a straightforward induction over formulas ϕ. The lemma follows by taking ϕ to be σ.

□

Lemma 4.2.6. *If σ is satisfiable then there exists an approximation $\sigma_\perp$ of σ such that $\sigma_{\perp sk}$ is satisfiable.*

Proof. By Lemma 4.2.5 and Lemma 4.2.3. □

4.3 A Finite Model Theorem for Order-Sorted Logic

Model $\mathbb{M}$ is a *submodel* of model $\mathbb{N}$ if (i) for each sort A, $\mathbb{M}_A \subseteq \mathbb{N}_A$ and (ii) each $f^{\mathbb{M}}$ and $R^{\mathbb{M}}$ are obtained as the restrictions of $f^{\mathbb{N}}$ and $R^{\mathbb{N}}$ to $\mathbb{M}$. Note that we use "submodel" in this strong sense rather than just requiring each $R^{\mathbb{M}}$ to be a subset of $R^{\mathbb{N}}$ (as is done by some authors).

If $X = \{X_A \mid A \in \mathcal{S}\}$ is a family of sets with $X_A \subseteq \mathbb{M}_A$ for each $A \in \mathcal{S}$ then we say that X is closed under a function $g : \mathbb{M}_{A_1} \times \cdots \times \mathbb{M}_{A_n} \to \mathbb{M}_A$ if whenever $(a_1, \ldots, a_n) \in X_{A_1} \times \cdots \times X_{A_n}$ we have $g(a_1, \ldots, a_n) \in X_A$. Note that this is a stronger claim than saying that the single set $\bigcup X$ is closed under g.

Lemma 4.3.1. *Let $h : \mathbb{P} \to \mathbb{M}$ be a homomorphism between models of $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. There is a unique submodel of $\mathbb{M}$ with universe $\{h_A(\mathbb{P}_A) \mid A \in \mathcal{S}\}$.*

Proof. It is easy to check that the family $\{h_A(\mathbb{P}_A) \mid A \in \mathcal{S}\}$ is closed under the interpretations in $\mathbb{M}$ of the function symbols in Σ. So if we define the interpretations of the relation symbols in Σ to be the restriction of the interpretations in $\mathbb{M}$ the result is a submodel. Since there is no choice in the interpretations of the symbols in Σ once the universe $\{h_A(\mathbb{P}_A) \mid A \in \mathcal{S}\}$ is determined, uniqueness follows. □

We will denote the submodel identified in Lemma 4.3.1 as $h(\mathbb{M})$.

Remark 4.3.1. *For future reference we observe that if $\mathbb{P}$ is a submodel of $\mathbb{M}$ and $e \in \mathbb{M}_B$ then it need not be the case that $e \in \mathbb{P}_B$ even if $e \in \bigcup\{\mathbb{P}_A \mid A \in \mathcal{S}\}$. Indeed this can happen even when $\mathbb{P}$ is obtained as the image of a homomorphism into $\mathbb{M}$. This has important consequences for the use of sorts as predicates, as we will discuss in Section 4.5.*

Next we establish the fundamental fact about preservation of universal sentences under submodel. The proof is a straightforward induction.

Theorem 4.3.1. *Let σ be a sentence that is existential-free and in negation-normal form and let $\mathbb{M}'$ be a submodel of $\mathbb{M}$. If $\mathbb{M} \models \sigma$ then $\mathbb{M}' \models \sigma$.*

Definition 4.3.1 (The *kernel* of a model)**.** *Let $\mathbb{M}$ be a model for the signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$. Let h be the unique homomorphism from $\mathcal{T}^{\mathcal{L}}$ to $\mathbb{M}$ (c.f. Theorem 2.0.1). The image of h is a submodel of $\mathbb{M}$ by Lemma 4.3.1; this is the* kernel *of $\mathbb{M}$.*

The crucially important fact for us is that for the kernel $\mathbb{K}$ of $\mathbb{M}$ we have, for each sort A, the cardinality of $\mathbb{K}_A$ is bounded by the cardinality of $\mathcal{T}_A^{\mathcal{L}}$, simply because $\mathbb{K}_A$ is the image of $\mathcal{T}_A^{\mathcal{L}}$ under h.

The kernel and the Skolem hull Recall the classical treatment of Skolemization (see *e.g.*, [CK73]): given a model $\mathbb{M}$, let $\mathbb{M}^*$ be a model interpreting the Skolem functions that satisfies the Skolem theory (the sentences saying that the Skolem functions witness the truth of the associated existential formula). Then given a subset X of the universe of $\mathbb{M}$, the Skolem hull $\mathcal{H}_{\mathbb{M}}(X)$ is the smallest subset of the universe containing X and closed under the functions and constants of the enriched language; this determines an elementary submodel $\mathcal{H}_{\mathbb{M}}(X)$ of $\mathbb{M}^*$. In particular $\mathcal{H}_{\mathbb{M}}(\emptyset)$ can be viewed as a "minimal" submodel of $\mathbb{M}$.

But in the order-sorted setting, *the kernel of a model is not in general the same as the Skolem hull.* The latter notion, although perfectly sensible in order-sorted logic, does not play the same role of "minimal" submodel as it does in the one-sorted setting. Indeed it is possible for the kernel of a model to be finite while the Skolem hull is infinite.

Example 2. *Consider $\mathcal{L} = (\{A, B\}, \emptyset, \Sigma)$ with $a \in \Sigma_{\langle\rangle,A}$ and $f \in \Sigma_{B,B}$ the only vocabulary symbols. Let $\mathbb{M}$ have $\mathbb{M}_A = \{b_0 = a^{\mathbb{M}}\}$, $\mathbb{M}_B = \{b_0, b_1, b_2, \ldots\}$, and $f^{\mathbb{M}}$ map b_i to b_{i+1}. Then the Skolem hull $\mathcal{H}(\emptyset)$ of $\mathbb{M}$ is $\mathbb{M}$ itself. Yet the kernel $\mathbb{K}$ of $\mathbb{M}$ is the model of size 1 with $\mathbb{K}_A = \{b_0\}$, $\mathbb{K}_B = \emptyset$, $f^{\mathbb{K}} = \emptyset$.*

Here we present our main theorem.

Theorem 4.3.2. *Let σ be an $\mathcal{L}$-sentence whose Skolemization σ_{sk} has signature $\mathcal{L}^*$. Then σ is satisfiable if and only if σ has a model $\mathbb{H}$ such that for each sort A, the cardinality of $\mathbb{H}_A$ is no greater than the cardinality of $\mathcal{T}_A^{\mathcal{L}^*}$.*

Proof. For the non-trivial direction, suppose σ is satisfiable. By Lemma 4.2.6 there is an approximation $\sigma_\perp$ of σ such that $\sigma_{\perp sk}$ is satisfiable. Let $\mathcal{L}^{**}$ be the signature for $\sigma_{\perp sk}$; note that $\mathcal{L}^{**}$ is a reduct of $\mathcal{L}^*$ and the sentence $(\sigma_\perp)_{sk}$ is existential-free.

Let $\mathbb{M}$ be a model of $(\sigma_\perp)_{sk}$, and let $\mathbb{H}$ be the kernel of $\mathbb{M}$. Since $(\sigma_\perp)_{sk}$ is existential-free, $\mathbb{H} \models (\sigma_\perp)_{sk}$. Since $\mathbb{H}$ is a kernel we have that for each sort A, the cardinality of $\mathbb{H}_A$ is no greater than the cardinality of $\mathcal{T}_A^{\mathcal{L}^{**}}$, and thus no greater than the cardinality of $\mathcal{T}_A^{\mathcal{L}^*}$. Since $(\sigma_\perp)_{sk} \models \sigma_\perp$ and $\sigma_\perp \models \sigma$, the model $\mathbb{H}$ is the desired model of σ. $\square$

Finally we can define precisely the key notion of the paper.

Definition 4.3.2. Order-Sorted Effectively Propositional Logic (order-sorted effectively-propositional logic) *is the class of sentences σ such that the signature of the Skolemization of σ has a finite term model.*

The next section shows how to decide whether a sentence is in order-sorted effectively-propositional logicand if so, to compute the sizes of the sorts in the term model. Taken together with Theorem 4.3.2, this establishes a decision procedure for satisfiability of order-sorted effectively-propositional logicsentences.

4.4 Algorithms

Let $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ be a signature. We say that sort A is *finitary* in $\mathcal{L}$ if $\mathcal{T}_A^{\mathcal{L}}$ is finite. Our membership algorithm reduces the problem of counting terms to one of asking whether a given context-free grammar yields only a finite number of strings; well-known algorithms solve the latter problem. Intuitively, the grammar captures the ground terms that can be generated from the signature.

Definition 4.4.1. *Given a signature $\mathcal{L} = (\mathcal{S}, \Sigma, \leq)$ with multiple sorts, we define a grammar $G_{\mathcal{L}}$ as follows. The set of nonterminals is $\mathcal{S} \cup \{A_0\}$, where A_0 is a fresh symbol not in $\mathcal{S}$, the set of* *terminals is $\bigcup\{\Sigma_{w,S} \mid (w,s) \in \mathcal{S}^* \times \mathcal{S}\}$, and the set of productions comprises:*

$$\begin{aligned} A_0 &\to A && \textit{for each } A \in \mathcal{S} \\ B &\to f A_1 \ldots A_n && \textit{whenever } f \in \Sigma_{\langle A_1 \ldots A_n\rangle, B} \\ B &\to A && \textit{whenever } A \leq B \end{aligned}$$

A non-terminal X in a context-free grammar G is said to be *useful* if there exists a derivation $A_0 \Rightarrow^* \alpha X \beta \Rightarrow^* u$ where u is a string of terminals, otherwise X is *useless.* If A is a useful non-terminal and u is a string of terminals we say that A *generates* u if there is a derivation $A \Longrightarrow^* u$.

Lemma 4.4.1. *Let A be a sort of $\mathcal{L}$ and let u be a string of terminals over $\bigcup\{\Sigma_{w.S} \mid (w,s) \in \mathcal{S}^* \times \mathcal{S}\}$. Then u is a term in $\mathcal{T}^{\mathcal{L}}_A$ if and only if there is a derivation $A \Rightarrow^* u$ in $G_{\mathcal{L}}$. A sort A is inhabited by a ground term if and only if A is useful in the grammar $G_{\mathcal{L}}$. When A is useful as a sort in $L(G_{\mathcal{L}})$, the set $\mathcal{T}^{\mathcal{L}}_A$ is finite if and only if A generates only finitely many terms in $L(G_{\mathcal{L}})$. In particular the set $\mathcal{T}^{\mathcal{L}}$ is finite if and only if $L(G_{\mathcal{L}})$ is finite.*

Proof. The first claim is easy to check: it holds essentially by the construction of $G_{\mathcal{L}}$. The second claim follows from the first and the facts that the u in question are strings of terminals of $G_{\mathcal{L}}$ and we have $A_0 \Rightarrow A$ for each $A \in \mathcal{S}$. □

Theorem 4.4.1. *There is an algorithm that, given an order-sorted signature $\mathcal{L}$, determines (uniformly) for each sort A, whether $\mathcal{T}^{\mathcal{L}}_A$ is finite. The algorithm runs in time linear in the total size of $\mathcal{L}$.*

Proof. By Lemma 4.4.1, $\mathcal{T}^{\mathcal{L}}_A$ is finite if and only if A generates only finitely many terms in in $L(G_{\mathcal{L}})$. There is a well-known algorithm for testing whether a non-terminal in a context-free grammar generates infinitely many terminal strings: after eliminating useless symbols from the grammar $G_{\mathcal{L}}$, form the graph whose nodes are the inhabited sorts, with an edge from B to A if and only if there is a production in $G_{\mathcal{L}}$ of the form $B \to \alpha\ A\ \beta$, that is, if and only if the set $\Sigma_{\langle A_1 \ldots A \ldots A_n\rangle, B}$ is non-empty or if $A \leq B$. Then a non-terminal A generates infinitely many terminal strings if and only if there is a path from A to a cycle. Since the size of $G_{\mathcal{L}}$ is linear in the size of $\mathcal{L}$, the overall complexity of our algorithm is linear in $\mathcal{L}$. □

Example 3. *Return to Equation 4.1 from Section 4.1.. After Skolemizing we have the signature with $b \in \Sigma_{\langle\rangle, B}$ and $f \in \Sigma_{A,B}$ in addition to those constants in the original signature. It is easy to check that the graph constructed for this signature has edges from the node A_0 to A and to B, and an edge from B to A. This graph is acyclic so we conclude that this class of sentences has the finite model property. On the other hand, if we were to postulate that $B \leq A$ (instead of $A \leq B$) then we cannot deduce the finite model property, since our grammar would have the production $A \to B$ in addition to $B \to A$ and the resulting graph would have a cycle.*

Corollary 4.4.1. *Membership in order-sorted effectively-propositional logicis decidable in linear time.*

Proof. Let σ be given, over signature $\mathcal{L}$. We can compute the skolemization σ_{sk} of σ in linear time, and extract the signature $\mathcal{L}^*$ of σ_{sk}. The size of this signature is clearly linear in σ, so by Theorem 4.4.1, we can decide whether all sorts of $\mathcal{L}^*$ are finitary in time linear in σ. □

Note that in the worst case, Σ may induce a number of terms exponential in its size. Thus we would like to avoid actually generating the terms, and merely count them if we can do so in polynomial time.

The intuition behind the algorithm is as follows. If a sort is finitary, its terms can be of height no greater than the number of functions in Σ. So we construct a table containing the number of terms of each height of each sort, starting with constants and then applying functions. The only complication is that when counting the ways to create a new term of height h using function f, we need to make certain that each has at least one subterm of height *exactly* $h-1$.

Theorem 4.4.2. *There is an algorithm that, given a signature $\mathcal{L}$, computes, in time cubic in the size of $\mathcal{L}$, the size of $\mathcal{T}_A^{\mathcal{L}}$ for each finitary sort A (returning "∞" for the non-finitary sorts).*

Proof. The algorithm is given as Algorithm 4.4 below. A sketch of the algorithm follows:

Lines 1 through 3 perform initialization. Lines 4 through 7 fill the first row of the table with the number of *constant* terms in each sort, remembering to propagate these constants to supersorts of the sorts in which they were declared. The main body of the algorithm, which begins on line 8, fills the rows for terms constructed using functions (and hence having "height" 1, 2, and so on.) At each height, a flag is set (lines 9 and 20–21) which indicates whether terms of that height have been found.

Line 10 begins iteration for every function f in the vocabulary Σ, attempting to count the number of terms of height h which have f at their head; a running total is kept via the *ways* variable, which is initialized on line 11.

To be of height h, a term must contain at least one subterm of height exactly $h-1$; exact counts of these terms have already been computed in the prior row. Line 12 checks each component of f's arity, counting the number of terms that have a $h-1$ height sub-term in that component. Line 13 initializes the counter to the number of $h-1$ height subterms available for this component, and lines 14–17 factor in the options for the other components. Finally, line 18 adds the subtotal to the running total for height h and function f. Once all options for f-headed, h-height terms have been explored, lines 22–23 propagate those options to the supersorts of f's declared output sort.

Finally, the algorithm confirms that terms of height h exist (lines 24–25). If not, it can safely terminate, since terms of height $h+1$ would require terms of height h to construct. The algorithm's result is the sum over each sort-column.

Proof of correctness Since the algorithm uses only FOR loops with finite arguments, it is easy to see that it terminates. Furthermore we claim that after termination, the totals of each column $\mathsf{Tbl}[\sum][A]$ contain exactly $|\mathcal{T}_A^{\mathcal{L}}|$ for each finitary sort A.

First observe that by the pigeonhole principle, all terms in $\mathcal{T}_A^{\mathcal{L}}$ (A finitary) must have *height* $\leq \mathsf{n_f}$. Therefore, when counting terms in finitary sorts it suffices to count only terms of *height* $\leq \mathsf{n_f}$, and thus we need only prove that the algorithm populates the table correctly.

Theorem 4.4.3. *For each h, A, $\mathsf{Tbl}[h][A]$ contains exactly the number of terms having height h within $\mathcal{T}_A^{\mathcal{L}}$.*

Proof. After a row is computed by our algorithm, it is never again modified. So we proceed by induction on h.

Base: If $h=0$ then we are concerned only with constant terms. The first block of our algorithm counts every constant $c : S$ exactly once in each $\mathsf{Tbl}[0][S']$ such that $S \leq S'$. So we can conclude that $\mathsf{Tbl}[0]$ contains a faithful count of height 0 terms for all sorts.

Induction: Suppose $h > 0$ and that each $\mathsf{Tbl}[x]$, $0 \leq x < h$ is correct. A non-constant term t is in $\mathcal{T}_A^{\mathcal{L}}$ if (by definition!):

1. t has a function at its head with result sort A
2. t has a function at its head with some result sort B with $B \leq A, B \neq A$.

The algorithm increments a table cell according to case (2) if and only if it has already incremented a cell according to case (1).

Each ground term of height $h > 0$ has one distinct function f at its head, and (with respect to the ordering in f's arity) exactly one left-most subterm of height $h-1$ at index lm, $1 \leq \mathsf{leftmost} \leq n$.

Algorithm 1 The Counting Algorithm

Input: A signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$

Output: For each sort s for which Σ is finitary, the number of terms of sort s.

```
1: $\mathsf{n_s} \leftarrow$ the number of sorts
2: $\mathsf{n_f} \leftarrow$ the number of functions
3: $\mathsf{Tbl}$, an array of size $[\mathsf{n_f}+1][\mathsf{n_s}]$, initialized with zeroes
4: for all $c \in S$ do
5:     $\mathsf{Tbl}[0][S] \leftarrow \mathsf{Tbl}[0][S] + 1$
6:     for all $S'$ such that $S < S'$ do
7:         $\mathsf{Tbl}[0][S'] \leftarrow \mathsf{Tbl}[0][S'] + 1$
8: for $h = 1$ to $\mathsf{n_f}$ do
9:     foundTermsOfThisHeight := false;
10:    for all $f : (A_1, ..., A_n) \to B \in \Sigma$ do
11:        $\mathsf{ways} \leftarrow 0$
12:        for $\mathsf{leftmost} = 1$ to $n$ do
13:            $\mathsf{ways}_n = \mathsf{Tbl}[h-1][\mathsf{leftmost}]$
14:            for $\mathsf{component} = 1$ to $(\mathsf{leftmost} - 1)$ do
15:                $\mathsf{ways}_n * = \sum\{\mathsf{Tbl}[k][\mathsf{component}] \mid 0 \leq k \leq h-2\}$
16:            for $\mathsf{component} = (\mathsf{leftmost} + 1)$ to $n$ do
17:                $\mathsf{ways}_n * = \sum\{\mathsf{Tbl}[k][\mathsf{component}] \mid 0 \leq k \leq h-1\}$
18:            $\mathsf{ways} + = \mathsf{ways}_n$
19:        $\mathsf{Tbl}[h][B] + = \mathsf{ways}$
20:        if $\mathsf{ways} > 0$ then
21:            foundTermsOfThisHeight := true
22:        for all $S'$ such that $B < S'$ do
23:            $\mathsf{Tbl}[h][S'] + = \mathsf{ways}$
24:    if not foundTermsOfThisHeight then
25:        break;
26: Return a 1-dimensional vector of size $\mathsf{n_s}$ which contains the column sums of $\mathsf{Tbl}$
```

So we only need to show that we correctly calculate the number of terms in $\mathcal{T}_A^{\mathcal{L}}$ having height h, head function f with result sort A and left-most $h-1$ subterm at index $\mathsf{leftmost}$

The number of such terms depends only on the number of subterms available to fill each index of f's arity. The number of usable subterms for A_i is:

- If $i = \mathsf{leftmost}$, terms of sort A_i having height exactly $h-1$ are admissible.
- If $i < \mathsf{leftmost}$, terms of sort A_i having height up to $h-2$ are admissible.
- If $i > \mathsf{leftmost}$, terms of sort A_i having height up to $h-1$ are admissible.

(The usable heights differ by index since index $\mathsf{leftmost}$ is the *leftmost* appearance of a height $h-1$ subterm.)

But this is exactly the calculation that the algorithm makes, and by our induction hypothesis, these subterm rows have been calculated correctly. □

Complexity Note that we could optimize the counting algorithm by memoizing column totals, saving us the trouble of repeatedly summing up $\sum\{\mathsf{Tbl}[k][\mathsf{component}] \mid 0 \leq k \leq h-1\}$ and $\sum\{\mathsf{Tbl}[k][\mathsf{component}] \mid 0 \leq k \leq h-1\}$. The code for this is omitted for clarity, but we assume it when calculating the complexity bounds below.

The initial pass for row 0 takes no more than $\mathsf{n_c n_s}$ steps. The loop structure of the main block (with memoization) nests $\mathsf{n_f}$, $\mathsf{n_f}$ once more, the arity of each f, and then (is bounded by) the arity of each f once more. Note that $\mathsf{n_f}$ times the maximum arity is in $O(\Sigma)$. Thus the complexity of the algorithm is $\mathsf{n_c n_s} + \mathsf{n_f}(|\Sigma| * \mathit{maxarity} + \mathsf{n_f n_s})$, which is cubic in the size of the signature.

□

We could use this algorithm to test for finitary signatures as well, since if we continue to iterate term height past $|\Sigma_F|$, there will be an increase in $\mathsf{Tbl}[h][S]$ if and only if S is infinitary. However, it benefits us to know *in advance* which sorts are finitary and which functions never produce ground terms, as that may greatly reduce the size of Tbl.

If we want to know the total number of terms across all sorts (without duplication), it is easy enough to add a counter and increment it on population (not propagation) in the algorithm above.

Summarizing, we have the following sound and complete procedure for testing satisfiability of order-sorted effectively-propositional logicsentences. Given sentence σ, compute its Skolemization σ_{sk}; let $\mathcal{L}^*$ be the signature of σ_{sk}. If the term model $\mathcal{T}^{\mathcal{L}^*}$ is finite then we know that if σ is satisfiable then σ has a model whose universe has cardinalities as given in Theorem 4.3.2. Since these bounds are computable we can effectively decide satisfiability for such sentences.

Remark 4.4.1. *The results of the algorithm in Theorem 4.4.2 can be useful even if not all sorts are finitary. Fontaine and Gribomont [FG03] have implemented an instantiation-based algorithm that takes advantage of the information that certain sorts are guaranteed to have finitely many ground terms. Their algorithm does not do a sophisticated test for this condition, in fact it succeeds only if there are no non-constant terms in the sort in question. Our algorithm here is simple yet will allow their methods to be applicable to a wider class of sentences.*

4.5 About Sorts-as-Predicates

Many formulations of sorted logic, and certain tools, allow sort names to be used as predicate names in formulas. We have not built this into our syntax; this section explains why. We start by considering Herbrand's Theorem:

Theorem 4.5.1 (Herbrand's Theorem for Order-Sorted Logic)**.** *Let $\tau \equiv \forall y_1, \ldots, y_n \,.\, \alpha$ be a sentence with α quantifier-free. Then τ is unsatisfiable iff there is a set $\{\alpha_1, \ldots, \alpha_k\}$ of ground instances of α such that $(\alpha_1 \wedge \cdots \wedge \alpha_k)$ is unsatisfiable.*

The standard model-theoretic proof of Herbrand's Theorem in first-order logic uses in an essential way the fact that every element in the Skolem hull of a model is named by a term. And the theorem holds with the syntactic rules we've used here, in particular when sorts may not be used as predicates. But when that restriction is not adopted, Herbrand's Theorem fails in a dramatic way, with obvious negative consequences for our finite model theorem.

Example 4. *Consider a signature $\mathcal{L} = (\mathcal{S}, \leq, \Sigma)$ with sorts A and B, a constant $b \in \Sigma_{\langle\rangle,B}$ and a function $f \in \Sigma_{A,B}$. Let σ be the following sentence expressing that f is one-to-one but not onto*

$$(\forall x^A \,.\, B(x)) \wedge (\forall y^B \,.\, A(y)) \wedge (\forall x^A \,.\, f(x) \neq b) \wedge (\forall x_1^A x_2^A \,.\, (x_1 \neq x_2) \rightarrow (f(x_1) \neq f(x_2)).$$

Since the first two conjuncts force A and B to be equal, this sentence has only infinite models. But the Herbrand universe for $\mathcal{L}$ is the singleton set $\{b\}$.

What went wrong? The fundamental fact that the truth of existential-free sentences is preserved under submodel, Theorem 4.3.1, fails when sorts are allowed as predicates. This is because in the definition of submodel there is no requirement that elements remain in each sort they inhabit in the original model: if $\mathbb{P}$ is a submodel of $\mathbb{M}$ and element e happens to be in $\mathbb{M}_A \cap \mathbb{M}_B$ then it is possible that, e is in, say, $\mathbb{P}_A$ but not in $\mathbb{P}_B$. So Theorem 4.3.1, crucial to Herbrand's Theorem, as well as to the soundness of our decision procedure, fails at the base case.

A natural response to this might be: if sorts are to be used as predicates then the notion of submodel should be refined to reflect this. In particular we might refine the definition of submodel and insist that an element in the universe of a submodel retain all of the "sort-memberships" it had in the original model. Unfortunately, if we do this something else breaks: the fact that the image of a homomorphism makes a submodel (Lemma 4.3.1). Recall that closure under under the functions of a vocabulary is a property of a *family* of sets (*e.g.* the family of images of sorts in the source model) and not the union of this family. When we put a sort-structure on the union, in the target model, of the images of sorts in the source by retaining the sort-memberships in the target the resulting family of sets can fail to be closed under the interpretations of the function symbols.

Example 5. *Refer to Example 2; consider the unique homomorphism $h : \mathbb{K} \rightarrow \mathbb{M}$, for which h_A maps b_0 to b_0 and $h_B = \emptyset$. The submodel of $\mathbb{M}$ generated by h interprets sort A as $\{b_0\}$, interprets B as $\emptyset$, and is the universe of a submodel of $\mathbb{M}$ (in which f is interpreted as the empty function). But if we were to insist that element b_0 inhabit sort B in the "submodel" induced by h, then we cannot interpret f: it is supposed to be the restriction of $f^{\mathbb{M}}$ to the universe of our submodel, but $f^{\mathbb{M}}(b_0) = b_1$, and b_1 is not in the range of h.*

We stress that these observations are not bound up with our project of trying to build finite models, they are general foundational problems with using sorts as predicates. If we permit sorts to be used as predicates, we must either give up the notion of models being closed under homomorphisms or give up the intuitively compelling model-theoretic result that universal sentences are preserved under submodel.

The solution is to view the use of sorts as predicates as syntactic sugar for formulas in the core language. The construction for de-sugaring is the following. Given σ with subterm $A(t)$ for a sort A, rewrite this to replace $A(t)$ with $(\exists z^A \,.\, z = t)$ (where z is a fresh variable). This is done before passing to negation-normal form, so as a consequence a subformula $\neg A(t)$ will be replaced with $(\forall z^A \,.\, z \neq t)$.

Lemma 4.5.1. *If σ' is obtained from σ by the process described in the previous paragraph then σ' and σ are logically equivalent.*

Proof. A formal proof is straightforward; here we simply point out that the fact that $\mathbb{M}_A$ might be empty does not cause any problems: if $\mathbb{M} \models_\eta A(t)$ this means that $\eta(t) \in \mathbb{M}_A$ and so we can bind z to $\eta(t)$ to witness the truth of $(\exists z^A \,.\, z = t)$; while if $\mathbb{M} \not\models_\eta A(t)$ this means that $\eta(t) \notin \mathbb{M}_A$ and so $\mathbb{M} \models (\forall z^A \,.\, z \neq t)$. □

Example 6. *We revisit Example 4. When the sentence there is de-sugared according to the recipe above we arrive at*

$$(\forall x^A \exists u^B \,.\, u = x) \wedge (\forall y^B \exists w^A \,.\, w = y) \wedge (\forall x^A \,.\, f(x) \neq b) \wedge (\forall x_1^A x_2^A \,.\, (x_1 \neq x_2) \rightarrow (f(x_1) \neq f(x_2))).$$

When this sentence is Skolemized we get a function from A to B and one from B to A; together with the constant b this obviously generates an infinite set of ground terms.

4.6 Related Work

The decidability of the satisfiability problem for the $\exists\forall$ class in pure logic is a classical result of Bernays and Schönfinkel [BS28] in the absence of equality, extended by Ramsey [Ram30] to allow equality. The problem is known to be EXPTIME-complete [Lew80].

Goguen and Meseguer did seminal work [GM92] on order-sorted algebra; order sorted predicate logic was first considered by Oberschelp [Obe89]. Harrison was one of the first to observe that many-sortedness can not only yield efficiencies in deduction but can also support new decidability results. In unpublished notes [Harpt] he presents some examples of this phenomenon, and suggests searching for typed analogs of classical decidability classes, as we have done here. Order-sorted signatures (without relation symbols) can be viewed as tree automata, so the question of whether the set of closed terms is finite can be answered using standard automata techniques. We believe that the algorithm we give for counting terms is new.

Fontaine and Gribomont [FG03], working in "flat" many-sorted logic (*i.e.*, without subsorting) prove that if there are no functions having result sort A and σ is a universal sentence then σ has a model if and only if it has a model in which the size of A is bounded by the number of constants of sort A. This result is used to eliminate quantifiers in certain verification conditions. This theorem has application even when not all sorts are finite and can be used in a setting where some functions and predicates are interpreted. As observed in Remark 4.4.1, the algorithm in Theorem 4.4.2 can be used to broaden application of their techniques.

Claessen and Sorensson [CS03] have integrated a *sort inference* algorithm into the Paradox model-finder that deduces sort information for unsorted problems and, under certain conditions, can bound the size of domains for certain sorts and improve the performance of the instantiation procedure. Order-sorting is not used, and there are restrictions on the use of equality.

Momtahan [Mom05] computes a refutationally-complete upper bound on the size of a single sort (as a function of the user-provided bounds on the other sorts) for a fragment of the Alloy kernel language. The conditions defining this fragment are not directly comparable to ours, but in some respects constrain the sentences rather severely. For example existential quantification in the scope of more than one universal quantifier is usually not allowed.

Abadi *et al.* [ARS10] identify, as we do, a decidable fragment of sorted logic that is decidable by virtue of having a finite Herbrand universe. Although they target Alloy in their examples they work in a many-sorted logic without subsorts or empty sorts; their condition for decidability is the existence of a "stratification" of the function vocabulary; they do not provide algorithms for checking the stratification condition or computing size bounds on the models.

Ge and de Moura [GdM09] present a powerful method for deciding satisfiability modulo theories with an instantiation-based theorem prover. Given a universal (Skolemized) sentence σ they construct a system of set constraints whose least solution constitutes a set of ground terms sufficient for instantiation; satisfiability is thus decidable for the set of sentences for which this solution-set is finite (in the many-sorted setting this subsumes the Abadi *et al.* class). They do not treat empty sorts nor subsorting. They can treat certain sentences that fall outside our order-sorted effectively-propositional logicclass; detection of whether a given sentence falls into their decidable class seems to require solving the associated set-constraints, as compared to our linear-time algorithm. Generally speaking they do detailed fine-grained analysis of individual sentences; we have focused on an easily recognized class of sentences.

The problem of efficiently deciding satisfiability in the EPL class is an active area of research. Jereslow [Jer88] described a "partial instantiation" approach to first-order theorem proving in the EPL fragment, constructing a sequence of propositional instantiations instead of working with the full set of possibilities from the outset. Work by Hooker *et al.* [HRCS02] builds directly on Jereslow's approach (see also many references there). Recent alternatives approaches include [dMB08b] and [LS04]. De Moura and Bjørner [dMB08a] have developed the SMT constraint solver Z3. SMT enriches propositional satisfiability by adding equality reasoning, arithmetic, bit-vectors, arrays, and some quantification. Z3 is used in software verification and analysis applications. De Moura and Bjørner [dMB08b]; and Piskac and de Moura and Bjørner [PdMB08], introduce a DPLL-based decision procedure for the EPL class; this has been implemented as part of Z3. Our work is complementary to these efforts in that it identifies an extended class of sentences to which contemporary techniques can hopefully be applied.

A preliminary version of our finite-model theorem work was presented at the workshop on Synthesis, Verification, and Analysis of Rich Models (SVARM), July 20-21, 2010.

Chapter 5

Disciplined Scenario-Finding

Almost any interesting specification will have many scenarios; in fact, most first-order logic specifications will have an infinite number of them. But even when scenarios are constrained to be finite (for example by the imposition of a size-bound), so that there are only finitely many distinct scenarios, making scenario-finding effective requires focusing on what scenarios to present, in what order, and how to help users navigate them. There is some work on this: for instance, the Alloy Analyzer tries to exclude isomorphic scenarios [TJ07], on the grounds that these present no additional information. Beyond that, however, Alloy lets the underlying SAT-solver dictate the order of presentation, which is effectively unordered, and lets users go from one to the next, again with no semantics associated with the order of presentation.

Alloy does, however, hope to present scenarios in a sensible order. As Jackson's book [Jac12, page 7] explains (*instance* is Alloy's name for a relational model):

> [T]he tool's selection of instances is arbitrary, and depending on the preferences you've set, may even change from run to run. In practice, though, the first instance generated does tend to be a small one. This is useful, because the small instances are often pathological, and thus more likely to expose subtle problems.

In other words, Alloy *wishes* to present the smallest scenarios first. It is therefore natural to ask whether it is possible to force it to do so. There are two difficulties we might encounter: semantic (can this be computed?), and performance (can it be done efficiently?), each of which can be an obstacle.

We present a theory for scenario exploration, which has been implemented in a modified version of Alloy called **Aluminum**.[1]. Aluminum has the following features:

- It *presents minimal* scenarios. Thus, when confronting a scenario given by Aluminum, a user can be confident that every tuple[2] in the scenario is necessary for that scenario to satisfy the specification's constraints. When a user chooses to view another scenario, Aluminum ensures this too is minimal. By browsing the initial set of scenarios, the user can quickly obtain a sense of the scope of scenarios engendered by the specification.
- Aluminum allows the user to *augment* a scenario with a tuple. Here again Aluminum computes a minimal scenario, which includes any other tuples that may necessarily follow from the augmentation.
- For a given scenario and specification, Aluminum *computes the set of tuples consistent with that scenario*—that is, consistent with the specification— but not currently realized. These suggest natural ways to augment the scenario, and hence continue the exploration.

[1]A preliminary version [NSD+13] of this work appeared at the International Conference on Software Engineering.

[2]We use *tuple* to represent an atomic truth (or falsity) instead of the standard logical *fact* in order to avoid potential confusion with Alloy's **fact** keyword.

```
abstract sig Subject {}
sig Student extends Subject {}
sig Professor extends Subject {}
sig Class {
      TAs: set Student,
      instructor: one Professor
}
sig Assignment {
      forClass: one Class,
      submittedBy: some Student
}
pred PolicyAllowsGrading(s: Subject,
                         a: Assignment) {
      s in a.forClass.TAs or
      s in a.forClass.instructor
}
pred WhoCanGradeAssignments() {
      some s : Subject | some a: Assignment |
         PolicyAllowsGrading[s, a]
}
run WhoCanGradeAssignments for 3
```

Figure 5.1: A simple gradebook specification

These features, respectively, comprise the core operations of Aluminum: GenerateMin, Augment, and ConsistentFacts. The precise specifications of these operations, comprising the basic semantics of Aluminum as a tool, are given in Theorem 4.3.2.

Exploration with minimal scenarios results in a different form of traversal of the space of scenarios than what Alloy currently provides. Putting the user in control of exploration is perhaps the chief merit of Aluminum's approach (Section 5.1). We present the theory (Section 5.2), and explain how we implement it atop Alloy (Section 5.3), including a brief discussion on the impact of symmetry-breaking on minimality. We show that users incur minimal performance penalty (Section 5.4), and finally (Section 5.6) examine the user experience that ensues from the minimality-driven approach.

Scenarios, Formally Because the term "scenario" has many informal meanings, it helps to pin down our terminology. A *specification* is a first-order logic description written by a user, e.g., in Alloy syntax. This will include an (Alloy) command to be run: the result of running a command is a set of *models.* Here "model" has its traditional meaning from logic: an assignment of values to variables that makes a formula true. A model can be either *propositional* or *relational*, the latter being the structures appropriate to first-order logic; we need to refer to both, because our specifications are first-order but the underlying SAT-solver produces propositional models. Which one we mean will usually be clear from context, but where necessary we will disambiguate. Finally, a *scenario* is a relational model that is shown to the user. It may thus have embellishments for compelling visual presentation, such as atom names drawn from the specification. Nevertheless, because its semantic content is just a relational model, we will feel free to use "scenario" and "model" interchangeably whenever it is clear that we are in a non-propositional context.

5.1 A Worked Example

Figure 5.1 provides the Alloy specification for a simple gradebook. There are two kinds of users (called `Subjects` in the specification): Student and Professor. A class has one Professor and a set of Students designated as TAs. An assignment is submitted by a set of students for a specific class. The gradebook specifies a policy on who may grade assignments: specifically, professors and TAs may grade assignments in their associated classes.

5.1.1 Scenario Selection in Alloy vs. Aluminum

The specification's **`run`** command asks for scenarios of who can grade assignments. Both Alloy and Aluminum present one scenario at a time; users may request another scenario by clicking the `Next` button. Aluminum produces a three initial scenarios in response, shown in Figure 5.3. In contrast, Figure 5.2 shows the first three scenarios that Alloy produces (out of over 10,000). The order in which each of Alloy and Aluminum produces scenarios is non-deterministic; the scenarios in Figure 5.2 are representative of what we got in several independent runs.

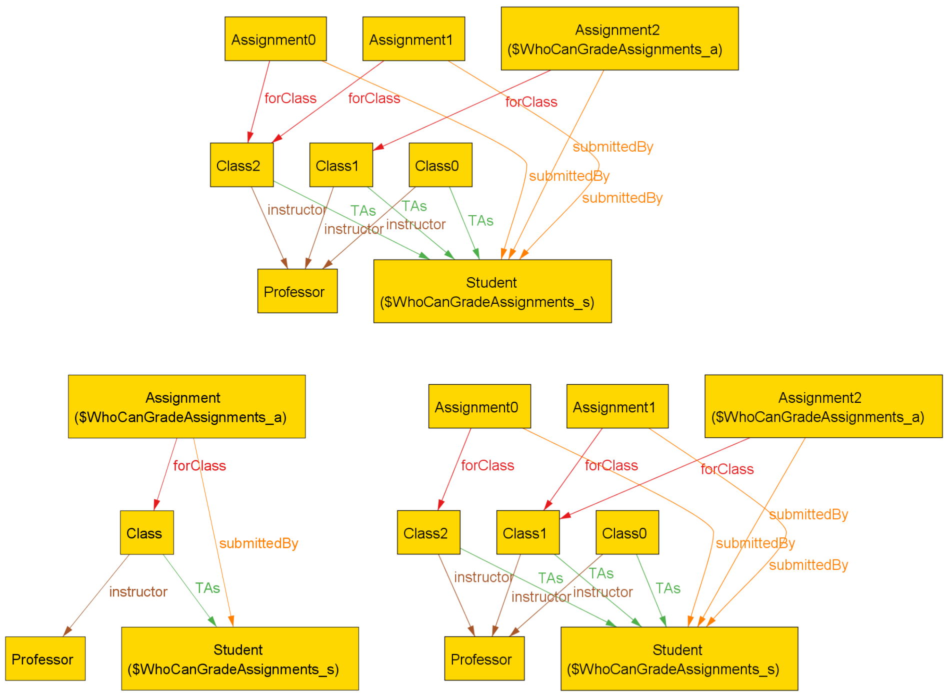

Assignment0
Assignment1
Assignment2 ($WhoCanGradeAssignments_a)
forClass
submittedBy
Class2
Class1
Class0
instructor
TAs
Professor
Student ($WhoCanGradeAssignments_s)
Assignment ($WhoCanGradeAssignments_a)
Class

Figure 5.2: Gradebook scenarios from Alloy

Aluminum's scenarios illustrate three conditions under which one can grade an assignment: (1) the subject is a TA and also the submitter of the assignment, (2) the subject is a TA for the class but not the submitter of the assignment, (3) the subject is a Professor for the class. Each of Alloy's first three scenarios illustrate the first condition, but with additional elements that are not necessary to satisfy the specification (such as two additional classes). Some of Alloy's scenarios (not shown) include additional tuples, such as a second student submitter of some assignment. While these extra elements and tuples *can* exist in a satisfying scenario, they are not *necessary*. Aluminum, in contrast, weeds out all unnecessary tuples, focusing instead on the essence of the scenario.

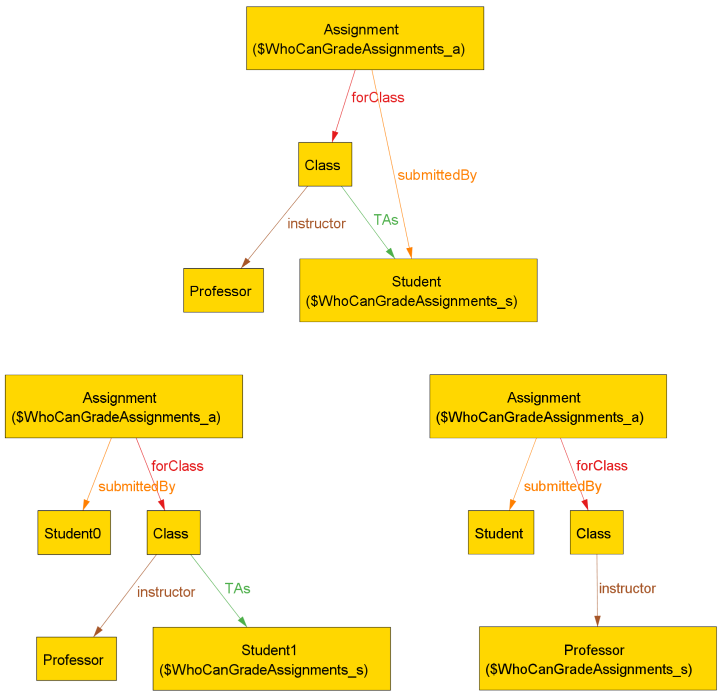

Assignment ($WhoCanGradeAssignments_a)
forClass
submittedBy
Class
instructor
TAs
Professor
Student ($WhoCanGradeAssignments_s)
Assignment ($WhoCanGradeAssignments_a)
forClass
submittedBy
Student0
Class
instructor
TAs
Professor
Student1 ($WhoCanGradeAssignments_s)
Assignment ($WhoCanGradeAssignments_a)
forClass
submittedBy
Student
Class
instructor
Professor ($WhoCanGradeAssignments_s)

Figure 5.3: Gradebook scenarios from Aluminum

For the particular scenarios that Alloy generated on this example, one might posit that the minimality problem is about the domain bounds used in the analysis: had we run the gradebook specification with tighter bounds (1 Class, 1 Professor, 1 Assignment, and 2 Students), Alloy would produce fewer and "tighter" scenarios. Minimality is, however, about more than just setting good domain bounds, for two reasons. First, the bounds only control the number of elements, not the number of tuples; even with tight bounds, non-minimal scenarios can contain unnecessary tuples. Second, by setting bounds too tight, a user can fail to learn about potentially dangerous scenarios, thus gaining inappropriate confidence in their specification. Thus, discovering good bounds is often a process of trial-and-error; minimal scenarios eliminate unnecessary elements and tuples automatically, without placing that burden on the user.

5.1.2 Partitioning the Scenario Space

Each of Aluminum and Alloy embodies a choice of which scenarios to present. Since both tools build upon externally-developed SAT-solvers, their choices are constrained to principles that can be encoded in the propositional formulas on which SAT-solvers operate. Within this constraint, Alloy attempts to partition the scenario space into equivalence classes based on isomorphism and presents one scenario per class (as we discuss in more detail in Sections 5.3 and 5.4)—but chooses the scenario indiscriminately. Aluminum instead organizes scenarios into a partial order based on the tuples they contain and presents the scenarios that are lowest in that ordering.

Figure 5.4 illustrates this fundamental difference between Alloy and Aluminum. The black dots in (A) are scenarios of a specification. The blobs group them into equivalence classes. Part (B) shows how Alloy might present this space: the red dots are representatives of each class; Alloy presents one representative each, and the other members of the class (hollow dots) are not presented. However, there is no ordering to how these representatives are presented, as we have already seen: a fairly complex scenario could precede a very simple one. Part (C) shows

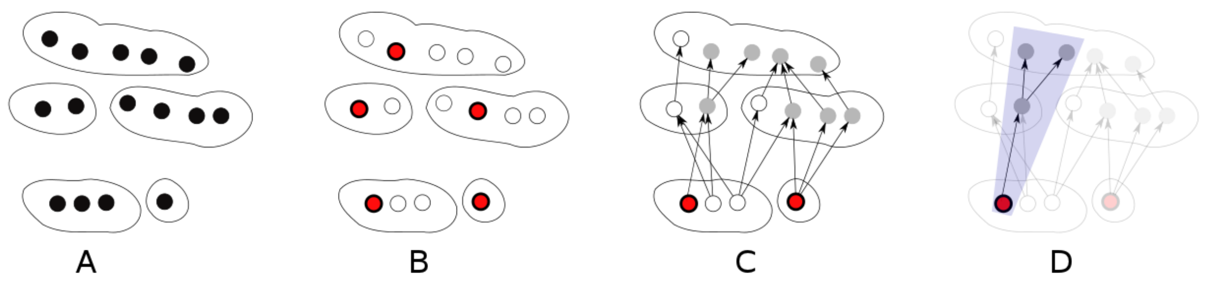


Figure 5.4: How Aluminum and Alloy organize the space of scenarios. Each circle is a scenario, and blobs represent equivalence classes. Red circles are those obtained using the Next button. Hollow circles are those never shown to the user. Arrows represent augmentation operations, and gray circles are scenarios presented after exploration. (A) shows a set of scenarios grouped by isomorphism. (B) shows Alloy's unordered presentation of representatives. (C) shows Aluminum's output, presenting minimal scenarios and augmentation. (D) shows a cone of scenarios.

what Aluminum does (we describe part (D) in Section 5.1.3.) It groups scenarios and picks minimal representatives to show. The Next button shows only minimal scenarios, but Aluminum's augmentation command (described in Section 5.1.3) enables users to find scenarios (the gray dots) with selected tuples added.

As part (C) suggests, augmentation might lead the user to scenarios (the gray circles) that are isomorphic to ones that have been seen previously. We believe showing these scenarios is more sensible than refusing to show a user-constructed scenario just because symmetry-breaking in the original generation procedure would have suppressed it. Nevertheless, unreachable isomorphic scenarios are still excluded, since the user cannot get to them through any operations; this therefore shrinks the exploration space.

5.1.3 Exploration via Augmentation

Aluminum views minimal scenarios as a *starting point* for understanding specifications, but not as sufficient. Alloy users commonly sanity-check specifications with a simple query that says "show me satisfying scenarios". The following query does this for the gradebook specification:

```
  run {some Class} for 3
```

Aluminum produces only one minimal scenario for this query; it contains a single Class and a single Professor who is the instructor for that Class. This is a sufficient scenario because the gradebook specification does not require Students, TAs, or Assignments in a Class. Sanity-checking, however, requires illustrating some of the optional components of a specification. These optional components are not, however, independent: any gradebook scenario that contains an Assignment, for example, must also contain a Student who submitted the Assignment. A systematic technique for exploring the space of scenarios should help the user understand these dependencies.

Aluminum provides two operations that support systematic exploration of the scenario space. Consistent Tuples suggests avenues for exploration by producing a list of all additional tuples that are consistent with the current scenario (and hence can be added to the scenario). Augment adds a user-selected tuple to the current scenario, producing a list of minimal scenarios that include both everything in the current scenario and the new tuple. Returning to our gradebook example, for the scenario showing just a single Class and Professor, Aluminum indicates that the following additions are consistent:

```
Student[NEW(Subject$0)]
```

```
Professor[NEW(Subject$0)]
Class[NEW(Class$0)]
Assignment[NEW(Assignment$0)]
Class.TAs[NEW(Class$0), NEW(Subject$0)]
Class.TAs[Class, NEW(Subject$0)]
Class.instructor[NEW(Class$0), NEW(Subject$0)]
Class.instructor[NEW(Class$0), Professor]
Assignment.forClass[NEW(Assignment$0), NEW(Class$0)]
Assignment.forClass[NEW(Assignment$0), Class]
Assignment.submittedBy[NEW(Assignment$0),
                       NEW(Subject$0)]
```

Each description string contains a relation name and, enclosed in square brackets, a list of atoms. If an atom already exists in the current scenario, it appears by the name given it in the visualizer (e.g., `Class` or `Professor`). If the atom is not present in the current scenario, its descriptor is `NEW`, along with its internal name (`Class$0`, `Subject$0`, etc.).

Visually, it helps to imagine the *cone* of scenarios that extend, and are consistent with, a given scenario. This is shown in part (D) of Figure 5.4. Aluminum's consistent tuple computation shows all the tuples that can inhabit this cone, thereby mapping out the cone's landscape. These tuples may not all, however, necessarily co-habit: a tuple appears in this list if it exists in *some* satisfying scenario that extends the current one, but the presence of some can exclude others, due to specification constraints or domain size bounds.

Suppose a user augments the scenario with the following:

```
Assignment.forClass[NEW(Assignment$0), Class]
```

There is only one extended scenario, and it includes a Student as well as an Assignment. Since Aluminum produces only minimal scenarios (even under augmentation), this tells the user that every addition to this model which adds an Assignment requires adding a Student—a form of deductive reasoning that Alloy does not provide. Asking Aluminum for the consistent tuples for this new scenario suggests some potentially interesting situations:

```
Class.TAs[Class, Student]
Assignment.submittedBy[Assignment, NEW(Subject$0)]
...
```

Specifically, (1) the Student who authored the Assignment could also be a TA in the class, and (2) more than one Student is allowed to submit the same Assignment. Each of these may suggest additional queries to a user. (Aluminum also implements a Backtrack button so users can undo augmentation and continue exploration from a previous scenario.)

Aluminum's combination of minimal scenarios and exploration helps users understand the implications of their specifications in a lightweight manner. Alloy, in contrast, does not offer an exploration mode (much less one based on minimality). If a user wants to see classes with assignments, she must create a new **run** command that adds the assignment constraint. The resulting scenarios may include unnecessary truths; determining the status of each is left to the user to sort out, typically by continuing to refine the query. Furthermore, each change is followed by a new execution, which may start the user out from a completely different initial scenario. By supporting interactive exploration, we feel that Aluminum (a) reduces context switching, (b) reduces "exploration clutter" in the specification, and (c) helps users stay with the same example, which can be lost when executing afresh.

5.1.4 Tuple Provenance

Because of the semantics of Aluminum, when confronted with a particular scenario, a user can understand the provenance of each tuple in it: it is present because it is either (a) part of a minimal scenario, or (b) chosen by a user for augmentation, or (c) the consequence of a user augmentation. While we have not modified Alloy to present this information explicitly, this could easily become part of the user interface.

5.2 Foundations

Models for first-order languages A *(relational) model* for a language L is a map $\mathbb{I}$ binding each relational variable R of L to an actual set-theoretic relation $\mathbb{I}(R)$. If F is sentence of L and $\mathbb{I}$ is an model making F true, we sometimes say that "$\mathbb{I}$ satisfies F" or "$\mathbb{I}$ witnesses F", and write $\mathbb{I} \models F$. If T is a set of first-order sentences we say that $\mathbb{I}$ is a model for T, or $\mathbb{I}$ *satisfies,* or *witnesses,* T if $\mathbb{I}$ satisfies each sentence of T, and we write $\mathbb{I} \models T$.

As Jackson notes [Jac12], one can view Alloy's specification language as first-order logic or relational algebra. A *tuple* over a model $\mathbb{I}$ is given by a n-ary relation R and a sequence $[a_1, \dots, a_n]$ of elements of $\mathbb{I}$ such that $[a_1, \dots, a_n]$ is in $\mathbb{I}(R)$.

If $\mathbb{I}_1$ and $\mathbb{I}_2$ are models, we define $\mathbb{I}_1 \leq \mathbb{I}_2$ to mean that for each relation R, $\mathbb{I}_1(R) \subseteq \mathbb{I}_2(R)$; we write $\mathbb{I}_1 < \mathbb{I}_2$ if for at least one R the inclusion is strict. The *cone* of a model $\mathbb{I}$ is $\{\mathbb{I}' \mid \mathbb{I} \leq \mathbb{I}'\}$: intuitively, this is the set of extensions of $\mathbb{I}$; such an extension can add elements and tuples, but never lose information. This relation is a partial order on the set $\mathrm{Mod}(T)$ of models for any T; we say that a model $\mathbb{I}$ is *minimal* for T if (i) $\mathbb{I} \models T$ and (ii) there is no $\mathbb{I}' \models T$ with $\mathbb{I}' < \mathbb{I}$. If T has any finite models then it has at least one minimal model; in general, of course, T may have several minimal models.

We will often consider models "up to isomorphism": models $\mathbb{I}$ and $\mathbb{I}'$ are isomorphic if they can be made identical by renaming of elements. We write $\mathbb{I}_1 \preceq \mathbb{I}_2$ to mean that, for some $\mathbb{I}_1'$ and $\mathbb{I}_2'$ isomorphic to $\mathbb{I}_1$ and $\mathbb{I}_2$ respectively, $\mathbb{I}_1' \leq \mathbb{I}_2'$.

Models for propositional languages A *(propositional) model* for a given set of atoms for propositional logic is a function M from atoms to $\{0, 1\}$, (*qua* $\{false, true\}$).

If $\mathbb{M}_1$ and $\mathbb{M}_2$ are propositional models, we define $\mathbb{M}_1 \leq \mathbb{M}_2$ to mean that, for each propositional atom p, $\mathbb{M}_1(p) \leq \mathbb{M}_2(p)$; write $\mathbb{M}_1 < \mathbb{M}_2$ if for at least one p, $\mathbb{M}_1(p) < \mathbb{M}_2(p)$. Analogously with relational models, the relation $\leq$ is a partial order on the set of models for any set B of propositional formulas and determines the obvious notion of "cone" over a propositional model. A propositional model $\mathbb{M}$ is minimal for a set B of propositional formulas if it satisfies B and B has no satisfying model $\mathbb{M}' < \mathbb{M}$.

Finite relational models can be encoded as propositional models using standard techniques. Indeed, Alloy's engine, Kodkod [TJ07], translates users' relational specifications to the propositional world, and the results back again for output.

Propositional encoding of specifications Alloy converts a "specification" into a constraint represented as a relational algebra expression encoding the axioms and declarations of an Alloy module together with the predicate for which we seek a (relational) model. Kodkod translates such a constraint into (following the language of [TJ07]) a *Kodkod problem,* which is a triple consisting of a universe of elements, a set of lower- and upper-bounds for each relation symbol in the language, and a relational formula.

Any Kodkod problem P gives rise to a formula $B(P)$ of propositional logic. So given an Alloy specification S, we eventually arrive at a propositional logic formula $B(P_S)$. Moreover – if we remove the secondary variables that Kodkod introduces when translating to conjunctive normal-form – Kodkod also gives us a one-to-one correspondence between relational models of P_S and propositional models of $B(P_S)$. The orderings defined above for relational and propositional models are preserved under this mapping.

Theorem 5.2.1 states our correctness claims about Aluminum. Proofs follow naturally from the algorithms in Section 5.3.

Theorem 5.2.1.

1. *(Completeness of Generation) Let S be an Alloy specification. Procedure* GenerateMin *generates a complete set of minimal models for S up to isomorphism. That is, for any model $\mathbb{I}$ witnessing S,* GenerateMin *will produce some minimal $\mathbb{I}_0$ such that $\mathbb{I}_0 \preceq \mathbb{I}$.*

```
Input: a model $M$ and formula $P$ with $M \models P$
Output: $M'$, a minimal model for $P$
  repeat
    $M' \leftarrow M$
    $M \leftarrow$ Reduce(M)
  until $M = M'$ // Cannot minimize any more
```

Figure 5.5: Minimize Algorithm

2. *(Correctness of Augmentation) Let $\mathbb{I}$ be an model witnessing specification S and let F be a tuple over $\mathbb{I}$. Procedure* Augment *either returns a model $\mathbb{I}'$ such that (i) $\mathbb{I} \preceq \mathbb{I}'$, (ii) $\mathbb{I}'$ satisfies S and F, and (iii) is minimal with respect to these properties, or detects failure if there is no such model.*

3. *(Completeness of Exploration) Let $\mathbb{I}$ and $\mathbb{I}^+$ each witness specification S, with $\mathbb{I} < \mathbb{I}^+$. There is a finite sequence of* Augment *steps leading from $\mathbb{I}$ to $\mathbb{I}^+$.*

4. *(Completeness of Consistency-Checking) Let $\mathbb{I}$ be an model witnessing specification S. Procedure* ConsistentFacts *returns the set of those tuples F such that there is at least one model $\mathbb{I}'$ with $\mathbb{I} \preceq \mathbb{I}'$ with $\mathbb{I}'$ witnessing S and F.*

5.3 Implementation

Aluminum modifies both the Alloy Analyzer's user interface and underlying constraint solver, Kodkod [TJ07]. Aluminum's user interface is based on that of Alloy: the user submits an Alloy specification, Kodkod translates the associated constraint to a propositional formula, the SAT-solver is iterated to produce propositional models, and these are eventually translated to scenarios by Kodkod and rendered by Alloy.

The differences between Alloy and Aluminum lie in the nature of the iterator to produce the initial suite of scenarios, and the facility for user-controlled exploration of the space of scenarios. This section outlines the algorithms underlying the functionality specific to Aluminum.

5.3.1 GenerateMin

The sequence of models produced by Aluminum as a result of the initial execution of the specification is produced by procedure GenerateMin. Given a Kodkod problem P, Aluminum initializes an iterator by invoking a SAT-solver (SAT4J [BP10], in our implementation) to return a model M of P; M and P are then passed to the minimizer.

Minimization consists of repeated calls to Algorithm Reduce (Algorithm 5.6), until there is no change: see Algorithm 5.5. Minimization is not a deterministic process, because a given model can lie in the cone of several different minimal models (as Figure 5.4 (C) and (D) show). This non-determinism manifests itself in practice in Aluminum, because the output of Reduce depends on choices made by the SAT-solver.

After each (minimal) model $\mathbb{M}$ is generated, we of course want to ensure that $\mathbb{M}$ is excluded from future generation. It is straightforward to filter the output to ensure this, as Alloy does. In our setting the problem is somewhat more subtle, since the process of minimization will of course return identical results from quite different models. But in fact this is an opportunity for a significant optimization. After each output model $\mathbb{M}$ is computed, Aluminum adds to the current problem the disjunction of the negations of the atoms true in $\mathbb{M}$. This ensures that no subsequent model from the SAT-solver will be in the cone of $\mathbb{M}$. This is a sound pruning of the model space, since $\mathbb{M}$ represents each model in its cone.

```
Input: model $M$ and formula $B$, with $M \models B$
Output: $M' < M$ with $M' \models B$, if one exists; otherwise $M$
  $C \leftarrow \bigwedge\{\neg p \mid p$ is false in $M\}$
  $D \leftarrow \bigvee\{\neg p \mid p$ is true in $M\}$
  if there is a model $N$ with $N \models B \wedge D \wedge C$ then
    return $N$
  else
    return $M$          // $M$ is minimal
```

Figure 5.6: Reduce Algorithm

Suppressing Isomorphic Models. We have seen how to suppress the generation of identical models, but eliminating *isomorphic models* is much harder. Kodkod tries to avoid generating isomorphs by adding "symmetry-breaking" formulas [Shl07, CGLR96] to the input. This is necessarily heuristic, since no polynomial-time algorithm is known for detecting isomorphisms of relational models.

Unfortunately, symmetry-breaking does not interact well with minimization. For intuition, consider part (C) of Figure 5.4, in which two non-isomorphic models each lie in the cone of the rightmost minimal model in the equivalence class with three entries. When these models are minimized they may return the same minimal representative, depending on choices made by the SAT-solver.

It is even the case that if Aluminum were to use the same mechanism as Kodkod for symmetry-breaking it could incorrectly conclude that a non-minimal model is minimal. Aluminum thus uses a slightly different heuristic for symmetry-breaking. One consequence is that Alloy sometimes eliminates more isomorphic models than Aluminum, a fact which we discuss in Section 5.3.4.

5.3.2 Augment

A key observation is that augmenting a model of a specification by a tuple is merely an instance of the core problem of minimal-model generation: the result of augmentation is precisely an iterator over minimal models of the specification given by the original specification, along with the tuples of the given model plus the new tuple.

One detail is important for performance. Letting B be the conjunctive normal form of the original specification and A_p the Boolean variable representing the augmenting tuple A, the input for model-generation is $B \cup \{p \mid M(p) = 1\} \cup \{A_p\}$. The point here is that since the augmentation is performed entirely at the propositional level, Aluminum avoids the cost of re-translating from relational to propositional logic.

5.3.3 ConsistentFacts

To aid the user in deciding how to explore the space of scenarios by augmentation, Aluminum can determine in advance which new tuples can be consistently added to a given scenario in the context of a given specification. If M is a model of B, we say that atom d is *consistent* with M and B if there is a model of B and d extending M. Aluminum's procedure for computing consistent tuples (modulo translation to and from the propositional level) is given in Algorithm 5.7.

If two unused atoms occur in the same position in otherwise identical consistent tuples, Aluminum presents only one such. For instance, suppose that following two augmentations are consistent:

```
Student[NEW(Subject$0)]
Student[NEW(Subject$1)]
```

```
Input: model $M$ and formula $B$, with $M \models B$
Output: the atoms false in $M$ consistent with $M$ and $B$
  $C \leftarrow \bigwedge\{p \mid p \text{ is true in } M\}$
  $D \leftarrow \bigvee\{p \mid p \text{ is false in } M\}$
  $R \leftarrow \emptyset$
  repeat
    if there is a model $N$ with $N \models B \wedge D \wedge C$ then
      $F \leftarrow \{p \mid N(p) = 1 \text{ and } M(p) = 0\}$
      $R \leftarrow R \cup F$
      $D \leftarrow D - F$
  until no change in $R$
  return $R$
```

Figure 5.7: Consistent Tuples Algorithm

That is, there are two unused Subject atoms, and either of them may be instantiated into Student. In this case, Aluminum will only present the first. Since minimal models often have many unused atoms, this filtering can substantially reduce the number of consistent tuples shown.

5.3.4 Symmetry Breaking in Aluminum

There is one technical issue remaining with our algorithms for minimization and augmentation. The instances of a specification often include several that are identical up to renaming, i.e., are *isomorphic* to one another. After translating to Boolean logic, Kodkod adds a symmetry-breaking formula [CGLR96, Shl07] to eliminate many of these isomorphic copies. Isomorph-elimination yields two benefits. It removes scenarios that provide no new information – "noise" in the output – reducing the burden of scenario-browsing. It can also give a substantial performance improvement on some specifications, especially ones that are unsatisfiable; it effectively reduces the search space. Due to inherent complexity reasons, Alloy does not attempt to remove *all* isomorphs, but its greedy approach is effective in practice.

Symmetry-breaking appears to be fairly straightforward, but it has interesting consequences for producing minimal models. Let the translation of the original spec be α and the symmetry-breaking predicate be β. Kodkod produces models for $\alpha \wedge \beta$. Since β is guaranteed to be satisfied by at least one model in every isomorphism class, this approach is complete (up to renaming) for model-finding. However, β can interfere with our minimization algorithm. If $P = \alpha \wedge \beta$, then the algorithm is guaranteed to produce models minimal for $\alpha \wedge \beta$. But the *original* spec is α alone. Minimizing with respect to $\alpha \wedge \beta$ is too restrictive: reduction may terminate on a model even if it can be further minimized with respect to α. If we did not address this problem then (depending on the order in which candidate models are found) Aluminum could produce models that were not minimal for the original specification.

One option would be to disable symmetry-breaking entirely. This is impractical, since doing so would rob us of both its benefits. Instead, Aluminum finds candidate models to minimize with respect $\alpha \wedge \beta$, but reduces those candidates with respect to α only. This assures that we produce models that are minimal for the spec. Furthermore, since the original candidate models are models of $\alpha \wedge \beta$, we retain much of what symmetry-breaking provides.

The end of every chain of reduction involves an unsatisfiable result, and symmetry-breaking constrains the solver, often making such a result easier to find. This is a potential downside of minimizing without β. However, the reduction cycle introduces a unit clause $\neg p$ for every negative variable p in $\mathbb{M}$. These unit clauses also reduce the state space to be searched, and in practice we have found that disabling β for minimization does not impair our performance. All minimization numbers reported in Section 5.4 were generated using this hybrid approch to minimal-scenario

```
sig U {}
fact {some U}
run {} for 3
```

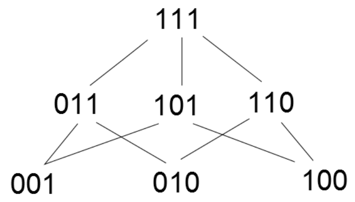


Figure 5.8: A simple specification and its space of models

finding.

But there is a price to pay: even if Alloy's symmetry-breaking were perfect (it is not!), Aluminum could now output isomorphic minimal models. Figure 5.8 illustrates how this can occur. The simple specification in the figure defines a universe of up to three objects, and requires that at least one object exist in a model of the specification. We represent each model with a three-digit binary bitstring indicating which objects are present in the model. For example, bitstring 011 indicates that the model contains objects 2 and 3, but not object 1. The partial order shows models of the specification (it omits the bitstring 000 because this violates the specification).

This model space has three equivalence classes under symmetry: models with one, two, and three objects, respectively. Each equivalence class occupies a row in the figure. This model space has three minimal models: 100, 010, and 001. The minimal models form the bottom row of the lattice; each model in an upper row adds facts to one of the minimal models.

A tool designed around both minimal models and symmetric elimination would initially yield just one model, from the bottom row. Ideally, this is what Aluminum would produce, but the subtleties mentioned above complicate the story. Suppose that all but the lex-leading models (001, 011, and 111) are ruled out by symmetry-breaking. If the first model M provided to our minimization algorithm is 001, then Aluminum will (correctly) stop after presenting that model. However, if the first M provided is 111, the reduction process may terminate with any of 001, 010, or 100, depending on SAT solver behavior. Aluminum will then rule out the models above what it returned and iterate, seeking a new candidate for minimization. If it returned either of the latter two models (010 or 100), there will still be room to find a candidate, and multiple isomorphic minimal models will be produced.

Symmetry-Breaking and Augmentation Symmetry-breaking can also interfere with augmentation and the detection of consistent facts. Consider the following specification:

```
sig U {}
sig P extends U {}
run {} for exactly 2 U
```

The specification has exactly one minimal instance. The instance has two elements $U0$ and $U1$, and neither is in sig `P`. Symmetry-breaking will rule out either the instance where $U0 \in P$ or the one where $U1 \in P$. When performing ordinary model-finding, this is acceptable, but Aluminum allows users to try adding tuples to instances on the fly. It has been noted before [TJ07, MR12] that exposing the underlying set of atoms in an instance (as we do via tuples) changes some standard intuitions of first-order logic. In particular, the "names" of atoms in a Kodkod-produced model do not necessarily behave like constants in a logical formula. This fact will affect which augmentation is possible: if the tuple $U1 \in P$ cannot be added *only because of symmetry-breaking*, yet $U2 \in P$ can be, we risk giving incorrect and confusing results. Because of this problem, we disable symmetry-breaking prior to generating consistent facts or augmenting instances; users are free to explore the entire model space that is reachable from the initial set of minimal models.

Consequences of Aluminum's Handling of Symmetry-Breaking Our implementation separates the original spec (α) and the symmetry-breaking formula (β) after conversation to propositional logic, but before conversion to conjunctive normal-form. Aluminum converts them to CNF separately, which means that we lose the ability to simplify the conjunction, which affects our performance. This can be seen in our performance numbers for the last (unsatisfiable) command in stable mutex ring (c.f. Table 5.2).

As our experience suggests, symmetry-breaking is a subtle operation. The clash between minimization, augmentation, and symmetry-breaking is inherent to the very nature of these operations, independent of the particular techniques used to implement them. Though Aluminum incorporates methods to ameliorate these effects, this is clearly an area for future foundational work.

5.4 Numeric Evaluation

We now compare Aluminum to Alloy numerically. We first study how the resulting scenarios compare mathematically to those produced by Alloy, and then explore how long it takes to compute these minimal scenarios.

We conduct our experiments over the following specifications, with a short name that we use to refer to them in the rest of the paper. (In the tables, where a file contains more than one command, we list in parentheses the ordinal of the command used in our experiments.) The following specifications are taken from the Alloy distribution: Addressbook 3a (Addr), Birthday (Bday), Filesystem (File), Genealogy (Gene), Grandpa (Gpa), Hanoi (Hanoi), Iolus (Iolus), Javatypes (Java), and Stable Mutex Ring (Mutex). In addition, we use three independent specifications: (1) Gradebook (Grade), which is defined in Figure 5.1, and enhanced with two more commands:

```
run WhoCanGradeAssignments for 3 but
1 Assignment , 1 Class , 1 Professor , 3 Student
```

and

```
run {some Class} for 3
```

(2) Continue (Cont), the specification of a conference paper manager, from our prior work. (3) The authentication protocol of Akhawe, et al.'s work [ABL+10] (Auth), a large effort that tries to faithfully model a significant portion of the Web stack.

5.4.1 Scenario Comparison

We consider a set of satisfiable specifications for which we can tractably enumerate *all* scenarios. This lets us perform an exhaustive comparison of the scenarios generated by Alloy and Aluminum. The results are shown in Table 5.1.

The first (data) column shows how many scenarios Alloy generates in all. This number represents one more than the number of times the user can press the `Next` button. Because the Alloy user interface suppresses some duplicate scenarios, this is a smaller number than the number of scenarios produced by Kodkod; we present this smaller number. The second column shows the corresponding number of minimal scenarios presented by Aluminum (the red dots in Figure 5.4 (C)).

The third column shows how many scenarios it takes before Alloy has presented at least one scenario in the cone of every minimal model from Aluminum. Because a given scenario can be in the cone of more than one minimal scenario, the number of scenarios needed for cone coverage may in fact be fewer than the number of minimal models. An important subtlety is that an Alloy

Table 5.1: Alloy's coverage of minimal models and their cones.

Spec.	**Models (Alloy)**	**Models (Aluminum)**	**Cone Coverage**	**Min. Scenario Coverage**	**Ordinal Sum (Alloy)**	**Ordinal Sum (Aluminum)**
Addr	2,647	2	1	5	8	3
Bday (2)	27	1	1	3	3	1
Bday (3)	11	1	1	1	1	1
Gene	64	64	64	64	2,080	2,080
Gpa	2	2	2	2	3	3
Grade (1)	10,837	3	289	10,801	11,304	6
Grade (2)	49	3	2	12	33	6
Grade (3)	3,964	1	1	105	105	1
Hanoi (1)	1	1	1	1	1	1
Hanoi (2)	1	1	1	1	1	1
Java	1,566	3	374	1,558	4,573	6

scenario may not fall in a new cone, but may be isomorphic to one that does (i.e., Alloy may have produced a hollow circle in Figure 5.4). We use Kodkod's symmetry-breaking algorithm to try to identify such situations and "credit" Alloy accordingly.

The fourth column shifts focus to minimal scenarios. If a user is interested in only minimal scenarios, how many scenarios must they examine before they have encountered all the minimal ones (up to isomorphism-detection)? This column lists the earliest scenario (as an ordinal in Alloy's output, starting at 1) when Alloy achieves this.

This number does not, however, give a sense of the distribution of the minimal scenarios. Perhaps almost all are bunched near the beginning, with only one extreme outlier. To address this, we sum the ordinals of the scenarios that are minimal. That is, suppose Aluminum produces two minimal models. If the second and fifth of Alloy's models are their equivalents, then we would report a result of $2 + 5 = 7$. The fifth and sixth columns present this information for Alloy and Aluminum, respectively. The sixth column is technically redundant, because its value must necessarily be $1 + \cdots + n$ where n is the number of Aluminum models; we present it only to ease comparison.

To ensure understanding, let us examine two rows in detail:

Addr: Aluminum finds two minimal models. The very first Alloy model is in both their cones, so the cone coverage value is lower than the number of minimal models. Such a model clearly cannot itself be minimal; indeed, since the minimal scenario coverage requires 5 models and the Alloy ordinal sum is 8, the 3rd and 5th Alloy models must have been (equivalent to) Aluminum's minimal ones.

Grade (3): Aluminum finds just one minimal model. Thus, any model found by Alloy (including the first one) must be in its cone, so the cone coverage value is 1. However, it takes Alloy another 104 models before the minimal one is found.

We can now see Aluminum's impact on covering the space of scenarios. Even on small examples such as Grade (2), there is a noticeable benefit from Aluminum's more refined strategy. This impact grows enormously on larger models such as Java and Grade (1). We repeatedly see a pattern where Alloy gets "stuck" exploring a strict subset of cones, producing numerous models that fail to push the user to a truly distinct space of scenarios. Even on not very large specifications (recall that Grade is presented in these pages in its entirety), it often takes hundreds of `Next`s before Alloy will present a scenario from a cone it has not shown earlier. The real danger here is that the user will have stopped exploring long before then, and will therefore fail to observe an important and potentially dangerous configuration. In contrast, Aluminum presents these at the very beginning, helping the user quickly get to the essence of the scenario space.

Table 5.2: Relative times (ms) to render an unsatisfiable result.

Spec.	Aluminum		Alloy		d
	Avg	σ	Avg	σ	
Bday (1)	9	8	5	3	1.57
Cont (6)	1	<1	1	<1	-0.37
Cont (8)	5	6	3	<1	5.88
File (2)	5	5	6	4	-0.28
Iolus	5,795	239	5,000	177	4.49
Mutex (3)	18,767	52	8,781	60	165.91

Table 5.3: Relative times (ms) per scenario (minimal, in Aluminum).

Spec.	Aluminum			Alloy			d
	Avg	σ	N	Avg	σ	N	
Addr	13	12	2	8	6		0.89
Auth	800	32		110	36		19.35
Bday (2)	9	9	1	5	6		0.59
Bday (3)	8	6	1	7	6		0.17
Cont (3)	4	2	3	1	<1	9	4.86
File (1)	16	13		7	4		2.39
Gene	10	5		6	5		0.85
Gpa	9	4	2	6	5	2	0.62
Grade (1)	6	6	3	3	3		1.10
Grade (2)	3	4	3	3	4		0.01
Grade (3)	8	6	1	4	4		0.96
Java	5	4	3	2	2		1.11
Hanoi (1)	2,509	11	1	1,274	1,239	1	1.00
Hanoi (2)	14	3	1	10	2	1	2.14

The Gene specification presents an interesting outlier. The specification is so tightly constrained that Alloy can produce *nothing but minimal models*! Indeed (and equivalently), it is impossible to augment any of these models with additional tuples, as we will see in Table 5.4.

One important caveat is that these experiments were conducted with one particular SAT-solver, using Alloy's default parameters. Different SAT-solvers and parameters will likely have different outcomes, and studying this variation is a worthwhile task. Nevertheless, the above numbers accurately represent the experience of a user employing Alloy (version **4.2**) in a state of nature.

5.4.2 Scenario Generation

Having examined the effectiveness of Aluminum, we now evaluate its running time (the space difference is negligible). All experiments were run on an OS X 10.7.4 / 2.26GHz Core 2 Duo / 4Gb RAM machine, using SAT4J version `2.3.2.v20120709`. We handled outliers using one-sided Winsorization [Wil12] at the 90% level. The times we report are obtained from Kodkod, which provides wall-clock times in *milliseconds* (ms). All experiments were run with symmetry-breaking turned on. Numbers are presented with rounding, but statistical computations use actual data, so that values in those columns do not follow precisely from the other data shown. Every process described below was run thirty (30) times to obtain stable measurements.

To measure effect strength, we use Cohen's d [Wil12]. Concretely, we subtract Alloy's mean from that of Aluminum, and divide by the standard deviation for Alloy. We use Alloy's in the denominator because that system is our baseline.

Because Aluminum slightly modifies Kodkod to better support symmetry-breaking, we begin by measuring the time to translate specifications into SAT problems. Across all these specifications, Aluminum's translation time falls between 81% and 113% that of Alloy, i.e., our modification has no effective impact. The absolute translation times range from 5ms (for Gradebook) to 55,945ms (for Auth). (We use commas as separators and our decimal mark is a point. Thus 55,945ms = 55.945s = almost 56 seconds.)

Though our focus is on the overhead of minimization, in the interests of thoroughness we also examine unsatisfiable queries. Table 5.2 shows how long each tool spends in SAT-solving (ignoring translation into SAT, and then presentation) to report that there are no scenarios. The d values show that in some cases, the time to determine unsatisfiability is much worse in Aluminum. (It is very important for the reader to remember that the d value is measuring the effect size, *not* the ratio of average times! Thus, a d of almost 166 still results in only a 2.1x increase in time.) The effect is because of the way Aluminum and Alloy handle symmetry-breaking: Aluminum splits the formula produced by Kodkod into two parts, one representing the specification and query and the other capturing symmetry-breaking, whereas Alloy keeps the formulas conjoined. The conjoined formula offers greater opportunities for optimization, which the SAT-solver exploits. Nevertheless, we note that even in some of the large effects, the *absolute* time difference is relatively small.

For satisfiable queries, we calculate the time to compute the first ten scenarios (equivalent to clicking Next nine times), which we feel is a representative upper-bound on user behavior. When the tool could not find ten scenarios, the N column shows how many were found (and the average is computed against this N instead).

When queries are satisfiable, Aluminum's performs well compared to Alloy. First, the overall running time is small for both tools, so even large effect sizes have small wall-clock impact. Indeed, in the most extreme case, Aluminum takes only about 1.2 seconds longer, for a total time of 2.5 seconds—surely no user can read and understand a scenario in less time than that, so Aluminum could easily pre-compute the next scenario. Second, in many cases Aluminum offers many fewer scenarios than Alloy, helping users much more quickly understand the space of models. Finally, the time taken by Kodkod to create the SAT problem can be vastly greater than that to actually solve it, which suggests that a more expensive SAT-solving step will have virtually no perceptible negative impact on the user experience.

We observe two outliers in the data. First, the time for minimization for Auth is very significant. For this specification, we found that the number of extraneous tuples eliminated during minimization is 78 on average. This shows a direct trade-off: 0.7 seconds in computing time for a possibly great impact on user comprehension. Second, the standard deviation for Alloy on Hanoi (1) looks enormous relative to the mean. This is because Kodkod is producing a second duplicate model (which in fact is discarded by the Alloy user interface) very quickly. This results in datapoints that take a long time for the first solution but close to zero for the second.

We also examine the time taken by Aluminum's exploration features: how long it takes to compute the consistent tuples, and how long it takes to augment a scenario. Since the complete space of models is enormous, we restrict attention to the first set of scenarios produced by Aluminum (i.e., the red circles in the bottom row of Figure 5.4 (C)).

Table 5.4 shows the consistent tuples times. We compute up to ten scenarios (five, in the case of Auth) and for each of these we determine the number of consistent tuples. The first column indicates the number of consistent tuples found, averaged over the number of minimal scenarios (i.e., consistent tuples per model). The zero-values are not errors: they arise because the specifications are sufficiently constrained that there is no room for augmentation beyond the initial scenarios. The next two columns show how long it took to compute the number of consistent tuples. These numbers are clearly very modest. Auth is the exception: it produces models with, on average, over 1,300 consistent tuples, more than can be explored by hand. Implementing strategies for helping the user in such situations is an interesting topic for future work.

The last column shows how long, on average, it takes to augment a scenario with a consistent

Table 5.4: Times (ms) to compute consistent tuples and to augment scenarios.

Spec.	**# Cons. Tupls**	**Cons. Tup. Time Avg**	**σ**	**Aug. Time**
Addr	57	45	25	4
Auth	1,335	106,456	17,268	194
Bday (2)	20	19	24	6
Bday (3)	8	9	10	2
Cont (3)	2	5	<1	3
File (1)	24	29	16	3
Gene	0	2	<1	N/A
Gpa	0	1	<1	N/A
Grade (1)	25	13	10	1
Grade (2)	6	2	1	1
Grade (3)	36	14	7	1
Java	25	17	11	2
Hanoi (1)	0	4	<1	N/A
Hanoi (2)	0	1	1	N/A

tuple (backtracking between each augmentation). The "N/A"s correspond to specifications that have no more consistent tuples, and hence cannot be augmented. This too takes very little time.

In Figure 5.9, we show the relationship between the performance cost of scenario minimization versus the gain, i.e. the factor by which Aluminum condenses the model space. Each point represents one of our satisfiable use-cases from Table 5.3 for which we have scenario information in Table 5.1. We plot the Cohen's d for each example in the X-axis versus the collapsing factor (i.e., the number of Alloy scenarios divided by the number of Aluminum scenarios) on the Y-axis. We see three broad clusters: the examples which are already highly constrained and thus gain little from minimization near $y = 1$, three examples with a modest gain around $y = 20$, and several examples with a high reduction factor around $y = 1000$. Even in these high-gain cases, the cost of minimization is fairly low, with a d-value near only one standard deviation; even when the benefits of minimization are enormous, the cost is sometimes quite small.

5.5 Related Work

Logic programming languages produce single, *least*, models as a consequence of their semantics. Because user specifications are not limited to the Horn-clause fragment, Aluminum can offer no such guarantee. The more general notion of minimal model presented here has already been used in specifying the semantics of disjunctive logic programming [LMR92] and of database updates [FUV83] and in non-monotonic reasoning, especially circumscription [Rob01].

The development of algorithms for the *generation* of relational models is an active area of research. The two most prominent methods are "MACE-style" [McC01], which reduce the problem to be solved into propositional logic and employ a SAT-solver, and "SEM-style" [ZZ95], which work directly in first-order logic. The goals of these works are mostly orthogonal to ours, since their concern is usually not the *exploration* of the space of all models of a theory. Generation of minimal models specifically usually relies on dedicated techniques, often based on tableaux (e.g., [Nie96]) or hyperresolution (e.g., [BY00]). Since we are more concerned with exploration as an enhancement of established software design methodology, we made a choice to work with the SAT-solver technology bundled with a tool (Alloy) designed for presenting models.

Koshimura et al. [KNFH09] uses minimality of propositional models to optimally solve job-scheduling problems. Our minimization algorithm is essentially identical to theirs. However,

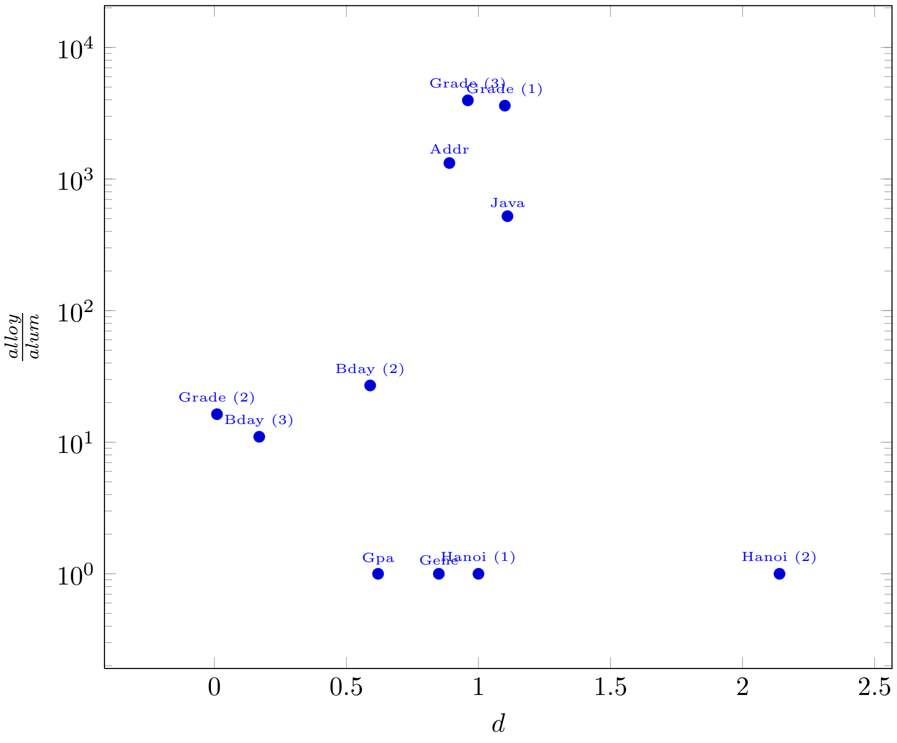


Figure 5.9: Scatterplot of the performance cost of Aluminum versus Alloy (in Cohen's d) versus the reduction factor in number of scenarios presented. Does not include Auth, Bday, Cont, or File, for which there were too many scenarios to count and thus were not presented in Table 5.1. The Y-axis has been logarithmically scaled.

their algorithms do not make use of symmetry-breaking, nor do they address augmentation or consistent-tuple generation.

Janota [Jan10] offers an algorithm to compute all minimal models of a formula once one is computed, and shows how to compute a minimal model by modifying a SAT-solver satisfying a certain technical property. Our technique works with unmodified solvers; it would still be interesting to compare the performance of Janota's method with ours. Another instance of changing the solver would be to use a specialized solver like Max-SAT. We opted to use an ordinary SAT-solver (SAT4J) because using Max-SAT would have involved significant changes that might have introduced confounding factors that would complicate a side-by-side evaluation.

The goals of the Cryptographic Protocol Shapes Analyzer [DGT07] are closely aligned with ours. In analyzing protocols, it too generates minimal models. However, its application domain and especially algorithms are quite different from ours.

5.6 Qualitative Impressions

We have run Aluminum on many representative uses of Alloy beyond those reported in Section 5.4. Here, we report on qualitative impressions. Instead of dwelling on successes, we focus on those aspects of Aluminum that are interesting, potentially controversial, and certainly deserving of future work (such as rigorous user studies).

Verification versus Realization There is an (informal) distinction between two modes of exploring a specification: *verification*, where the user has a property to check (and whose failure yields a counter-example), versus *realization*, in which a user wants to see which scenarios are compatible with a specification. Minimality and orderly growth appear especially useful for the former (where the pithiest counterexample is often the most useful) but perhaps less so for the latter, since the scenarios presented are less likely to surprise the user. Adding some of the other features discussed below would ameliorate this weakness.

Negative Information Aluminum's consistent tuple enumeration does not list what *cannot* become true. This negative information can be vital in comprehending scenarios, and should ideally be incorporated into the exploration process.

Random Exploration Using exploration, an Aluminum user can *eventually* reach any scenario that Alloy would have generated (Theorem 4.3.2). But our experience suggests that there is an important benefit to Alloy's more random generation of scenarios: richer scenarios raise situations that a user didn't think to ask about, and can thus be insightful and thought-provoking. Perhaps Aluminum could add a Surprise Me! button that presents a random, non-minimal model to encourage adventitious discovery (or Alloy could be enhanced with Aluminum's minimization and exploration facilities).

Transition Systems and Framing Conditions Framing conditions describe what should not change between states of a system. We found Aluminum good at highlighting some missing frame conditions. For instance, in an address book (as in [Jac12, chapter 2]), if the `add` predicate fails to mention that all existing entries must be retained, the book will contain only the entry just added. Thus, two consecutive `add` operations will starkly illustrate the absence of a frame condition. This absence may be masked by the non-minimal models of Alloy.

However, frame conditions express not only "lower bounds"—what must remain—but also "upper bounds": what must not be added. In the absence of upper-bounds, a scenario-finder is free to add extra domain elements or relational tuples. Alloy will frequently present just such models, alerting the user to the lack of appropriate framing. In contrast, Aluminum is guaranteed to excise such superfluity! These superfluous entries are actually still present in the consistent tuples, but we believe it is too difficult for users to find them there.

5.7 User Study on Scenarios and Minimality

While we have quantitatively evaluated Aluminum's performance (Section 5.4) and discussed our own qualitative observations about its use (Section 5.6), it remains to show that Aluminum is appealing from a user perspective. That is, are users' needs truly met by a combination of minimal scenarios and exploration via augmentation? Thus, we have created a survey with two goals in mind: to help us better understand how users of tools like Alloy use scenarios in practice, and to inform us whether users actually prefer minimal scenarios to larger ones. We attempted to administer the survey at a conference that typically attracts many Alloy users, but the response rate was too low to render meaningful results. Because of this fact, we present the survey as an appendix, rather than as a significant contribution of this dissertation. The survey can be found in its entirety in Appendix A, along with a more detailed exposition on its design.

Chapter 6

Analysis of Router Configurations

Writing a sensible firewall policy from scratch can be difficult; maintaining existing policies can be terrifying. Oppenheimer, Ganapathi, and Patterson [OGP] have shown that operator errors, specifically configuration errors, are a major cause of online-service failure. Configuration errors can result in lost revenue, breached security, and even physical danger to co-workers or customers. The pressure on system administrators is increased by the frenetic nature of their work environment [BKM$^+$04], the occasional need for urgent changes to network configurations, and the limited window in which maintenance can be performed on live systems.

Many questions arise in checking a firewall's behavior: Does it permit or block certain traffic? Does a collection of policies enforce security boundaries and goals? Does a specific rule control decisions on certain traffic? What prevents a particular rule from applying to a packet? Will a policy edit permit or block more traffic than intended? These questions demand flexibility from firewall-analysis tools: they cover various levels of granularity (from individual rules to networks of policies), as well as reasoning about multiple versions of policies (to check the impact of edits). Margrave handles all these and more, offering more functionality than other published firewall tools.[1]

The key to how we apply Margrave to IOS is in our *decomposition* of IOS configurations into simple policies for analysis. Single firewall configurations cover many functions, such as access filtering, routing, and switching. Margrave's IOS compiler generates separate policies for each task, thus enabling analysis of either specific functionality or whole-firewall behavior. Task-specific policies aid in isolating causes of problematic behaviors. Our firewall models support standard and most extended ACLs, static NAT, ACL-based and map-based dynamic NAT, static routing, and policy-based routing. Our support for state is limited to reflexive access-lists; it does not include general dynamic NAT, deep packet inspection, routing via OSFP, or adaptive policies. Margrave has an iptables compiler in development; other types of firewalls, such as Juniper's JunOS, fit our model as well.

A reader primarily interested in a tool description can read Sections 6.1, 6.4, and 6.5 for a sense of Margrave and how it differs from other firewall-analysis tools. Section 6.1 illustrates Margrave's query language and scenario-based output using a multi-step example. Section 6.2 shows how firewall questions map into Margrave. Section 6.3 describes a query-rewriting technique that often improves performance on firewall policies. Section 6.4 presents experimental evaluation on both network-forum posts and an in-use enterprise firewall. Section 6.5 describes related work. Section 6.6 concludes with perspective and future work.

[1] A preliminary version [NBD$^+$10] of this work appeared at the USENIX Large Installation System Administration conference.

Table 6.1: Feature comparison between Margrave and other available firewall-analysis tools. In each cell, ✓ denotes included features; $✓^{\text{nip}}$ denotes features reported by the authors in private communication but not described in published papers; $✓^{-}$ denotes included features with more limited scope than in Margrave; ∼ denotes features that can be simulated, but aren't directly supported; ? denotes cases for which we aren't sure about support. Section 6.5 describes nuances across shared features and discusses additional research for which tools are not currently available. (Table was generated in August 2010.)

	ITVal	Fireman	Prometheus	ConfigChecker	Fang/AlgoSec	Vantage
Which packets	✓	✓	✓	✓	✓	✓
User-defined queries	✓		?	✓	✓	$✓^{\text{nip}}$
Rule Responsibility	✓	?	$✓^{-}$	∼	✓	✓
Rule Relationships	∼	$✓^{-}$	✓	$✓^{-}$	$✓^{\text{nip}}$	✓
Change-impact	?			✓	$✓^{\text{nip}}$	$✓^{-}$
First-order queries	?		?			?
Support NAT	✓		✓	✓	✓	
Support Routing	✓		✓	✓	✓	$✓^{\text{nip}}$
Firewall Networks	✓	✓	✓	✓	✓	$✓^{\text{nip}}$
Language integration						✓
Commercial Tool?	no	no	yes	no	yes	yes

6.1 Margrave in Action on Firewalls

As discussed in Chapter 3, Margrave presents scenarios that satisfy user-specified queries about configuration behavior. Queries state a behavior of interest and optional controls on which data to consider when computing scenarios. In the firewall domain, scenarios contain attributes of packet contents that make the query hold. The extended example in this section highlights Margrave's features on firewall configurations, echoing the different kinds of query laid out in Chapter 3. Table 6.1 summarizes which of these features are supported by other available (either free or commercial) firewall analyzers. The Margrave website [Mar] contains sources for all examples.

For our purposes, a *firewall* encompasses filtering (via access-lists), NAT transformation, and routing; we reserve the term *router* for the latter component. The IOS configuration in Figure 6.1 defines a simple firewall with only filtering. This firewall controls two interfaces (`fe0` and `vlan1`). Each has an IP address and an access-list to filter traffic as it enters the interface; in lines 3 and 7, the number (101 or 102) is a label that associates access rules (lines 9-16) with each interface, while the `in` keyword specifies that the rules should apply on entry. Rules are checked in order from top to bottom; the first rule whose conditions apply determines the decision on a packet. This firewall allows inbound web and mail traffic to the corresponding servers (the `.10` and `.11` hosts), but denies a certain blacklisted IP address (the `10.1.1.2` host). All traffic arriving at the inside-facing interface `vlan1` is allowed. As this filter is only concerned with packets as they arrive at the firewall, our queries refer to the filter as `InboundACL`.

Basic Queries: All firewall analyzers support basic queries about which packets traverse the firewall. The following Margrave query asks for an inbound packet that `InboundACL` permits:

```
EXPLORE InboundACL:Permit(<req>)

SHOW ONE
```
Query 1

```
 1 interface fe0
 2 ip address 10.150.1.1 255.255.255.254
 3 ip access-group 101 in
 4 !
 5 interface vlan1
 6 ip address 192.128.5.1 255.255.255.0
 7 ip access-group 102 in
 8 !
 9 access-list 101 deny ip host 10.1.1.2 any
10 access-list 101 permit tcp
11             any host 192.168.5.10 eq 80
12 access-list 101 permit tcp
13             any host 192.168.5.11 eq 25
14 access-list 101 deny any
15 !
16 access-list 102 permit any
```

Figure 6.1: Sample IOS configuration

`EXPLORE` clauses describe firewall behavior; here, the behavior is simply to permit packets. `<req>` is shorthand for a sequence of variables denoting the components of a request (detailed in Section 6.2):

$$\langle \mathit{ahostname}, \mathit{src\text{-}addr\text{-}in}, \mathit{src\text{-}port\text{-}in}, \mathit{protocol}, ...\rangle.$$

Users can manually define this shorthand within Margrave; details and instructions for passing queries into Margrave are in the tool distribution [Mar]. `SHOW ONE` is an output-configuration command that instructs Margrave to display only a single scenario. The resulting output indicates the packet contents:

```
 1 ********* SOLUTION FOUND at size = 15
 2 src-addr-in: IPAddress
 3 protocol: prot-tcp
 4 dest-addr-in: 192.168.5.10
 5 src-port-in: port
 6 exit-interface: interface
 7 entry-interface: fe0
 8 dest-port-in: port-80
 9 length: length
10 ahostname: hostname-router
11 src-addr-out: IPAddress
12 message: icmpmessage
```

Result

This scenario shows a TCP packet (line 3) arriving on the fast-ethernet interface (line 7), bound for the web server (line 4, with line 11 of Figure 6.1) on port 80 (line 8). The generic `IPaddress` in lines 2 and 11 should be read as “any IP address not mentioned explicitly in the policy”; lines 5 and 6 are similarly generic. The `size=15` report on line 1 indicates that the scenario involved 15 objects overall.

A user can ask for additional scenarios that illustrate the previous query via the command `SHOW NEXT`: Once Margrave has displayed all unique scenarios, it responds to `SHOW NEXT` queries with `no results`.

To check whether the filter accepts packets from the blacklisted server, we constrain `src-addr-in` to match the blacklisted IP address and examine only packets that arrive on the external interface. Both `src-addr-in` and `entry-interface` are variable names in `<req>`. The `IS POSSIBLE?` command instructs Margrave to display false or true, rather than detailed scenarios.

```
EXPLORE
InboundACL:Permit(<req>) AND
10.1.1.2 = src-addr-in AND
fe0 = entry-interface

IS POSSIBLE?
```
Query 2

In this case, Margrave returns false. Had it returned true, the user could have inspected the scenarios by issuing a `SHOW ONE` or `SHOW ALL` command.

Rule-level Reasoning: Tracing behavior back to the responsible rules in a firewall aids in both debugging and confirming that rules are fulfilling their intent. To support reasoning about rule effects, Margrave automatically defines two formulas for every rule in a policy (where *R* is a unique name for the rule):

- *R*_`matches(<req>)` is true when `<req>` satisfies the rule's conditions, and
- *R*_`applies(<req>)` is true when the rule both matches `<req>` and determines the decision on `<req>` (as the first matching rule within the policy).

Distinguishing these supports fine-grained queries about rule behavior. Margrave's IOS compiler constructs the *R* labels to uniquely reference rules across policies. For instance, ACL rules that govern an interface have labels of the form *hostname*-*interface*-`line`#, where *hostname* and *interface* specify the names of the host and interface to which the rule is attached and # is the line number at which the rule appears in the firewall configuration file.

The following query refines query 2 to ask for decision justification: the `EXPLORE` clause now asks for `Deny` packets, while the `INCLUDE` clause instructs Margrave to compute scenarios over the two `Deny` rules as well as the formulas in the `EXPLORE` clause:

```
EXPLORE
InboundACL:Deny(<req>) AND
10.1.1.2 = src-addr-in AND
fe0 = entry-interface
INCLUDE
InboundACL1:Router-fe0-line9_applies(<req>),
InboundACL1:Router-fe0-line14_applies(<req>)

SHOW REALIZED
InboundACL1:Router-fe0-line9_applies(<req>),
InboundACL1:Router-fe0-line14_applies(<req>)
```
Query 3

The `SHOW REALIZED` command asks Margrave to display the subset of listed facts that appear in some resulting scenario. The following results indicate that the rule at line 9 does (at least sometimes) apply. More telling, however, the absence of the rule at line 14 (the catch-all deny) indicates that that rule *never* applies to any packet from the blacklisted address. Accordingly, we conclude that line 9 processes all blacklisted packets.

```
< InboundACL:line9_applies(<req>) >
```
Result

The `INCLUDE` clause helps control Margrave's performance. Large policies induce many rule-matching formulas; enabling these formulas only as needed trims the scenario space. `SHOW REALIZED` (and its dual, `SHOW UNREALIZED`) controls the level of detail at which users view scenarios. The lists of facts that do (or do not) appear in scenarios often raise red flags about firewall behavior (such as an unexpected port being involved in processing a packet). Unlike many verification tools, Margrave does not expect users to have behavioral requirements or formal security goals on hand. Lightweight summaries such as `SHOW REALIZED` try to provide information that suggests further queries.

Computing Overshadowed Rules through Scripting: Query 3 checks the relationship between two rules on particular packets. A more general question asks which rules *never* apply to *any* packet; we call such rules *superfluous.* The following query computes superfluous rules:

```
EXPLORE true
UNDER InboundACL
INCLUDE
 InboundACL:router-fe0-line9_applies(<req>),
 InboundACL:router-fe0-line10_applies(<req>),
 InboundACL:router-fe0-line12_applies(<req>),
 InboundACL:router-fe0-line14_applies(<req>),
 InboundACL:router-vlan1-line16_applies(<req>)

SHOW UNREALIZED
 InboundACL:router-fe0-line9_applies(<req>),
 InboundACL:router-fe0-line10_applies(<req>),
 InboundACL:router-fe0-line12_applies(<req>),
 InboundACL:router-fe0-line14_applies(<req>),
 InboundACL:router-vlan1-line16_applies(<req>)
```
Query 4

As this computation doesn't care about request contents, the `EXPLORE` clause is simply `true`. The heart of this query lies in the `INCLUDE` clause and the `SHOW UNREALIZED` command: the first asks Margrave to consider all rules; the second asks for listed facts that are never true in any scenario. `UNDER` clauses load policies referenced in `INCLUDE` but not `EXPLORE` clauses.

While the results tell us which rules never apply, they don't indicate which rules overshadow each unused rule. Such information is useful, especially if an overshadowing rule ascribes the opposite decision. Writing queries to determine justification for each superfluous rule, however, is tedious. Margrave's query language is embedded in a host language (Racket [FP10], a descendent of Scheme) through which we can write scripts over query results. In this case, our script uses a Margrave command to obtain lists of rules that yield each of *Permit* and *Deny*, then issues queries to isolate overshadowing rules for each superfluous rule. These are similar to other queries in this section. Scripts could also compute hotspot rules that overshadow a large percentage of other rules.

Change-Impact: Sysadmins edit firewall configurations to provide new services and correct emergent problems. Edits are risky because they can have unexpected consequences such as allowing or restricting traffic that the edit should not have affected. Expecting sysadmins to have formal security requirements against which to test policy edits is unrealistic. In the spirit of lightweight analyses that demand less of users, Margrave computes scenarios illustrating packets whose decision or applicable rule changes in the face of edits.

For example, suppose we add the new boldface rule below to access-list 101 (the line numbers start with 14 to indicate that lines 1–13 are identical to those in Figure 6.1):

```
14  access-list 101 deny tcp
15        host 10.1.1.2 host 192.168.5.10 eq 80
```

If we call the modified filter `InboundACL_new`, the following query asks whether the original and new `InboundACL`s ever disagree on `Permit` decisions:

```
EXPLORE
(InboundACL:Permit(<req>) AND
 NOT InboundACL_new:Permit(<req>)) OR
(InboundACL_new:Permit(<req>) AND
 NOT InboundACL:Permit(<req>)))

IS POSSIBLE?
```
Query 5

Margrave returns false, since the rule at line 9 always overrides the new rule. If instead the new rule were:

```
access-list 101 deny tcp
      host 10.1.1.3 host 192.168.5.10 eq 443
```

Margrave would return true on query 5. The corresponding scenarios show packet headers that the two firewalls treat differently, such as the following:

```
********* SOLUTION FOUND at size = 15
src-addr-in: 10.1.1.3
protocol: prot-tcp
dest-addr-in: 192.168.5.10
src-port-in: port
exit-interface: interface
entry-interface: fe0
dest-port-in: port-80
```
Result

As we might expect, this scenario involves packets from `10.1.1.3`. A subsequent query could confirm that no other hosts are affected.

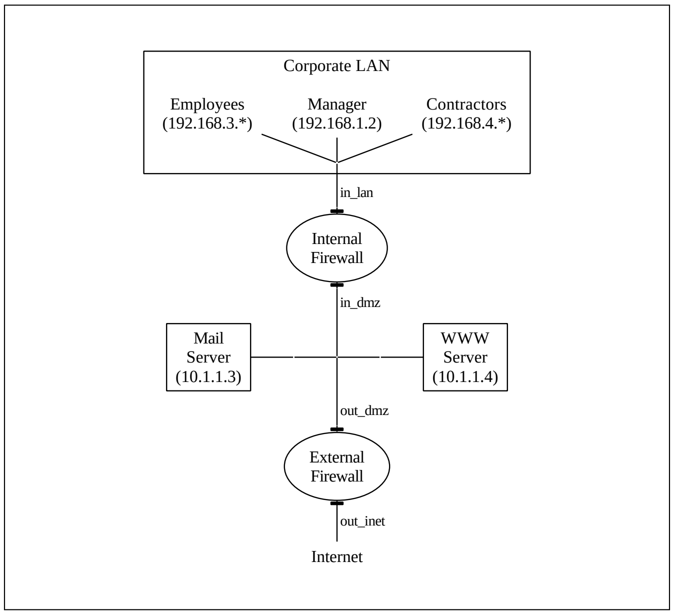


Figure 6.2: A small-business network-topology

Networks of Firewalls: So far, our examples have considered only single firewalls. Margrave also handles networks with multiple firewalls and NAT. Figure 6.2 shows a small network with web server, mail server, and two firewalls to establish a DMZ. The internal firewall performs both NAT and packet-filtering, while the external firewall only filters. The firewall distinguishes machines for employees (`192.168.3.*`), contractors (`192.168.4.*`), and a manager (`192.168.1.2`). This example captures the essence of a real problem posted to a networking help-forum.

```
 1  hostname int
 2  !
 3  interface in_dmz
 4  ip address 10.1.1.1 255.255.255.0
 5  ip nat outside
 6  !
 7  interface in_lan
 8  ip access-group 102 in
 9  ip address 192.168.1.1 255.255.0.0
10  ip nat inside
11  !
12  access-list 102 permit tcp any any eq 80
13  access-list 102 deny any
14  !
15  ip nat inside source list 1 interface
16                in_dmz overload
17  access-list 1 permit 192.168.1.1 0.0.255.255
18  !
19  ip route 0.0.0.0 0.0.0.0 in_dmz
20
```
Internal Firewall

Lines 15–17 in the internal firewall apply NAT to traffic from the corporate LAN.[2] Line 11 in the external firewall blacklists a specific external host (`10.200.200.200`). Despite the explicit rule on lines 19–20 in the external firewall, the manager cannot access the web. We have edited the configurations to show only those lines relevant to the manager and web traffic.

```
 1  hostname ext
 2  !
 3  interface out_dmz
 4  ip access-group 103 in
 5  ip address 10.1.1.2 255.255.255.0
 6  !
 7  interface out_inet
 8  ip access-group 104 in
 9  ip address 10.200.1.1 255.255.0.0
10  !
11  access-list 104 deny 10.200.200.200
12  access-list 104 permit tcp any host 10.1.1.4
13                  eq 80
14  access-list 104 deny any
15  !
16  access-list 103 deny ip any
17                  host 10.200.200.200
18  access-list 103 deny tcp any any eq 23
19  access-list 103 permit tcp host 192.168.1.2
20                  any eq 80
21  access-list 103 deny any
22
```
External Firewall

The following query asks "What rules deny a connection from the manager's PC (line 2) to port 80 (line 10) somewhere outside our network (line 8) other than the blacklisted host (line 9)?"

[2]In this example, we use the 10.200.* private address space to represent the public IP addresses.

```
 1  EXPLORE prot-TCP = protocol AND
 2  192.168.1.2 = fw1-src-addr-in AND
 3  in_lan = fw1-entry-interface AND
 4  out_dmz = fw2-entry-interface AND
 5  hostname-int = fw1 AND
 6  hostname-ext = fw2 AND
 7
 8  fw1-dest-addr-in IN 10.200.0.0/255.255.0.0
 9  NOT 10.200.200.200 = fw1-dest-addr-in AND
10  port-80 = fw1-dest-port-in AND
11
12  internal-result(<reqfull-1>) AND
13
14  (NOT passes-firewall(<reqpol-1>) OR
15   internal-result(<reqfull-2>) AND
16   NOT passes-firewall(<reqpol-2>))
17
18  UNDER InboundACL
19  INCLUDE
20  InboundACL:int-in_lan-line-12_applies
21    (<reqpol-1>),
22  InboundACL:int-in_lan-line-17_applies
23    (<reqpol-1>),
24  InboundACL:ext-out_dmz-line-19_applies
25    (<reqpol-2>),
26  InboundACL:ext-out_dmz-line-21_applies
27    (<reqpol-2>),
28  InboundACL:ext-out_dmz-line-24_applies
29    (<reqpol-2>)
```

Query 6

Lines 12–16 capture both network topology and the effects of NAT. The `internal-result` and `passes-firewall` formulas capture routing in the face of NAT and passing through the complete firewall (including routing, NAT and ACLs) whose hostname appears in the request, respectively; Section 6.2 describes them in detail. The variables sent to the two `passes-firewall` formulas through `<reqpol-1>` and `<reqpol-2>` encode the topology: for example, these shorthands use the same variable name for *dest-addr-out* in the internal firewall and *src-addr-in* in the external firewall. The `fw1-entry-interface` and `fw2-entry-interface` variables (bound to specific interfaces in lines 3–4) appear as the entry interfaces in `<reqpol-1>` and `<reqpol-2>`, respectively.

A `SHOW REALIZED` command over the `INCLUDE` terms (as in query 3) indicates that line 21 of the external firewall configuration is denying the manager's connection. Asking Margrave for a scenario for the query (using the `SHOW ONE` command) reveals that the internal firewall's NAT is changing the packet's source address:

```
...
fw1-src-addr-out=fw2-src-addr_=
  fw2-src-addr-out: 10.1.1.1
fw1-src-addr_=fw1-src-addr-in: 192.168.1.2
```

Result

The external firewall rule (supposedly) allowing the manager to access the Internet (line 19) uses the internal pre-NAT source address; it never matches the post-NAT packet. Naïvely editing the NAT policy, however, can leak privileges to contractors and employees. Change-impact queries are extremely useful for confirming that the manager, and *only* the manager, gain new privileges from an edit. An extended version of this example with multiple fixes and the change-impact queries, is provided in the Margrave distribution.

Summary: These examples illustrate Margrave's ability to reason about both combinations of policies and policies at multiple granularities. The supported query types include asking which packets satisfy a condition (query 1), verification (query 2), rule responsibility (query 3), rule relationships (query 4) and change-impact (query 5). A formal summary of the query language and its semantics is provided with the Margrave distribution.

6.2 Mapping Firewalls to the Theory

There is a sizeable gap between the theory and general tool presented in Chapter 3 and a Cisco IOS configuration, such as the example in Figure 6.1. To represent policies in the theory, we must describe the shapes of requests, the available decisions, what relations can appear in formulas, and how policy rules translate into formulas. Section 6.1 used several relations relevant to firewalls, such as `passed-firewall`. Margrave defines these relations and other details automatically via several mechanisms. We note that the policies and vocabularies shown here will differ slightly in syntax from those in Chapter 3, as this this work was done on a preliminary version of Margrave.

Cisco IOS Configurations as Policies: Figure 6.3 shows part of the result of compiling the IOS configuration in Figure 6.1 to Margrave's intermediate policy language. The fragment captures the IOS rule on line 10. `(Permit hostname, ...)` specifies the decision and states a sequence of variable names corresponding to a request. The `:-` symbol separates the decision from the conditions of the rule. Formula `(prot-tcp protocol)`, for example, captures that TCP is the expected protocol for this rule. Margrave represents constants (such as decisions, IP addresses, and protocols) as elements of singleton unary relations. A scenario that satisfies this rule will map the *protocol* variable to some element of the universe that populates the `prot-tcp` relation. The other conditions of the original rule are captured similarly. The `(RComb FAC)` at the end of the policy tells Margrave to check the policy rules in order (`FAC` stands for "first applicable"). The first line of the policy ascribes the name `InboundACL`.

```
(Policy InboundACL uses IOS-vocab
 (Rules
  ...
  (Router-fe0-line10 =
   (Permit hostname, ...) :-
   (hostname-Router hostname)
   (fe0 entry-interface)
   (IPAddress src-addr-in)
   (prot-tcp protocol)
   (Port src-port-in)
   (192.168.5.10 dest-addr-in)
   (port-80 dest-port-in))
  ...)
 (RComb FAC))
```

Figure 6.3: A Margrave policy for an IOS configuration

Decomposing IOS into policies: Figure 6.4 shows our high-level model of IOS configurations. Firewalls perform packet filtering, packet transformation, and internal routing; the first two may occur at both entry to and exit from the firewall. Specifically, packets pass through the inbound ACL filter, inside NAT transformation, internal routing, outside NAT transformation, and finally the outbound ACL filter on their way through the firewall. The intermediate stages define additional information about a packet (as shown under the stage names): inside NAT may yield new address and port values; internal routing determines the next-hop and exit interface; outside NAT may yield further address and port values.

Internal routing involves five substages, as shown in Figure 6.6. Margrave creates policies (à la Figure 6.3) for each of the five substages. The `-Switching` policies determine whether a destination

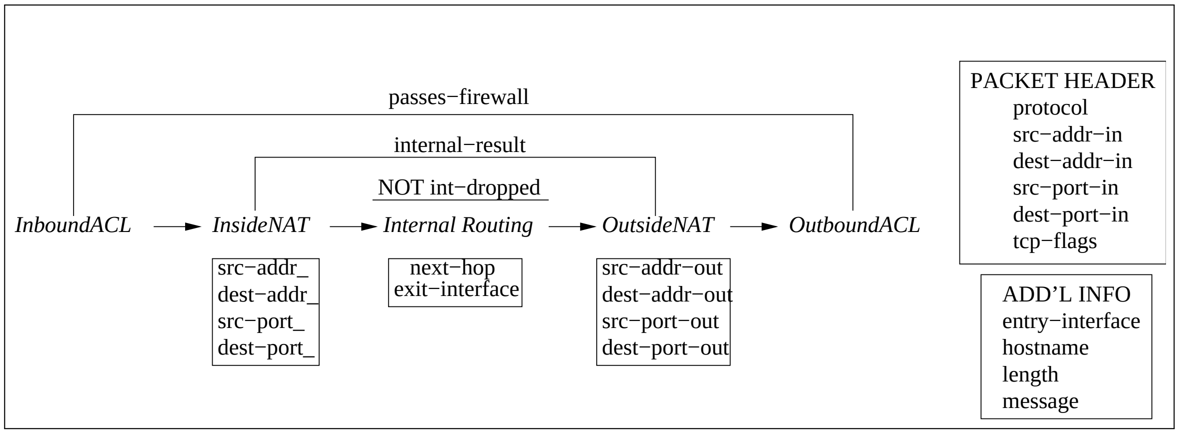


Figure 6.4: Margrave's decomposition of firewall configurations

is directly connected to the firewall; the `-Routing` policies bind the `next-hop` IP address for routing. In addition, Margrave generates four policies called `InboundACL`, `OutboundACL`, `InsideNAT`, and `OutsideNat`. The two `-ACL` policies contain filtering rules for all interfaces.

Requests and Decisions: Margrave automatically defines a relation for each decision rendered by each of the 9 subpolicies (e.g., `InboundACL:Permit` in query 1). Each relation is defined over requests, which contain packet headers, packet attributes, and values generated in the intermediate stages; the boxes in Figure 6.4 collectively list the request contents. As Margrave is not stateful, it cannot update packet headers with data from intermediate stages. The contents of a request reflect the intermediate stages' actions: for example, if the values of `src-addr_` and `src-addr-out` are equal, then `OutsideNAT` did not transform the request's packet. Currently, Margrave shares the same request shape across all 9 subpolicies (even though `InboundACL`, for example, only examines the packet header portion).

Flows between subpolicies: Margrave encodes flows among the 9 subpolicies through three relations (over requests) that capture the subflows marked in Figure 6.4.

- Internal routing either assigns an exit interface and a next-hop to a packet or drops the packet internally. Margrave uses a special exit-interface value to mark dropped packets; the `int-dropped` relation contains requests with this special exit-interface value. Any request that is not in `int-dropped` successfully passes through internal routing.
- Unlike internal routing, NAT never drops packets. At most, it transforms source and destination ports and addresses. Put differently, NAT is a function on packets. `internal-result` captures this function: it contains all requests whose `next-hop`, `exit-interface`, and `OutsideNAT` components are consistent with the packet header and `InsideNAT` components (as if the latter were inputs to a NAT function).
- ACLs permit or deny packets. The relation `passes-firewall` contains requests that the two ACLs permit, are in `internal-result` (i.e., are consistent with NAT), and are not in `int-dropped` (i.e., are not dropped in internal routing).

Our IOS compiler automatically defines each of these relations as a query in terms of the 9 IOS subpolicies (capturing topology as in query 6). Margrave provides a `RENAME` command that saves query results under a user-specific name for use in later queries. Users can name any set of resulting scenarios in this manner.

```
(PolicyVocab IOS-vocab
 (Types
  (Interface : interf-drop
               (interf-real vlan1 fe0))
  (IPAddress :
    192.128.5.0/255.255.255.0
    10.1.1.0/255.255.255.254
    192.168.5.11
    192.168.5.10
    10.1.1.2)
  (Protocol : prot-ICMP prot-TCP prot-UDP)
  (Port: port-25 port-80)
  (Decisions Permit Deny ...)
  ...
  (disjoint-all Protocol)
  (nonempty Port)
  ...
)
```

Figure 6.5: A Margrave vocabulary for an IOS configuration

Cisco IOS Configuration Vocabularies: The 9 subpolicies share ontology about ports and IP addresses. Margrave puts domain-knowledge common to multiple policies in a *vocabulary* specification; the first line of a policy specification references its vocabulary through the `uses` keyword. Figure 6.5 shows a fragment of the vocabulary for IOS policies: it defines datatypes (such as `Protocol`) and their elements (correspondingly, `prot-ICMP`, `prot-TCP`, `prot-UDP`).

Vocabularies also capture domain constraints such as "all protocols are distinct" or "there must be at least one port" (both shown in Figure 6.5). While these constraints may seem odd, they support Margrave's scenario-finding model. Some potential "solutions" are nonsensical, such as one which assigns two distinct numbers to the same physical port. Domain constraints rule out nonsensical scenarios.

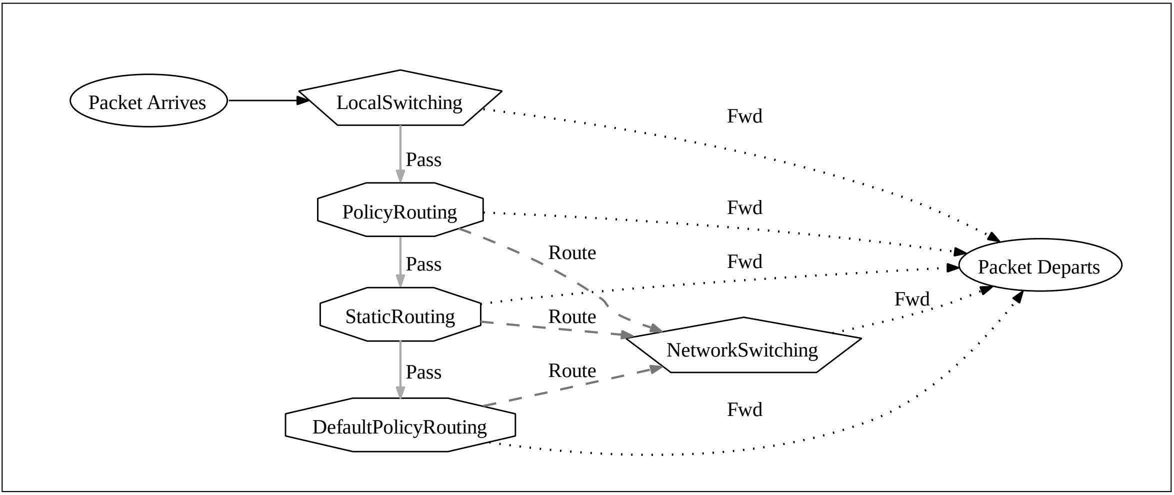


Figure 6.6: Internal flow of packets within a router. Edges are labeled with decisions rendered by the policies at the source of the edge. Routing policies determine the next-hop IP address, while switching policies send traffic to directly to a connected device.

Table 6.2: Run-time impact of `TUPLING` on ACL queries. The first column contains the number of rules in each ACL. The second column lists the number of existentially-quantified variables in the query; we include one 3-variable (non-firewall) query to illustrate the smaller gains on smaller variable counts. The 14-variable ACLs are older firewall examples with smaller request tuples. The "Min Size" column indicates the universe size for the smallest scenario that satisfied the query. Larger minimum sizes have a larger search space.

Rules	# Vars	Min Size	Not Tupled	Tupled
100	3	3	694ms	244ms
1000	14	6	7633ms	1221ms
1000	14	10	17659ms	1219ms
1000	14	14	32116ms	1205ms

6.3 Rewriting Firewall Queries

Under large universe sizes, both the time to compute scenarios and the number of resulting scenarios increase. The latter puts a particular burden on the end-user who has to work through the scenarios. Query language constructs like `SHOW REALIZED` summarize details about the scenarios in an attempt to prevent the exhaustive from becoming exhausting. However, query optimizations that reduce universe sizes have more potential to target the core problem. This section describes an optimization that was developed exclusively for the large request-vectors seen in our Cisco IOS work.

Most firewall queries have the form $\exists\ \overline{req}\ \alpha$, where α typically lacks quantifiers. Requests have 16 or 20 components (as shown in Figure 6.4), depending on whether they reference `internal-result`. Margrave therefore analyzes all-existential queries under a universe size of 16 or 20. However, these queries effectively reference a *single request* with attributes as detailed in α. This suggests that we could rewrite this query with a single quantified variable for a request and additional relations that encode the attributes. For example:

$$\exists\, pt\_in\ \exists\, pt\_out : route(pt\_in, pt\_out)$$

becomes

$$\exists\, pkt : is\_ptIn(pkt, i) \wedge is\_ptOut(pkt, o) \wedge route(i, o)$$

Effectively, these new relations lift attributes from the individual packet fields to the packet as a whole.

Formulas rewritten in this way require a universe size of only 1, for which scenario generation stands to be much faster and to yield fewer solutions. The tradeoff, however, lies in the extra relations that Margrave introduces to lift attributes to the packet level. Additional relations increase the time and yield of scenario computations, so the rewriting is not guaranteed to be a net win.

Table 6.2 presents experimental results on original versus rewritten queries. In practice, we find performance improves when the query is unsatisfiable or the smallest model is large. A user who expects either of these conditions to hold can enable the rewriting through a query-language flag called `TUPLING`. All performance figures reported were computed using `TUPLING`.

6.4 Evaluation

We have two main goals in evaluating Margrave on firewalls. First, we want to confirm that our query language and its results support debugging real firewall configuration-problems; in particular, the scenarios should accurately point to root causes of problems. We assume a user who knows enough firewall basics to ask the questions underlying a debugging process (Margrave does not, for example, pre-emptively try queries to automatically isolate a problem). Second, we want to check that Margrave has reasonable performance on large policies, given that we have traded efficient propositional models for richer first-order ones.

We targeted the first goal by applying Margrave to problems posted to network-configuration help-forums (Sections 6.4.1 and 6.4.2). Specifically, we phrased the poster's reported problem through Margrave queries and sought fixes based on the resulting scenarios. In addition, we used Margrave to check whether solutions suggested in follow-up posts actually fixed the problem without affecting other traffic. The diversity of firewall features that appear in forum posts demanded many compiler extensions, including reflexive access-lists and TCP flags. That we could do this purely at the compiler level attests to the flexibility of Margrave's intermediate policy- and vocabulary-languages (Section 6.2).

We targeted the second goal by applying Margrave to an in-use enterprise firewall-configuration containing several rule sets and over 1000 total rules (Section 6.4.3). Margrave revealed some surprising facts about redundancy in the configuration's behavior. Individual queries uniformly execute in seconds.

Notes on Benchmarking Our figures report Margrave's steady-state performance; they omit JVM warmup time. Policy-load times are measured by loading different copies of the policy to avoid caching bias. All performance tests were run on an Intel Core Duo E7200 at 2.53 Ghz with 2 GB of RAM, running Windows XP Home. Performance times are the mean over at least 100 individual runs; all reported times are ±200ms at the 95-percent confidence level. Memory figures report private (i.e., not including shared) consumption.

6.4.1 Forum Help: NAT and ACLs

"My servers cannot get access into the internet, even though I will be able to access the website, or even FTP... I don't really know what's wrong. Can you please help? Here is my current configuration..."

In our first forum example [azs08], the poster is having trouble connecting to the Internet from his server. He believes that NAT is responsible, and has identified the router as the source of the problem. The configuration included with the post appears in Figure 6.7 (with a slight semantics-preserving modification[3]).

A query (not shown) confirms that the firewall is blocking the connection. Our knowledge of firewalls indicates that packets are rejected either enroute to, or on return from, the webserver. Queries for these two cases are similar; the one checking for response packets is:

```
EXPLORE
NOT src-addr-in
  IN 192.168.2.0/255.255.255.0 AND
FastEthernet0 = entry-interface AND
prot-TCP = protocol AND
port-80 = src-port-in AND
internal-result(<reqfull>) AND
passes-firewall(<reqpol>)

IS POSSIBLE?
```
Query 7

Margrave reports that packets to the webserver are permitted, but responses are dropped. The resulting scenarios all involve source ports 20, 21, 23, and 80 (easily confirmed by re-running the

[3]We replaced named interface references in static NAT statements with actual IP addresses; our compiler does not support the former.

```
1  name-server 207.47.4.2
2  name-server 207.47.2.178
3  !
4  interface FastEthernet0
5  ip address 209.172.108.16 255.255.255.224
6  ip access-group 102 in
7  ip nat outside
8  speed auto
9  full-duplex
10 !
11 interface Vlan1
12 ip address 192.168.2.1 255.255.255.0
13 ip nat inside
14 !
15 ip route 0.0.0.0 0.0.0.0 209.172.108.1
16 !
17 ip nat pool localnet 209.172.108.16 prefix-length 24
18 ip nat inside source list 1 pool localnet overload
19 ip nat inside source list 1 interface FastEthernet0
20 ip nat inside source static tcp 192.168.2.6 80 209.172.108.16 80
21 ip nat inside source static tcp 192.168.2.6 21 209.172.108.16 21
22 ip nat inside source static tcp 192.168.2.6 3389 209.172.108.16 3389
23 !
24 access-list 1 permit 192.168.2.0 0.0.0.255
25 access-list 102 permit tcp any host 209.172.108.16 eq 80
26 access-list 102 permit tcp any host 209.172.108.16 eq 21
27 access-list 102 permit tcp any host 209.172.108.16 eq 20
28 access-list 102 permit tcp any host 209.172.108.16 eq 23
29 access-list 102 deny tcp any host 209.172.108.16
```

Figure 6.7: The original configuration for the forum post for Section 6.4.1

query with a `SHOW REALIZED` command asking for only the port numbers). This is meaningful to a sysadmin: an outgoing web request is always made from an *ephemeral* port, which is never less than 1024. This points to the problem: the router is rejecting all returning packets. ACL 102 (Figure 6.7, lines 25–29) ensures that the server sees only incoming HTTP, FTP, and TELNET traffic, at the expense of rejecting the return traffic for any connections that the server initiates.

Enabling the server to access other webservers involves allowing packets *coming from* the proper destination ports. Methods for achieving this include:

1. Permit TCP traffic *from* port 80, via the edit:

```
28 access-list 102 permit tcp
29             any host 209.172.108.16 eq 23
30 access-list 102 permit tcp any eq 80 any
31 access-list 102 deny tcp
32             any host 209.172.108.16
```

2. Allow packets whose *ack* flags are set via the *established* keyword (or, in more recent versions, the *match-all +ack* option). This suggestion guards against spoofing a packet's source port field and allows servers to listen on unusual ports.

3. Use stateful monitoring of the TCP protocol via reflexive access-lists or the *inspect* command. This guards against spoofing of the TCP ACK flag.

Follow-up posts in the forum suggested options 1 and 3. Margrave can capture the first two options and the reflexive access-list approach in the third (it does not currently support *inspect* commands). For each of these, we can perform verification queries to establish that the `InboundACL` no longer blocks return packets, and we can determine the extent of the change through a change-impact query. To apply reflexive ACLs here, we first apply an ACL to the inside interface:

```
interface Vlan1
ip address 192.168.2.1 255.255.255.0
ip access-group frominside in
ip nat inside
```

and then create the reflexive ACL:

```
ip access-list extended frominside
  permit tcp any any reflect returntcp
```

leaving the original ACL for the external interface intact save for an invocation to *evaluate* the dynamic list being built by outgoing packets:

```
ip access-list extended fromoutside
  permit tcp any host 209.172.108.16 eq 80
  permit tcp any host 209.172.108.16 eq 21
  permit tcp any host 209.172.108.16 eq 20
  permit tcp any host 209.172.108.16 eq 23
  evaluate returntcp
  deny tcp any host 209.172.108.16
```

Reflexive ACLs allow return traffic from hosts to which prior packets were permitted. Margrave encodes prior traffic through a series of `connection-` relations over requests. Intuitively, a request is in a `connection-` relation only if the same request with the source- and destination-details reversed would pass through the firewall. Although the connection state is dynamic in practice, its stateless definition enables Margrave to handle it naturally through first-order relations.

Performance: Loading each version of the configuration took between 3 and 4 seconds. The final change-impact query took under 1 second. After loading, running the full suite of queries (including those not shown) required 2751ms. The memory footprint of the Java engine (including all component subpolicies) was 50 MB (19 MB JVM heap, 20 MB JVM non-heap).

6.4.2 Forum Help: Routing

"there should be a way to let the network `10.232.104.0/22` access the internet, kindly advise a solution for this..."

In our second example [Oel09], the poster is trying to create two logical networks: one "primary" (consisting of `10.232.0.0/22` and `10.232.100.0/22`) and one "secondary" (consisting of `10.232.4.0/22` and `10.232.104.0/22`). These logical networks are connected through a pair of routers (TAS and BAZ) which share a serial interface (Figure 6.8). Neither logical network should have access to the other, but both networks should have access to the Internet—the primary via `10.232.0.15` and the secondary via `10.232.4.10`.

The poster reports two problems: first, the two components of the primary network, `10.232.0.0/22` and `10.232.100.0/22`, cannot communicate with each other; second, the network `10.232.104.0/22` cannot access the Internet. The poster suspects errors in the TAS router configuration, which is shown in Figure 6.9.

We start with the first problem. The following query confirms that network `10.232.0.0/22` cannot reach `10.232.100.0/22` via the serial link. The `hostname` formulas introduce names for each individual router based on the hostname specification in the IOS configuration; these names appear in the `tasvector-` and `bazvector-` requests. (The `-full-` requests extend the corresponding `-pol-` requests with additional variables needed for `internal-routing`.

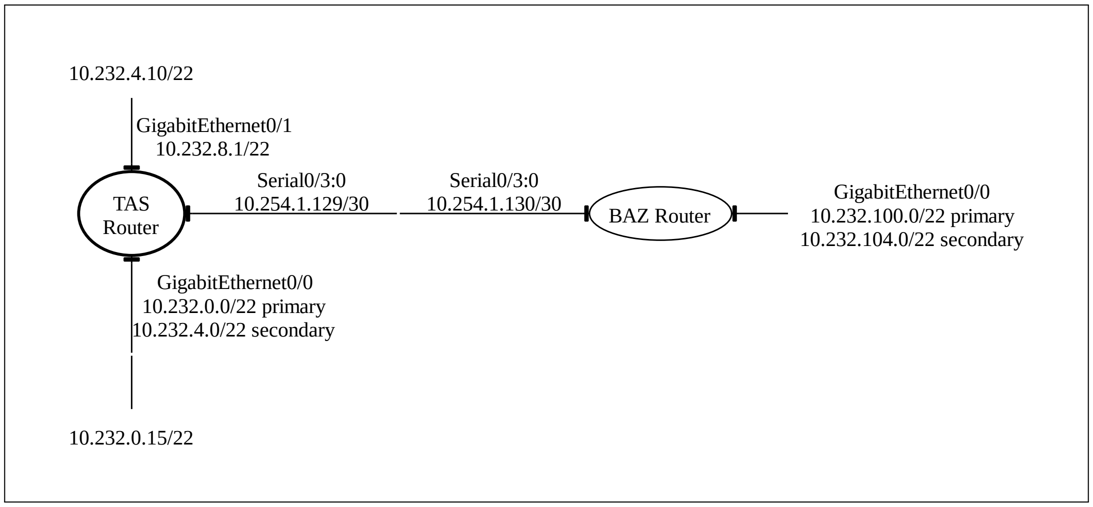


Figure 6.8: Structure of the network for the forum post for Section 6.4.2

```
1  EXPLORE hostname-tas = tas AND
2  hostname-baz = baz AND
3
4  internal-result(<tasvectorfull-fromtas>) AND
5  internal-result(<bazvectorfull-fromtas>) AND
6  passes-firewall(<tasvectorpol-fromtas>) AND
7  passes-firewall(<bazvectorpol-fromtas>) AND
8
9  GigabitEthernet0/0 = tas-entry-interface AND
10 tas-src-addr-in IN
11   10.232.0.0/255.255.252.0 AND
12 tas-dest-addr-in IN 10.232.100.0/255.255.252.0
13 AND "Serial0/3/0:0" = tas-exit-interface AND
14
15 "Serial0/3/0:0" = baz-entry-interface AND
16 GigabitEthernet0/0 = baz-exit-interface
17
18 IS POSSIBLE?
```
Query 8

Margrave returns false, which means that no packets from `10.232.0.0/22` reach `10.232.100.0/22` along this network topology.

By the topology in Figure 6.8, packets reach the TAS router first. We check whether packets pass through TAS by manually restricting query 8 to TAS (by removing lines 2, 5, 7, 14, and 15); Margrave still returns false. Firewall knowledge suggests three possible problems with the TAS configuration: (1) internal routing could be sending the packets to an incorrect interface, (2) internal routing could be dropping the packets, or (3) the ACLs could be filtering out the packets. Margrave's formulas for reasoning about internal firewall behavior help eliminate these cases: by negating `passed-firewall` on line 6, we determine that the packet does pass through the firewall, so the problem lies in the interface or next-hop assigned during routing. This example highlights the utility of not only having access to these formulas, but also having the ability to negate (or otherwise manipulate) them as any other subformula in a query.

To determine which interfaces the packets are sent on, we relax the query once again to remove the remaining reference to `Serial0/3/0:0` (on line 12) and execute the following `SHOW REALIZED` command:

```
hostname tas
!
interface GigabitEthernet0/0
description $ETH-LAN$$ETH-SW-LAUNCH$$INTF-INFO-GE 0/0$
ip address 10.232.4.1 255.255.252.0 secondary
ip address 10.232.0.1 255.255.252.0
ip access-group 101 in
ip policy route-map internet
duplex auto
speed auto
!
interface GigabitEthernet0/1
ip address 10.232.8.1 255.255.252.0
duplex auto
speed auto
!
interface Serial0/3/0:0
ip address 10.254.1.129 255.255.255.252
ip access-group 102 out
encapsulation ppp
!
ip route 10.232.100.0 255.255.252.0 10.254.1.130
ip route 10.232.104.0 255.255.252.0 10.254.1.130
!
access-list 102 deny ip 10.232.0.0 0.0.3.255 10.232.104.0 0.0.3.255
access-list 102 deny ip 10.232.4.0 0.0.3.255 10.232.100.0 0.0.3.255
access-list 102 permit ip any any
access-list 101 deny ip 10.232.0.0 0.0.3.255 10.232.4.0 0.0.3.255
access-list 101 deny ip 10.232.4.0 0.0.3.255 10.232.0.0 0.0.3.255
access-list 101 permit ip any any
!
access-list 10 permit 10.232.0.0 0.0.3.255
access-list 10 permit 10.232.100.0 0.0.3.255
access-list 20 permit 10.232.4.0 0.0.3.255
access-list 20 permit 10.232.104.0 0.0.3.255
!
route-map internet permit 10
match ip address 10
set ip next-hop 10.232.0.15
!
route-map internet permit 20
match ip address 20
set ip next-hop 10.232.4.10
```

Figure 6.9: Original Configuration (2)

```
SHOW REALIZED
     GigabitEthernet0/0 = exit-interface,
     "Serial0/3/0:0" = exit-interface,
     GigabitEthernet0/1 = exit-interface
```
Query 9

The output contains only one interface name:

```
{ GigabitEthernet0/0[exit-interface] }
```
Result

According to the topology diagram, packets from `10.232.0.0/22` to `10.232.100.0/22` should be using exit interface `Serial0/3/0:0`; the results, instead, indicate exit interface `GigabitEthernet0/0`. Firewall experience suggests that the router is either switching the correct next-hop address (`10.254.1.130`) to the wrong exit interface, or using the wrong next-hop address. The next query produces the next-hop address:

```
1  EXPLORE hostname-tas = tas AND
2  internal-result(<tasvectorfull-fromtas>) AND
3  passes-firewall(<tasvectorpol-fromtas>) AND
4  GigabitEthernet0/0 = tas-entry-interface AND
5  tas-src-addr-in IN
6    10.232.0.0/255.255.252.0 AND
7  tas-dest-addr-in IN 10.232.100.0/255.255.252.0
8
9  INCLUDE
10 10.232.0.15 = tas-next-hop,
11 10.232.4.10 = tas-next-hop,
12 tas-next-hop IN 10.254.1.128/255.255.255.252,
13 tas-next-hop IN 10.232.8.0/255.255.252.0
14
15 SHOW REALIZED
16 10.232.0.15 = tas-next-hop,
17 10.232.4.10 = tas-next-hop,
18 tas-next-hop IN 10.232.8.0/255.255.252.0,
19 tas-next-hop IN 10.254.1.128/255.255.255.252
```
Query 10

```
{ 10.232.0.15[tas-next-hop] }
```
Result

The next-hop address is clearly wrong for the given destination address. To determine the extent of the problem, we'd like to know whether *all* packets from the given source address are similarly misdirected. That question is too strong, however, as `LocalSwitching` may (rightfully) handle some packets. To ask Margrave for next-hops targeted by some source packet that `LocalSwitching` ignores, we replace line 7 in query 10 with:

```
NOT LocalSwitching:Forward(<routingpol-tas>)
```
Query 11

This once again highlights the value of exposing `LocalSwitching` as a separate relation. The revised query yields the same next-hop, indicating that all non-local packets are routing to `10.232.0.15`, despite the local routing policies. A simple change fixes the problem: insert the keyword **default** into the routing policy:

```
route-map internet permit 10
match ip address 10
set ip default next-hop 10.232.0.15
```

This change ensures that packets are routed to the Internet only as a last resort (i.e., when static destination-based routing fails). Running the original queries against the new specification confirms that the primary subnets now have connectivity to each other. Another query checks that this change does not suddenly enable the primary sub-network `10.232.0.0/22` to reach the secondary sub-network `10.232.4.0/22`.

Now we turn to the poster's second problem: the secondary network `10.232.4.0/22` still cannot access the Internet. As before, we confirm this then compute the next-hop and exit interface that TAS assigns to traffic from the secondary network with an outside destination. The following query (with `SHOW REALIZED` over interfaces and potential next-hops) achieves this:

```
EXPLORE
tas = hostname-tas AND

internal-result2(<tasvectorfull-fromtas>) AND
firewall-passed2(<tasvectorpol-fromtas>) AND

GigabitEthernet0/0 = tas-entry-interface AND
tas-src-addr-in IN
  10.232.4.0/255.255.252.0 AND

NOT tas-dest-addr-in IN
  10.232.4.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.104.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.0.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.100.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.254.1.128/255.255.255.252 AND
NOT tas-dest-addr-in IN
  192.168.1.0/255.255.255.0 AND
NOT tas-dest-addr-in IN
  10.232.8.0/255.255.252.0
```

Query 12

```
{   gigabitethernet0/0[tas-exit-interface],
     10.232.4.10[tas-next-hop] }
```

Result

The next-hop for the secondary network's Internet gateway is as expected, but the exit-interface is unexpectedly `GigabitEthernet0/0` (instead of `GigabitEthernet0/1`). In light of this scenario, the network diagram reveals a fundamental problem: the gateway `10.232.4.10` should be "on" the same network as the `GigabitEthernet0/1` interface (address `10.232.8.1/22`); otherwise `LocalSwitching` will send the packet to the wrong exit interface.

This problem can be resolved by changing the address of either the `GigabitEthernet0/1` interface or the next-hop router (`10.232.4.10`). We chose the latter, selecting an arbitrary unused address in the `10.232.8.0/22` network:

```
route-map internet permit 20
match ip address 20
set ip default next-hop 10.232.8.10
```

Re-running the queries in this new configuration confirms that both goals are now satisfied.

Performance: Loading each version of the configuration took between 3 and 4 seconds. Query 12 took 351 ms. After loading, running the full suite of queries (including those not shown) finished in 8725ms. The memory footprint of the Java engine (including all component subpolicies) was 74 MB (49 MB JVM heap, 21 MB JVM non-heap).

6.4.3 Enterprise Firewall Configuration

Our largest test case to date is an in-use enterprise iptables configuration. In order to stress-test our IOS compiler, we manually converted this configuration to IOS. The resulting configuration contains ACLs for 6 interfaces with a total of 1108 `InboundACL` rules (not counting routing subpolicies). The

Table 6.3: Run-time performance of various queries on the enterprise ACLs. For the change-impact query, we switched the decision from *deny* to *permit* on one non-superfluous rule. The overshadowing-rules computation asked only for overshadows with the opposite decision.

Query	**Time (ms)**
Permit pkt from addr X on interface Y?	1587
Previous with rule responsibility	23317
Change-impact after 1 decision edit	3167
Previous with rule responsibility	24039
Detect all superfluous rules	22578
List overshadows per rule in previous	72178

routing component of this firewall was fairly simple; we therefore focus our performance evaluation on `InboundACL`.

From a performance perspective, we have illustrated three fundamentally different types of queries: (1) computing over a single policy or network with just the default relations (which-packets and verification queries), (2) computing over a single policy or network while including additional relations (rule-responsibility and rule-relationship queries), and (3) computing over multiple, independent policies or networks (change-impact queries). The third type introduces more variables than the first two (to represent requests through multiple firewalls); it also introduces additional relations to capture the policies of multiple firewalls. The second type has the same number of variables, but more relations, than the first type. We therefore expect the best performance on the first type, even under `TUPLING`.

Table 6.3 reports run-time performance on each type of query over the enterprise firewall-configuration. Loading the policy's `InboundACL` component required 10694ms and consumed 51 MB of memory. Of that, 40 MB was JVM heap and 7 MB was JVM non-heap.

Section 6.1 described how we compute superfluous rules through scripting. For this example, these queries yielded surprising results: 900 of the 1108 rules in `InboundACL` were superfluous. Even more, 270 of the superfluous rules were (at least partially) overshadowed by a rule with a different decision. The sysadmins who provided the configuration found these figures shocking and subsequently expressed interest in Margrave.

6.5 Related Work

Studies of firewall-configuration errors point to the need for analysis tools. Oppenheimer, *et al.* [OGP] survey failures in three Internet services over a period of several months. For two of these services, operator error—predominately during configuration edits—was the leading cause of failure. Furthermore, conventional testing fails to detect many configuration problems. Wool [Woo04] studies the prevalence of 12 common firewall-configuration errors. Larger rule-sets yield a much higher ratio of errors to rules than smaller ones; Wool concludes that complex rule sets are too difficult for a human administrator to manage unaided.

In a seminal paper on firewall analysis, Guttman [Gut97] gives algorithms for converting a network-wide connectivity policy into a set of router-level logical specifications (a vendor-independent "filtering posture") that implements the policy. The paper also describes an algorithm for checking existing filtering postures to see if they conform to the global policy. Both algorithms are based on graph traversal, which is a natural approach given that connectivity is inherently about host reachability. Margrave does not support the automated generation of router configurations, and its SAT-based analyses are not geared toward reachability. Instead, it provides support for richer analyses, consideration of mutation of packets between routers (e.g. NAT), and the ability to find the cause of unexpected network behavior at the rule level.

Mayer, Wool and Ziskind [MWZ00, MWZ05] and Wool [Woo01] describe a tool called Fang that has evolved into a commercial product called the AlgoSec Firewall Analyzer [Alg]. AlgoSec supports most of the same analyses as Margrave, covering NAT and routing, but it does not support first-order queries or integration with a programming language. AlgoSec captures packets that satisfy queries through sub-queries, which are a form of abstract scenarios.

Marmorstein and Kearns' [MK05b, MK05a] ITVal tool uses Multi-way Decision Diagrams (MDDs) to execute SQL-like queries on firewall policies. ITVal supports NAT, routing, and chains of firewall policies. Later work [MK06] supports a useful query-free analysis: it generates an equivalence relation that relates two hosts if identical packets (modulo source address) from both are treated identically by the firewall. This can detect policy anomalies and help administrators understand their policies. Additional debugging aids in later work includes tracing decisions to rules and showing examples similar to scenarios. Margrave is richer in its support for change-impact and first-order queries.

Al-Shaer *et al.*'s ConfigChecker [ASH03, ASH04] is a BDD-based tool that analyses networks of firewalls using CTL (temporal logic) queries. Rules responsible for decisions can be isolated manually through queries over sample packets. For performance reasons, the tool operates at the level of policies, rather than individual rules (other of the group's papers do consider rule-level reasoning); Margrave, in contrast, handles both levels.

Bhatt *et al.*'s Vantage tool [BBOR08, BOR08, BR08] supports change-impact on rule-sets and other user-defined queries over combinations of ACLs and routing; it does not support NAT. Some of their evaluations [BOR08] exploit change-impact to isolate configuration errors. This work also supports generating ACLs from specifications, which is not common in firewall-analysis tools.

Liu and Gouda [LG08, LG09] introduce Firewall Decision Diagrams (FDDs) to answer SQL-like queries about firewall policies. FDDs are an efficient variant of BDDs for the firewall packet-filtering domain. Extensions of this work by Khakpour and Liu [KL10] present algorithms for many of the firewall analyses we discuss, including user-defined queries, rule responsibility, and change-impact, generally in light of NAT and routing. A downloadable tool is under development.

Yuan, *et al.*'s Fireman tool [YMS$^+$06] analyzes large networks of firewall ACLs using Binary Decision Diagrams (BDDs). Fireman supports a fixed set of analyses, including whitelist and blacklist violations and computing conflicting, redundant, or correlated rules between different ACLs. Fireman examines all paths between firewalls at once, but does not consider NAT or internal routing. Margrave's combination of user-defined queries and support for NAT and routing makes it much richer. Oliveira, *et al.* [OLK09] extend Fireman with NAT and routing tables. Their tool, Prometheus, can also determine which ACL rules are responsible for a misconfiguration. It does not handle change-impact across firewalls, though it does determine when different paths through the same firewall render different decisions for the same packet. In certain cases, Prometheus suggests corrections to rule sets that guarantee desired behaviors. Margrave's query language is richer.

Verma and Prakash's FACE tool [VP05] aids both configuration of distributed firewalls and analyzing existing distributed firewalls expressed in iptables. It supports user-defined queries, as well as a form of change-impact over multiple firewalls. Its depth-first-search approach to propagating queries through a network resembles Mayer, Ziskind, and Wool's work. It does not handle routing or NAT. The tool is no longer available.

Gupta, LeFevre and Prakash [GLP09] give a framework for the analysis of heterogeneous policies that is similar to ours. While both works provide a general policy-analysis language inspired by SQL, there are distinct differences. Their tool, SPAN, does not allow queries to directly reference rule applicability and the work does not discuss request-transformations such as NAT. However, SPAN provides tabular output that can potentially be more concise than Margrave's scenario-based output. SPAN is currently under development.

Lee, Wong, and Kim's NetPiler tool [LWK08, LWK09] analyzes the flow graph of routing policies. It can be used to both simplify and detect potential errors in a network's routing configurations. The authors have primarily applied NetPiler to BGP configurations, which address the propagation of routes rather than the passage of packets. However, their methods could also be applied to firewall policies. Margrave does not currently support BGP, though its core engine is general enough to support them.

Jeffrey and Samak [JS09] present a formal model and algorithms for analyzing rule-reachability and cyclicity in iptables firewalls. This work does not address NAT or more general queries about firewall behavior.

Eronen and Zitting [EZ01] perform policy analysis on Cisco router ACLs using a Prolog-based Constraint Logic Programming framework. Users are allowed to define their own custom predicates (as in Prolog), which enables analysis to incorporate expert knowledge. The Prolog queries are also first-order. This work is similar to ours in spirit, but is limited to ACLs and does not support NAT or routing information.

Youssef *et al.* [BYBJ09] verify firewall configurations against security goals, checking both for configurations that violate goals and goals that configurations fail to cover. The work does not handle NAT or routing.

6.6 Perspective on Applying Margrave

We introduced Margrave as a general-purpose policy analyzer in Chapter 3. Its most distinctive features lie in and arise from embracing scenario finding over first-order models. First-order languages provide the expressive power of quantifiers and relations for capturing both policies and queries. Expressive power generally induces performance cost. By automatically computing universe bounds (Chapter 4) for key queries, however, Margrave gets the best of both worlds: first-order logic's expressiveness with propositional logic's efficient analysis. Effectively, Margrave distinguishes between propositional *models* and propositional *implementations*. Most logic-based *firewall*-analysis tools conflate these choices.

First-order modeling lets Margrave uniformly capture information about policies at various levels of granularity. We have illustrated relations capturing policy decisions, individual rule behavior, and the effects of NAT and internal routing. The real power of our first-order modeling, however, lies in building new relations from existing ones. Each of the relations capturing behavior internal to a firewall (`passes-firewall`, `internal-routing`, and `int-dropped`) is defined within Margrave's query language and exported to the user through standard Margrave commands. While our firewall compilers provide these three automatically, users can add their own relations in a similar manner. Technically, Margrave allows users to define their own named views (in a database sense) on collections of policies. Thus, Margrave embraces policy-analysis in the semantic spirit of databases, rather than just the syntactic level of SQL-style queries.

Useful views build on fine-grained atomic information about policies. Margrave's unique decomposition of IOS configurations into subpolicies for nine distinct firewall functions provides that foundation. Our pre-defined firewall views would have been prohibitively hard to write without a clean way to refer to components of firewall functionality. Margrave's intermediate languages for policies and vocabularies, in turn, were instrumental in developing the subpolicies. Both languages use general relational terms, rather than domain-specific ones. Vocabularies allow authors to specify decisions beyond those typically associated with policies (such as *Permit* and *Deny*). Our IOS compiler defines separate decisions for the different types of flows out of internal routing, such as whether packets are forwarded internally or translated to another interface. The routing views are defined in terms of formulas capturing these decisions. The policy language defines the formulas through rules that yield each decision (our rule language is effectively stratified Datalog). Had we defined Margrave as a firewall-specific analyzer, rather than a general-purpose one, we likely would have hardwired domain-specific concepts that did not inherently support this decomposition.

User-defined decisions and views support extending Margrave from within. Integrating Margrave into a programming language supports external extension via scripting over the results of commands. Margrave produces scenarios as structured (XML) objects that can be traversed and used to build further queries. `SHOW REALIZED` produces lists of results over which programs (such as superfluous rule detection in Section 6.1) can iterate to generate additional queries.

Chapter 7

Analysis of Software-Defined Networks

The primary function of a network is to route traffic from point to point. To do so, each switch on the network must know how to route arriving packets. There is a conceptual distinction between a network's *control plane*, which settles on such a routing policy for each switch, and its *data plane*, which actually performs the routing. In a traditional network, switches implement both planes themselves, with the control plane distributed between the switches via standardized protocols. These switches must be *configured*, usually in a fairly inexpressive language such as Cisco IOS (Chapter 6). They do not tend to be *programmed* in the full sense of the word.

In a software-defined network (SDN), switches receive their routing orders from programs running on a centralized controller. This strategy allows developers and operators to truly program their network, rather than being restricted by proprietary, black-box abstractions and limited configuration languages. Many network programs are written using libraries in widely-used production languages like Java and Python. Other languages, tailor-made for programming SDNs, are beginning to emerge as well. Such languages must be sufficiently low-level and expose enough networking primitives to enable network programming.

In the rush to create languages for a new domain, expressive power often wins. This is natural: the first task of a language is to express functionality. Thus, most software-defined network controller programs are currently written using libraries in general-purpose (henceforth, "full") programming languages like Java and Python. Even tailor-made SDN languages (e.g., Frenetic [FHF$^+$11] and NetCore [MFHW12]) are embedded in full programming languages to support dynamic policies and state.

Because of the power of these languages, analysis of controller programs is non-trivial. In languages with concurrency and state it can be extremely difficult to form useful static summaries of program behavior. Add features like `eval` and dynamic loading, and it can be hard or impossible to even determine what the program's full code even *is*. As a result, analyses are either dynamic or use unsound techniques such as symbolic evaluation. Any form of reliable static analysis is heavily compromised and tools can usually only offer a "best effort" outcome. Approaches like directly analyzing static forwarding tables [ASAH10] cannot avail of the rich semantic knowledge contained in the original controller program. This is sufficient in some contexts but not when reliability is a must. The Margrave tool, discussed in Chapters 3 and 6, provides sound, complete, and rich analyses for configuration languages. Thus, our goal is to sustain the Margrave perspective in the far more complex world of SDN programming.

We contend that languages for network programming can be effectively designed with respect to *both* expressive power and analyzability, resulting in a powerful new idiom in which to program networks. As a first step on this hybrid path, we have created FlowLog[1], a declarative language for programming SDN controllers. We show that FlowLog is expressive enough to be used for many real-world controller programs. It is also a finite-state language, and thus amenable to many types of analysis, which we discuss. We compile arbitrary FlowLog programs for automated bug-finding via standard model-checking techniques, showing that useful and efficient static analyses are possible with minimal programmer effort.

7.1 FlowLog by Example

We introduce FlowLog through a pair of examples.

7.1.1 MAC Learning

We begin with a controller that learns which physical ports have connectivity to which layer-2 (*MAC*) addresses. It also provides forwarding instructions to the switches that take this learning into account.

```
   +learned(pkt, sw, pt, mac) ←
     pkt.locSw == sw,
     pkt.dlSrc == mac,
     pkt.locPt == pt;
 5 emit(pkt, newp) ←
     learned(pkt.locSw,newp.locPt,pkt.dlDst),
     pkt.locSw == newp.locSw,
     pkt.header == newp.header;
   emit(pkt, newp) ←
10   not learned(pkt.locSw, any, pkt.dlDst),
     pkt.locSw == newp.locSw,
     newp.locPt != pkt.locPt,
     pkt.header == newp.header;
   -learned(pkt, sw, pt, mac) ←
15   pkt.locSw == sw,
     pkt.dlSrc == mac,
     pkt.locPt != pt;
```

Listing 7.1: MAC Learning in FlowLog

As the name suggests, FlowLog is inspired by Datalog [AHV95]. FlowLog programs consist of a set of *rules*, each of which has a *head* and a *body* separated by the implication ←. Each rule describes one of two things: a modification of the controller's internal state (e.g., the rules with **+learned** and **−learned** in their head) or a packet-handling action (the rules with **emit** in the head). When the conditions in the body are met, the action specified in the head is performed.

The first rule tells the controller when to learn a port-to-address mapping. The rule's head (line 1) instructs the controller that when it receives a packet `pkt`, it should add any tuples `<sw, pt, mac>` that satisfy the rule body to its `learned` relation. The body simply ensures that the tuple's contents represent the packet's location switch (line 2), source MAC address (line 3), and arrival port (line 4).

The second rule, beginning on line 5, handles forwarding when the controller has learned an appropriate port. It informs the controller that when a packet `pkt` is received, all new packets `newp` that match the rule body should be sent. That is, the new packet must be sent out the port

[1]This chapter describes joint work with Arjun Guha, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi.

dictated by the `learned` relation (line 6), on the same switch as received it (line 7). All other attributes of the packet are left unmodified (line 8). This last stipulation is necessary because FlowLog will cause *all* packets that match the conditions in the rule body to be produced. This behavior is extremely useful for flooding packets on multiple ports (e.g., in the next rule).

The third rule, beginning on line 9, handles the other forwarding case: when the controller has not yet learned an appropriate port to send a packet on. Barring two differences, it is identical to the second rule: the negative use of the `learned` relation on line 10 enforces that, for the rule to fire, *no* appropriate port has been learned. Line 12 adds a condition on the output port that ensures that the switch will not backflow traffic.

Finally, the rule beginning on line 14 allows the controller to account for *mobility* of hosts by removing outdated port-to-address mappings.

7.1.2 ARP Cache

Our second example involves the controller acting as a cache for address-resolution protocol responses. ARP allows hosts to discover which datalink-layer (MAC) addresses are currently bound to a specific network-layer (IP) address. The program must, therefore, do several things: allow the dissemination of ARP requests across the network, spy on the replies to those requests in order to update its internal state, intercept requests for which the reply is already known, and spawn reply packets for known requests. Excluding the un-learning of outdated ARP assignments, the entire program is given below:

```
+learned(pkt, ip, mac) ←
  pkt.dlTyp == 0x0806,
  pkt.nwProto == 2,
  not learned(pkt.nwSrc, any),
  ip == pkt.nwSrc,
  mac == pkt.dlSrc;
emit(pkt, newpkt) ←
  pkt.dlTyp == 0x0806,
  (pkt.nwProto == 2 ||
    (pkt.nwProto == 1 &&
     not learned(pkt.nwSrc, any))),
  newpkt.locSw == pkt.locSw,
  newpkt.locPt != pkt.locPt,
  newpkt.header == pkt.header;
emit(pkt, newpkt) ←
  pkt.dlTyp == 0x0806,
  pkt.nwProto == 1,
  newpkt.dlTyp == 0x0806,
  newpkt.nwProto == 2,
  learned(pkt.nwDst, newpkt.dlSrc),
  newpkt.loc == pkt.loc,
  newpkt.nwDst == pkt.nwSrc,
  newpkt.dlDst == pkt.dlSrc,
  newpkt.nwSrc == pkt.nwDst,
  newpkt.otherwise == pkt.otherwise;
```

Listing 7.2: ARP Cache in FlowLog

In this program, `pkt.dlTyp == 0x0806` checks that the packet is an ARP packet, for which `pkt.nwProto == 1` denotes a request and `pkt.nwProto == 2` a reply.

The first rule updates the controller's state when the network sees an ARP reply for a previously-unknown IP address. The negated condition on line 4 encodes that the IP is indeed not in the current state.

The first **emit** rule, beginning on line 7, allows the propagation of all ARP replies as well as ARP requests for unknown IP addresses. Line 9 floods reply packets, and line 10 stipulates that the IP address must unknown to apply to request packets. Line 13 prevents backflow, and line 14 explicitly preserves the packet header.

The second **emit** rule (starting line 15) spawns reply packets for IP addresses that the controller knows. The generated packet `newpkt` is constrained to be an ARP reply (line 18), while the originating packet must be an ARP request (line 16). Its source MAC address is obtained from the controller's state (line 20). The reply is sent from the same switch and port that observed the request (line 21) but with the packet's destination and source addresses reversed (lines 22–24). All other header fields are preserved in the new packet (line 25). As before, omitting this restriction would cause FlowLog to produce packets to cover all possibilities.

7.2 FlowLog Under the Hood

These examples show that FlowLog offers a concise means for describing how a controller ought to react to packets. It can govern how those packets are forwarded, cause entirely new packets to be produced and sent, and modify the controller's internal state, by both adding and removing information.

FlowLog syntax is inspired by Datalog [AHV95] with negation, to which we have added a notion of packet fields and slight syntactic sugar. Furthermore, we require that there be no cyclic dependencies between relation names across programs, i.e., FlowLog is *non-recursive*. FlowLog's concrete syntax as used in Section 7.1 is currently provisional, and thus we do not give a grammar for it here. However, FlowLog's core language (i.e. after desugaring) is simply non-recursive Datalog with negation — the same formalism that Margrave uses for its policies. The heart of FlowLog lies in its semantics.

FlowLog Semantics FlowLog programs have a relational notion of state, which the program can both read and update. A relational state is simply a collection of concrete tables (equivalent to a relational database). For example, in our ARP-cache implementation, the controller's knowledge of which IP addresses are mapped to which ethernet addresses is contained in the `learned` relation.

Let P be a FlowLog program, and let $sig(P)$ denote the state relations used in P. If rules exist to define them, it is natural to consider also the following generated relations, which have special meaning in FlowLog:

1. **emit**, a special binary relation;
2. $\texttt{+R}$, where $\texttt{R} \in sig(P)$ and $arity(\texttt{+R}) = arity(\texttt{R}) + 1$;
3. $\texttt{-R}$, where $\texttt{R} \in sig(P)$ and $arity(\texttt{-R}) = arity(\texttt{R}) + 1$

Together, these relations define the semantics of P. Given a relational state S for $sig(P)$ (i.e., a set of tuples for each state relation), the meaning of P at S is a function that accepts individual packets and produces two things: the next relational state S' and a set of packets to output on the network.

Let S^+ be the interpretation of the **emit**, $\texttt{+R}$, and $\texttt{-R}$ relations given by Datalog semantics [AHV95]. We denote the relation $\texttt{R}$ in a state S by $\texttt{R}^S$. Now the semantics of P is the function $[[P]](S,p) = (S', \overline{p}')$ such that:

1. For each $\texttt{R} \in sig(P)$: $\texttt{R}^{S'} = (\texttt{R}^S - \texttt{-R}^{S^+}) + \texttt{+R}^{S^+}$; and
2. $\overline{p}' = \{p' \mid \langle p, p' \rangle \in \textbf{emit}^{S^+}\}$

7.3 Analyzing FlowLog

A FlowLog program naturally defines a finite-state transducer[2] on controller states. This is easy to see from our operational semantics: each transducer state is a set of relations, and each transition has arriving packets as input and the resulting output packets as output. Due to this fact, many rich analyses of FlowLog programs are not only possible [HU79], but even efficient. These include all of the analyses mentioned in Chapter 3 (e.g. change-impact analysis, verification, and bug-finding) and more; the key is that, regardless of the questions asked or the tools used, FlowLog's simple structure and carefully chosen restrictions ease analysis in general.

As an illustrative example, we describe the application of model-checking below. Model-checking is a sound and complete exploration of a finite state-space. For our experiments we used the SPIN model checker [Hol03]. First we discuss the properties to verify, then examine performance.

Verification Properties We begin by considering MAC-learning. Properties one might want to verify include:

Consistency of State The controller never learns an inconsistent state: all switch-address pairs are mapped to at most one port.

Preservation of Connectivity If a packet appears at an endpoint, it will eventually exit the network at the appropriate point.

No Loops The algorithm never induces routing loops.

Eventually Never Flood The algorithm will eventually stop flooding.

The "Eventually Never Flood" property is especially interesting. Though it is a behavioral property, it is also equivalent to saying that the second **emit** rule (listing 7.1, line 9) will always eventually stop firing. In our experience analyzing rule-based languages, we often find it easier to ascribe semantic properties to a small number of syntactic rules, making it easier to understand and debug programs.

ARP-cache shares some properties with MAC-learning and differs on others. The ARP-cache properties we considered are:

Consistency of State The controller state must not become inconsistent, meaning one IP address must not map to multiple MAC addresses.

Reliability All ARP requests will be answered.

Cache All ARP requests for known addresses will be stopped and replied to, without propagation in the network.

We haved verified each of these properties, as well as caught errors that were artificially injected. Checking against injected errors is important even when a system passes, because all system descriptions—even incorrect ones—will pass (erroneous) vacuously true property statements. Also, it is important to distinguish the times for verification and falsification, since these can be quite different.

Performance of Analysis To evaluate the performance of our analyses, we use metrics and network topologies similar to those of Canini et al. [CVP+12]. We note that our rough use of out-of-the-box model-checking tools sees performance similar to theirs, which used a custom-built checker. This is because the restricted nature of FlowLog allows for fairly simple modeling. All

[2]A finite-state automaton that reads *and writes* a symbol at each step.

tests are run on an Intel Core i5-2400 CPU with 8 GB of memory, running Windows 8 64-bit. We use Spin version 6.2.3, and make the modeling assumption that packets are injected into the network serially. For MAC-learning, we allow host mobility, limiting the number of times hosts can relocate to three. We test all MAC-learning properties on two different network topologies: an acyclic topology with two switches and a cyclic topology with three switches. For ARP-cache, we test without allowing host mobility on a two-switch topology.

Spin verifies **Consistency of State**, **Reliability** and **Cache** in under 20 seconds each, using no more than 500 MB of memory. **Preservation of Connectivity** fails in the presence of host-mobility, and Spin yields an example of this on the 2-switch topology in 140 milliseconds. **No Loops** is correct on the acyclic topology, but Spin finds potential routing loops on the cyclic topology in 40 milliseconds.

Eventually Never Flood fails on both topologies. Spin returns a short illustration of this failure in 160 milliseconds. The property fails for a somewhat subtle reason: address-port mappings are only learned locally. Thus, packets from an unlearned source address to a previously learned destination will not be flooded, preventing other switches from learning the right port for that address. This oversight can be fixed with a single FlowLog rule (not shown in the listing) that floods the *first* (and only the first) packet seen from an unknown source, even if its destination is known.

7.4 Executing FlowLog Programs

Since FlowLog is a refinement of Datalog, a first option for execution would be using an out-of-the-box Datalog engine. Given a specific packet, running such an engine would certainly yield the set of packets that ought to be emitted, as well as any necessary state edits. However, notifying the controller of every packet arrival places unnecessary overhead on both the network and the controller hardware. A real-world controller must provide flow rules proactively to the switches, and only be informed when a packet arrival might cause the controller's state to be updated or cause behavior that the switch perform do on its own.

Fortunately, there is a way to accomplish this with FlowLog. The NetCore [MFHW12] controller provides a runtime environment and a verified compiler [GRF13] from programmatically generated policies to OpenFlow switch tables. The compiler allows controllers to issue rules proactively, making use of wildcarding. We therefore intend to execute FlowLog programs within the NetCore environment, using its verified compiler whenever possible.

FlowLog programs are, however, not just NetCore policies! Instead, FlowLog programs are richer in three ways: they can access controller state, they can modify that same state, and they can make non-constant modifications to packet fields. NetCore policies are stateless, and moreover their ability to modify packet fields is restricted by the nature of OpenFlow 1.0. For instance, a flow table cannot express "swap the source and destination addresses". Thus some pieces of a FlowLog program will not be expressible in a NetCore policy.

To circumvent this problem, we divide a FlowLog program in two: rules which are simple enough to be (modulo the current controller state) compiled to flow tables, and rules which will require direct controller action in some way. Here we find another advantage of our non-recursive, rule-based design: it is easy to characterize the packets that the controller may need to take direct action on. These packets are the ones that can be matched by the bodies of state-modification rules or rules that modify packets in a non-constant fashion. The controller can process these exceptional packets using either a Datalog engine or custom algorithms. By proactively compiling as much of the program as possible to flow tables, we reduce overhead; furthermore, we can statically determine when the overhead will occur, making it easier to tune for performance if the cost of updating switches is found to be expensive. As an added benefit, automatic computation of controller-notifications eliminates the potential for bugs (and inefficiency) in controller notification.

7.5 Related Work

There have been many Datalog-derived and -related programming languages, especially in the SDN domain. Hinrichs, et al. [HGC+09] present the FML language for network management, which, like FlowLog, is based on non-recursive Datalog with negation. FML programs have far more expressivity in their rule conditions than do FlowLog programs, yet they do not provide a way for the program to modify state.

The Declarative networking effort [LCG+09] uses declarative languages to program distributed systems, focusing on temporal reasoning and message-passing between network nodes, rather than programming for a centralized controller. While these languages are designed to encourage code correctness, analysis has not been a goal equal to expressive power in their design.

Katta et al. [KRW12] describe a language called Flog which has a similar appearance to ours. Flog does not give the ability to explicitly remove tuples in the next state (as we do with `-R` rules). Instead, Flog's next state is empty unless tuples are explicitly carried over. In terms of expressive power, Flog allows recursion in its rules, but not explicit negation; we allow the opposite. Flog's forwarding policies allow *implicit* negation via rule-overriding, as in their example implementation of MAC learning. Unsurprisingly, the inclusion of recursion gives Flog a significant advantage in expressiveness over FlowLog. However, Flog is not designed with analysis in mind; both are equal considerations in FlowLog.

Field et al. [FMS09] present a declarative language model, designed for web applications, that allows a database to react to external updates. Like FlowLog, their language is inspired by Datalog, but it has significantly more expressive power than FlowLog. It supports recursion, provides numerics, and offers features designed for interaction between processes. Thus, one could view their language as a possible evolution of FlowLog, were we only concerned with expressive power and usability, and not with analysis or the embedding of third-party code. They do not discuss analysis, and the addition of recursion renders some analysis questions undecidable.

Voellmy et al. [VKF12] present Procera, a functional-reactive policy language for SDNs. Procera allows the reactive modification of state as well as packet filtering. It is more expressive than FlowLog, for instance allowing arithmetic, but does not address analysis.

Skowyra et al. [SLBK13] compile their rapid-prototyping language to model-checkers in order to verify properties and find bugs in specifications. Their language (VML) is designed for prototyping and not execution, although it is widely applicable outside SDNs. Like us, they use MAC learning as a testbed.

NICE [CVP+12] uses a mix of symbolic execution and model-checking to find bugs in NOX controllers written in Python. However, since Python is a full language, symbolic execution is unsound and required implementing a specialized model-checker. We propose a language-design strategy that will increase the available efficiency and soundness of analysis via a finite-state surface language, while still embracing tools like NICE for analysis of arbitrary call-outs, thereby ringfencing the regions of the controller program that may need to rely on unsound methods.

FlowChecker [ASAH10] analyzes low-level OpenFlow flow tables. switches. In contrast, FlowLog is an efficiently analyzable language for *controller* programming, and thus operates at a different level of abstraction. Similarly, Anteater [MKA+11] uses SAT-solving techniques to analyze the switch forwarding rules on a network. Likewise, Gutz et al. [GSSF12] perform analysis of stateless controller policies when analyzing their slice abstractions. In contrast, analysis of FlowLog must take shifting controller state into account, rather than assuming a fixed switch forwarding base. Guha et al. [GRF13] recently developed a machine-verified controller for NetCore that ensures that *static* NetCore policies are correctly translated to OpenFlow messages. In contrast, our work addresses the verification of *dynamic* control programs.

Other analysis tools such as FortNOX [PSY+12] and Veriflow [KZCG12] act only at runtime; we have focused on designing a language for efficient *static* analysis as well.

Chapter 8

Conclusion and Future Work

Misconfiguration of enterprise systems can result in severe consequences, including lost profits, identity theft, and even physical danger. Nevertheless, systems become more complex every year, and misconfigurations abound. Analysis tools such as Margrave provide a layer of defense against misconfiguration. Through Margrave and the rest of this thesis, we hope to increase the reliability of systems, as well as make static-analysis techniques more available and approachable for users who have neither the time nor the expertise to dabble in formal methods. As this work evolves, we plan to focus on providing tools not only to practitioners, but also to designers of new configuration languages.

We have shown that predicate logic–in particular, declarative first-order formalisms such as non-recursive Datalog with negation– is a suitable representation for configurations, access-control policies, and network programs. Margrave, our scenario-finding tool for configuration and policy analysis (Chapter 3), uses non-recursive Datalog with negation to represent configurations. We apply Margrave's first-order scenario-finding techniques to great effect in debugging real-world network configurations (e.g. Cisco IOS in Chapter 6). The FlowLog language for network programming (Chapter 7) also uses non-recursive Datalog with negation, but adds an explicit notion of state-change to the language, something that Margrave supports only implicitly. We used both Margrave and off-the-shelf model-checking software to analyze FlowLog programs, demonstrating how declarative first-order formalisms are amenable to different kinds of analysis.

We have also shown that declarative predicate-logic foundations support configuration-analysis tools that are *user-focused* in ways beyond just scenario-finding. Our Aluminum tool (Chapter 5) improves the concision of output scenarios as well as encourages user-driven scenario exploration. Our evaluation demonstrates that presenting *minimal* scenarios reduces both the number and the complexity of scenarios that are seen. Our finite-model theorem gives completeness guarantees for bounded scenario-finding in many common analysis situations (Chapter 4). By implementing algorithms that use this theorem in Margrave, we have shown that there are real world settings (e.g., networks) in which our finite-model theorem applies, allowing for complete, trustworthy scenario-finding.

Lessons Learned In the Margrave tool, we used a logical foundation for queries–full first-order logic–that is technically undecidable. However, we learned that general undecidability is not necessarily a deal-breaker in analysis; finite-model properties can be established that allow us to avoid undecidability in everyday use. It can also be reasonable, however, to restrict expressiveness, as seen in FlowLog. We have come to believe that configuration analysis ought to be pursued from both sides, simultaneously designing languages and analysis techniques for them. Indeed, designing new languages with analysis in mind will become more important as configurations themselves become richer, as in the software-defined networking domain. Languages designed only with expressivity in mind are easy to adopt, but tend to eventually become an analysis nightmare;

witness the Turing-completeness of `sendmail.cf`.

In Aluminum, we found that removing unnecessary information from output scenarios gave us great concision in the results. We used the same algorithm–removing unnecessary facts–to expose the consequences of adding fresh information to a scenario. However, we also discovered that presenting only minimal scenarios can unintentionally conceal important information. In particular, a fact that is possible but not necessary is not shown in a minimal scenario; we addressed this by showing a list of possible facts to add.

8.1 Future Work

Each sub-project in this thesis presents multiple avenues for future work.

8.1.1 Margrave

Our long-term goal is to grow Margrave into a mature tool that is useful to both academics and industry practitioners alike. This will involve supporting more and richer configuration languages, but also continuing to push Margrave's user focus to make the tool more appealing. Some selected improvements are:

- We have applied Margrave to many different kinds of policies, including access-control, simple hypervisors, product-line configuration, firewalls, and routing configuration. Margrave's general-purpose flexibility also supports reasoning about *interactions* between different kinds of policies, such as a firewall and an access-control policy for database access. Support for heterogeneous policy federations is increasingly relevant in cloud deployments, and is an exciting possibility for future work.

- Margrave's performance is reasonable, but slower than other domain-specific analysis tools; this is especially true in the firewall analysis, where tailoring the underlying data structures to the domain yields dividends (as seen in, e.g. Liu and Gouda's work [LG09]). In particular, we expect Margrave will scale poorly to large networks of firewalls, as our formulas grow linearly with the number of firewalls. Our use of SAT-solving instead of BDDs may be another factor, though Jeffrey and Samak's comparisons between these for firewall analysis [JS09] are inconclusive. Exploring alternative back-ends—whether based on BDDs or other solvers—is another area for future work. Another, similar direction would be to create better data structures for representing policies and formulas in Margrave. The current tool represents formulas using the Kodkod abstract-syntax classes; this leads to inefficiency in larger policies since the syntax tree does not self-simplify. For instance, $\neg P(x) \wedge P(x)$ (and far more lengthy such formulas, in practice) are not reduced to *false* before being handed to Kodkod. The right balance of speed and expressiveness must be found, and the obvious data structures, like BDD s, only generally apply to propositional, not first-order, logic. The fact that Kodkod converts first-order formulas to propositional ones is no help here; the conversion happens too late to deliver the benefits we desire.

- Real-world configurations often involve arithmetic or at least numerics. Margrave can currently support a limited notion of numerics, but doing so requires building scaffolding for (among other things) linear ordering into the vocabulary. Better numeric support would increase Margrave's usefulness in the system administration domain, among others. Margrave also lacks support for recursively-defined policies; allowing recursive policies would enable Margrave to better handle aspects of trust management, such as delegation.

- Policies are often embedded in a larger system. Before taking an action, the system queries the policy. This interaction leads to subtle questions about policy analysis, especially when *system state* is involved. We intend to make a deeper exploration of state and *state updates* in policies, and an understanding of what kinds of state can be expressed fully in a policy rather than a wrapper program. In our firewall analysis work, we accounted for reflexive access lists, a simple kind of state that appears in firewalls. But support for stateful, protocol-aware packet inspection would be far more complex, and has yet to be modeled in Margrave. Our FlowLog language from Chapter 7 takes a first step in this direction.

- Finally, Margrave would greatly benefit from better support for additional real-world policy languages and improved interfaces for the languages it already supports. For example, Margrave supports a reasonable fragment of Cisco's IOS configuration language, but the scenario output is generic, and not tailored to the networking domain, and network topology must be encoded into the queries themselves. We wish to make Margrave an attractive option for users of existing languages as well as creators of new languages. By allowing the tool to tailor both its interface and its output to a domain, we would improve usability in both cases.

8.1.2 Finite-Model Theorems

Our work on a finite-model theorem for order-sorted predicate logic suggests two major lines of further inquiry. The first is the exploration of new efficient algorithms for working with order-sorted effectively-propositional logic sentences. A natural approach is to leverage insights from existing tools for model-finding and theorem-proving that are currently optimized for the traditional EPL class. The other direction is to pursue a program of classifying fragments of order-sorted logic according to decidability. Abadi *et al.* [ARS10] suggest a taxonomy based on quantifier prefix patterns but, as pointed out in the introduction, prenex-normal form is not available when sorts are allowed to be empty. We propose that a combinatorial analysis of the signature of Skolemizations of sentences is the proper generalization of the analysis of classical quantifier prefix classes.

8.1.3 Aluminum

The most obvious continuation of the Aluminum project is to make minimization and augmentation standard features of the Alloy Analyzer. This modification would enable users to maximize their gain from the scenarios that Alloy returns, and not require them to perform their minimization in our special tool.

There exist situations where minimal scenarios are actually counterproductive (such as detecting failed upper-bound frame conditions). We believe that this fact only underscores the need to investigate new strategies for scenario-exploration as well as how they might interact. For example, a tool presenting only *maximal* scenarios (up to a finite size bound) would be a dual to Aluminum: always demonstrating failed upper-bound frame conditions, but not lower-bound failures. Providing both modes of exploration to a user might be useful.

There are unanswered questions revolving around minimization. Aluminum was originally conceived to produce *homomorphically* minimal scenarios. That is, minimality would be computed with respect to three things: existence of atoms in the scenario, relational facts about those atoms (corresponding to tuples in relations), and the *equation* of atoms. However, the existing tool only minimizes with respect to the first two aspects. To obtain the third, we would need to encode equality separately from Kodkod's built-in support, which was unfeasible at the time.

At this time, evaluation of these tools has been either quantitative or via informal user feedback. We have not conducted rigorous user-studies on either Margrave or Aluminum. Our quantitative results have been extremely strong, and initial user feedback has been positive. In fact, we have received substantial interest in incorporating Aluminum's functionality into Alloy itself.

8.1.4 FlowLog

FlowLog, our proposed language for network programming from Chapter 7, is only part of a larger research program to produce a tractable, expressive, analysis-friendly language that enables rich reasoning to create reliable controllers. The goal of FlowLog was to broaden the notion of state to which we could bring Margrave-style analyses. Thus, Flowlog rules can modify state as well as read it.

Programming controllers entirely in languages with limited expressive power, e.g., finite-state languages like FlowLog, is attractive from an analytic standpoint, but impractical in the general case. Controller authors need to take advantage of the full power of programming on some instances, and methodologies that don't provide it will (rightly) be deemed impractical.

Future work will make possible an alternative SDN programming experience based on the following tenets:

- Controller authors should have the freedom to use both restricted and full languages. Which one they use is a function of the analytic power they need, previously written code they wish to reuse, etc.
- A single controller program should be able to combine elements from both restricted and full languages. This provides the most flexibility and puts the trade-off in the hands of the author.
- Because of the difficulty of recovering reasoning power once we permit full expressive power, we prefer that the restricted language reside "outside" and the "full" language play the role of callouts or libraries.

In short, we are aiming for the Alan Kay principle of "Easy things should be easy, while hard things should be possible".

From an analysis perspective, the restricted outer language will be amenable to rich, sound, and complete analyses. We are not interested only in "verification", as can be seen in Chapters 3 and 6. Change-impact analysis and other forms of property-free analysis will feature prominently. Having a restricted language also opens the door to sophisticated forms of synthesis and much else.

But what to make of the embedded full-language components? Our view is that performing the analysis over the restricted language will enable us to automatically infer *constraints*, such as preconditions, postconditions, and invariants, on the behavior of the full language components being invoked. Discharging those constraints becomes a language-specific, task-specific problem: mechanisms may range from using existing tools for controller languages (such as NICE [CVP+12] for Python) to applying richer model-checking techniques to performing dynamic monitoring of invariants.

We have focused so far on the design of a decidable yet expressive language. Because our design is based on permitting external code in full languages, future work will proceed on two fronts:

1. Automatically extracting interfaces on callouts. We can exploit the long history of work on interface generation in the verification community, though the nature of composition here—being sequential rather than parallel—has been studied less.
2. Focusing on tasks *other than* traditional verification. Our primary emphasis will be change-impact analysis, but we are also interested in program synthesis approaches.

Appendix A

Preliminary User Study

This appendix contains a two-part user study that we designed to evaluate how people use scenario-finding tools, along with how scenarios might be tailored (e.g. the minimal scenarios of Chapter 5) to improve the benefit they gain from these tools. Concretely, we crafted this study to help us learn:

1. How are scenario-finding tools such as Alloy used in practice?
2. Given a flood of scenarios, how soon will a user simply stop browsing?
3. What kinds of scenarios might a user wish to see *first*? Are minimal scenarios useful to see first?
4. What scenarios might a user wish to see *next*, having already seen one? Is the "augmentation" paradigm, in which the next scenario adds facts to the prior one, something that they find useful?

We are also interested in the qualitative reasoning behind users' answers.

A.1 Survey Discussion

In order to encourage participation, we divided the study into two parts: a short-form survey containing broad but informative questions and a longer-form version where participants are asked to consider which scenarios they would like to see for three different specifications. We designed the short-form survey to take roughly five minutes, and the longer version approximately ten additional minutes. This expectation was borne out by internal testing.

A.1.1 Short-Form Survey

The short-form part of this survey contains nine questions, and is reproduced in its entirety in Section A.2. Questions 1 – 3 obtain information about the participant's usage of formal-methods tools, as well as their current position. Question 4 inquires about what motivates them to use scenarios, and Questions 5 and 7 delve deeper into how they use scenarios. Question 6 is meant to discover whether participants even consider necessity (and thus, minimality) when looking at scenario output. Questions 8 and 9 give the participant an open-ended space in which to write about features that they would like to see in scenario-finding tools and additional information about how they use scenarios.

While this survey is brief, it gathers a great deal of information about how people use scenario-finding tools. Question 6 is particularly useful: do users of these tools even think about facts in scenarios in terms of necessity, or is minimality an entirely new way of thinking to them?

A.1.2 Long-Form Survey

The long-form survey (reproduced in Section A.3) contains three example specifications and questions about each. The three specifications use different mathematical concepts and different domains, which we believe will help rule out or discover user bias. Each specification contains some surprising behavior, and some example scenarios illustrate that behavior. When there are multiple variations of the bug (as in Specification 1), we use different scenarios to illustrate each variation and one to illustrate both simultaneously. In each question, we have included at least one minimal scenario, at least one extension of that scenario (e.g. by adding new facts or items), and at least one "large" scenario that may involve several unnecessary facts.

A.1.2.1 Long-Form Specification 1: Address Book

The address-book specification comes from Jackson [Jac12]. The specification has a bug: an address-book name (either a group or an alias) can fail to point to a concrete address. We are interested in whether the user notices the bug, and which scenarios help them to do so. The bug manifests in Scenarios 1, 3, 4, and 5. Scenarios 1 and 5 are minimal, up to a bound requiring at least two names to exist. Thus, the bug is present in both minimal and non-minimal scenarios.

Scenario 1 demonstrates that aliases can be empty. Scenario 5 shows that groups can be empty. Scenario 3 contains both examples of the bug: it contains both an empty group and an empty alias, yet it is an extension (up to isomorphism) of Scenario 1, and thus not minimal. Scenario 4 contains only an empty alias, and is not minimal.

Our expectation is that users would prefer the non-minimal, but doubly useful Scenario 3. If first given Scenario 1 or Scenario 5, we expect users to prefer either the appropriate counterpart (Scenario 1 leading to Scenario 5 or vice versa) or Scenario 3. If our expectation is false, we expect to gain insight into this different preference via users' qualitative comments.

A.1.2.2 Long-Form Specification 2: Grade Book

The grade-book specification is of our own design. It contains a bug that is only seen in *non-minimal* scenarios: it is possible for TAs to grade their own assignments.

Scenarios 1 and 5 illustrate the bug. The minimal scenario 2 has no TAs, and in Scenarios 3 and 4, no TA authored the assignment. Scenario 5 is an extension of Scenario 1 that adds an (unnecessary) extra instructor. Since the specification does not require the presence of a TA, only scenario 2 is truly minimal.

We expect Scenario 1 to be preferred, since it demonstrates the problem with the least amount of extra information. If the minimal Scenario 2 is seen first, we hope that scenario 1 will be most desirable as a second scenario.

A.1.2.3 Long-Form Specification 3: Conference Manager

This specification, also of our own design, models a fragment of a conference-manager. The surprising behavior here is that a single review can cover multiple papers. Since the constraints only say that there must be one of each type of object (paper, reviewer, etc.) only Scenario 1 is minimal. Scenarios 2, 3, and 4 all illustrate the bug, but Scenario 2 is minimal *among the three.*

We expect that Scenario 2 would be preferred, followed by either Scenario 3 or Scenario 4. If Scenario 3 or 4 is preferred over Scenario 2, we believe that this may be due to perception of the minimal scenarios as an "incomplete" corner-case; we will validate this through qualitative feedback.

A.1.3 Cautions

There are some caveats to keep in mind about these surveys.

Possibility and Augmentation This survey has focused mainly on minimality and modes of scenario use. While no example scenario for Specification 3 shows a paper with more than one author, nothing in the specification prevents it. If the specification did in fact prevent multiple-author papers, it would be negative information that could not be extracted from a single scenario. Such bugs are beyond the scope of this study, although one might create a followup study to evaluate the effectiveness enriching scenarios with (im)possibility information. Our Aluminum tool (Chapter 5) also gives users the ability to explore the consequences of adding new facts to output scenarios. An expanded study could evaluate the usefulness of this feature as well.

Preference vs. Effectiveness This survey tests which scenarios users *prefer*, but not which scenarios are actually most *effective in leading them to the bugs* in each specification. Even if users do not prefer minimal scenarios, it may be that minimality and augmentation, as seen in Aluminum, are still helpful. Thus, this study could benefit from revision or extension to test for scenario effectiveness. There is also a risk that the small size of these scenarios do not allow the benefits of minimality to come across clearly; an on-line version of this study could include a fourth long-form example with larger scenarios.

A.2 Short-Form Survey

The short-form survey follows on the next page. Page formatting of the survey has been preserved.

This survey explores how people use concrete models/instances/counterexamples to analyze and understand formal specifications. It is being conducted by formal methods researchers from WPI and Brown University: Kathi Fisler, Tim Nelson, Shriram Krishnamurthi, and Dan Dougherty.

This survey uses the term **scenario** for a concrete example that witnesses the success or failure of a property or query. It includes instances, counterexamples, and models (from mathematical logic).

1. Which of the following formal-methods tools do you use? Check all that apply:

___ Alloy	___ ASM
___ B	___ SAT solvers (directly, not through tools like Alloy)
___ Z	___ Model checkers

Other (please specify): ____

2. Which of the following best describe your current position?
   - ☐ *Masters student*
   - ☐ *PhD student*
   - ☐ *University faculty (professor, lecturer, etc.)*
   - ☐ *Industry or government employee*
   - ☐ *Other:* ______________________________________

3. For Alloy and other formal methods tools, which statement best describes your usage?

	Alloy	Other tools
I've never used it		
I've worked through small examples (e.g., from a tutorial or textbook)		
I've used it in class (as a student or professor), but not outside of class		
I do research on it, but don't model or analyze significant projects with it		
I've built other systems that incorporate it, but don't use it directly		
I've used it for a nontrivial part of a research project		
Other (please explain):		

4. How often do you find scenarios useful for each of the following tasks?

	Rarely	Sometimes	Often
Help find the source of a behavior or bug in a specification			
Suggest additional properties to check of a specification			
Suggest additional constraints required in a specification			
Identify unexpected relationships between parts of a specification			
Foster human confidence in the behavior of a specification			
Other (please describe):			

5. Imagine you use a tool to run a property or query against a specification. The tool produces a set of scenarios. How often do you use each of the following approaches to explore those scenarios?

	Rarely	Sometimes	Often
Trace each scenario through the specification before viewing another			
Inspect several scenarios at a high-level, then go back and trace (some of) their contents to the specification			
Refine query to include some fact(s) in a recently-viewed scenario			
Ask a query that includes some fact(s) ***not*** *seen in the scenarios so far*			
Ask a refined query that checks whether two distinct elements in an existing scenario could merge into a single element			
Edit the specification as soon as a scenario indicates an issue, then re-run the original query			
Other (please describe):			

6. When using a tool that produces scenarios in response to a query, how often do you wonder whether a particular fact or element in the scenario is *necessary* to satisfy the query?

Rarely Sometimes Often

7. When using a tool that produces scenarios, how often do the contents of one scenario suggest a new property or query that you hadn't previously identified?

Rarely Sometimes Often

8. What are your favorite or desired features in tool support for exploring scenarios?

9. Any additional comments on your usage of scenarios in formal analysis?

Thank you! If you'd be willing to respond to further questions about how you use scenarios, please continue to the next section and/or provide a contact email address:__________________________

A.3 Long-Form Survey

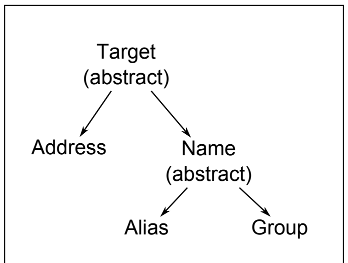
The long-form survey begins on the next page. Page formatting of the survey has been preserved.

Detailed Questions

This survey evaluates which scenarios are most helpful for confirming that a specification behaves as expected. We will show you three specifications. For each, we show five (of the many) scenarios that are consistent with the specification. We want to know which of the five are helpful to see first, and which you would want to see next. Please fill in the answer table under each specification. Thank you for participating!

Specification 1: Address Book

Consider the following specification for an address book that supports groups of addresses and aliases for single addresses. The hierarchy of types (aka, sorts, signatures, or domains) is on the left. The relations and constraints are on the right.



Address-book Type Hierarchy

Relations

```
addr: Name × Target
```

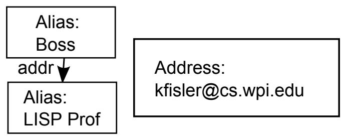
Constraints

1) `addr is acyclic`
2) `addr maps each alias to` *`no more than one`* `name`
3) `there is at least one Address`

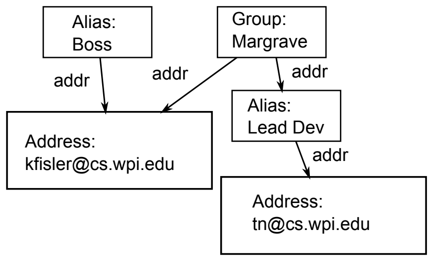
Assuming your analysis tool were going to show the five scenarios on the **next page** (numbered 1 to 5) in some linear order, please fill in the following table:

Scenario Number	Helpful to see first? (check all that apply)	If checked, numbers of up to two other scenarios to show next	Please explain your response (for the whole row)
1	☐		
2	☐		
3	☐		
4	☐		
5	☐		

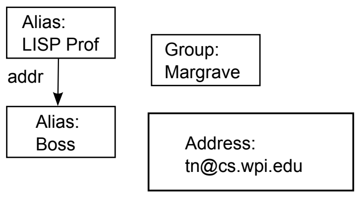
Address book Scenario 1



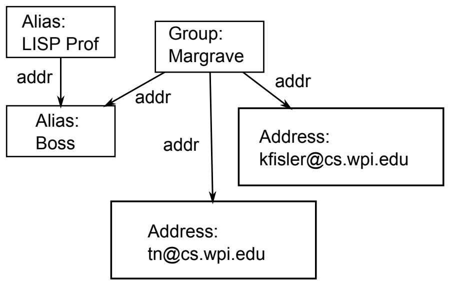
Address book Scenario 2



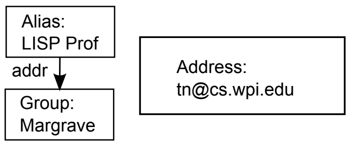
Address book Scenario 3



Address book Scenario 4

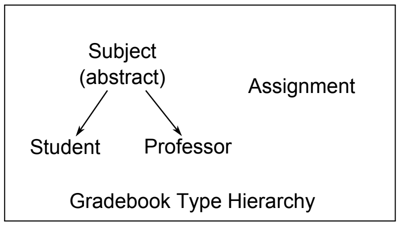


Address book Scenario 5



Specification 2: Homework and Grading Tool

Consider the following specification of which users of a homework and grading tool are allowed to submit work and grade assignments. The hierarchy of types (aka, sorts, signatures, or domains) is on the left. The relations and constraints are on the right.



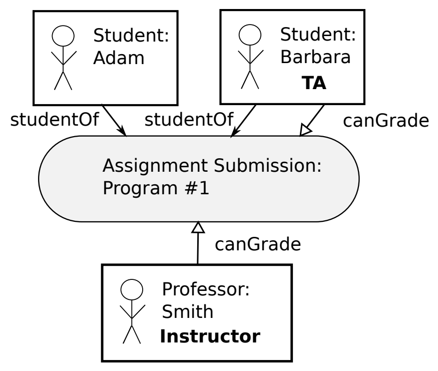
```
Relations
TA: Student
instructor: Professor
studentOf: Assignment × Student

canGrade: Subject × Assignment
```

```
Constraints
1) canGrade(s, a) ⟺
          TA(s) or instructor(s)
2) studentOf(a) is non-empty
   for all Assignments a
```

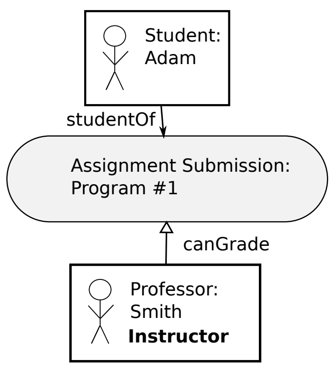
Assuming your analysis tool were going to show the five scenarios on the **next page** (numbered 1 to 5) in some linear order, please fill in the following table:

Scenario Number	Helpful to see first? (check all that apply)	If checked, numbers of up to two other scenarios to show next	Please explain your response (for the whole row)
1	☐		
2	☐		
3	☐		
4	☐		
5	☐		

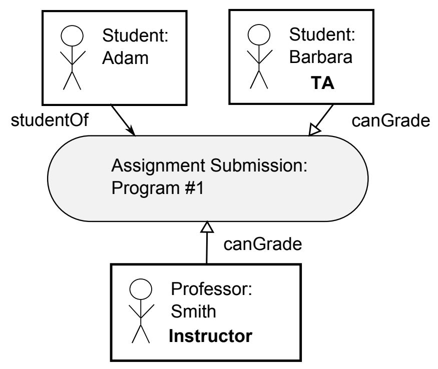
Grading Scenario 1



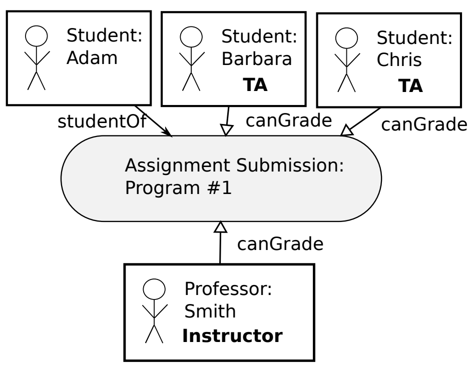
Grading Scenario 2


Student: Adam
studentOf
Assignment Submission: Program #1
canGrade
Professor: Smith Instructor

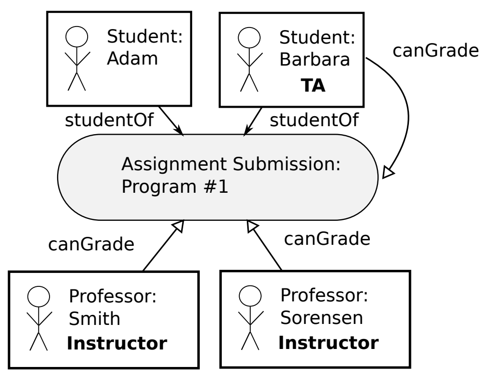
Grading Scenario 3


Student: Adam
Student: Barbara TA
studentOf
canGrade
Assignment Submission: Program #1
canGrade
Professor: Smith Instructor

Grading Scenario 4

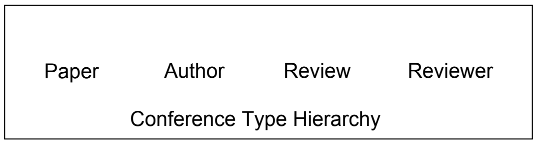


Grading Scenario 5


Student: Adam
Student: Barbara TA
canGrade
studentOf
studentOf
Assignment Submission: Program #1
canGrade
canGrade
Professor: Smith Instructor
Professor: Sorensen Instructor

Specification 3: Conference Manager

Consider the following specification of the relationships between reviews, reviewers, papers, and authors in a conference manager. The hierarchy of types (aka, sorts, signatures, or domains) is on the left. The relations and constraints are on the right.



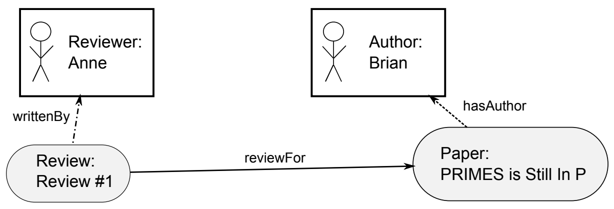
```
Relations
hasAuthor: Paper × Author
reviewFor: Review × Paper
writtenBy: Review → Reviewer
```

```
Constraints
1) There is at least one paper,
   author, review, and reviewer;
2) Relations hasAuthor and reviewFor
   are total (i.e. every paper has at
   least one author and every review is
   about at least one paper)
```

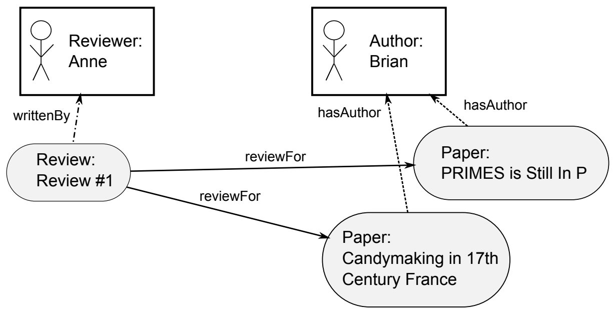
Assuming your analysis tool were going to show the five scenarios on the **next page** (numbered 1 to 5) in some linear order, please fill in the following table:

Scenario Number	Helpful to see first? (check all that apply)	If checked, numbers of up to two other scenarios to show next	Please explain your response (for the whole row)
Scenario 1	☐		
Scenario 2	☐		
Scenario 3	☐		
Scenario 4	☐		
Scenario 5	☐		

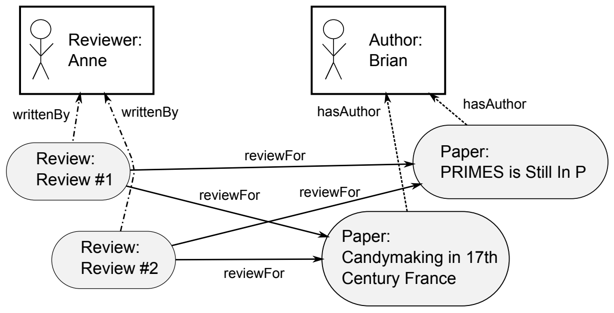
Conference Scenario 1



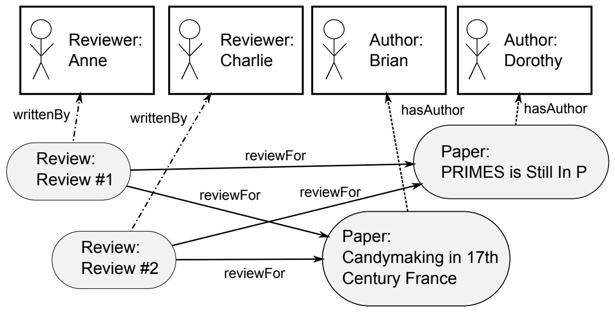
Conference Scenario 2



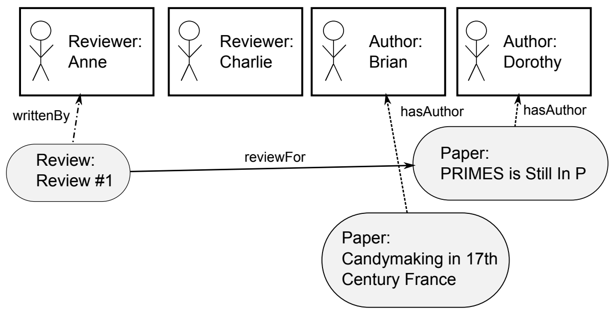
Conference Scenario 3



Conference Scenario 4



Conference Scenario 5



Bibliography

[ABL$^+$10] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *IEEE Computer Security Foundations Symposium*, 2010.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[Alg] The AlgoSec Firewall Analzyer. `www.algosec.com`.

[Ama11] Amazon Web Services Team. Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region. `http://aws.amazon.com/message/65648/`, 2011. Accessed September 23, 2011.

[ARS10] Aharon Abadi, Alexander Rabinovich, and Mooly Sagiv. Decidable fragments of many-sorted logic. *Journal of Symbolic Computation*, 45(2):153 – 172, 2010. Automated Deduction: Decidability, Complexity, Tractability.

[ASAH10] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Workshop on Assurable and Usable Security Configuration*, 2010.

[ASH03] Ehab S. Al-Shaer and Hazem H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Integrated Network Management*, pages 17–30, 2003.

[ASH04] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of policy anomalies in distributed firewalls. In *IEEE Conference on Computer Communications*, 2004.

[azs08] azsquall. "ACL and NAT conflict each other. router stop working". `www.networking-forum.com/viewtopic.php?f=33&t=7635`, August 2008. Access Date: July 20, 2010.

[BBOR08] Sruthi Bandhakavi, Sandeep Bhatt, Cat Okita, and Prasad Rao. End-to-end network access analysis. Technical Report HPL-2008-28R1, HP Laboratories, November 2008.

[BEW95] R.A. Becker, S.G. Eick, and A.R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1995.

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer, 1997.

[BKM$^+$04] Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila Takayama, and Madhu Prabaker. Field studies of computer system administrators: Analysis of system management tools and practices. In *ACM Conference on Computer Supported Cooperative Work*, pages 388–395, 2004.

[BOR08] Sandeep Bhatt, Cat Okita, and Prasad Rao. Fast, cheap, and in control: A step towards pain-free security! In *USENIX Large Installation System Administration Conference*, pages 75–90, 2008.

[BP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 2010.

[BR08] Sandeep Bhatt and Prasad Rao. Enhancements to the Vantage Firewall Analyzer. Technical Report HPL-2007-154R1, HP Laboratories, June 2008.

[BS28] Paul Bernays and Moses Schönfinkel. Zum entscheidungsproblem der mathematischen Logik. *Mathematische Annalen*, 99:342–372, 1928.

[BY00] F. Bry and A. Yahya. Positive unit hyperresolution tableaux and their application to minimal model generation. *Journal of Automated Reasoning*, 2000.

[BYBJ09] Nihel Ben Youssef, Adel Bouhoula, and Florent Jacquemard. Automatic verification of conformance of firewall configurations to security policies. In *IEEE Symposium on Computers and Communications*, pages 526 – 531, July 2009.

[CGLR96] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning*, 1996.

[CK73] Chen Chung Chang and Jerome Keisler. *Model Theory.* Number 73 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1973. Third edition, 1990.

[Cla78] Keith L. Clark. Negation as failure. *Logic and Data Bases*, 1978.

[CS03] K. Claessen and N. Sorensson. New techniques that improve MACE-style finite model finding. In *International Conference on Automated Deduction*, 2003.

[CVP$^+$12] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *Networked Systems Design and Implementation*, 2012.

[DGT07] Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. Searching for shapes in cryptographic protocols. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2007.

[dMB08a] Leonardo de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, page 337. Springer, 2008.

[dMB08b] Leonardo de Moura and Nikolaj Bjørner. Deciding Effectively Propositional Logic Using DPLL and Substitution Sets. In *International Joint Conference on Automated Reasoning*, pages 410–425, 2008.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[EZ01] Pasi Eronen and Jukka Zitting. An expert system for analyzing firewall rules. In *Nordic Workshop on Secure IT Systems*, pages 100–107, 2001.

[FG03] Pascal Fontaine and E. Pascal Gribomont. Decidability of invariant validation for parameterized systems. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 97–112. Springer-Verlag, 2003.

[FHF$^+$11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *International Conference on Functional Programming (ICFP)*, 2011.

[FKMT05] Kathi Fisler, Shriram Krishnamurthi, Leo Meyerovich, and Michael Tschantz. Verification and change impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, 2005.

[FMS09] John Field, Maria-Cristina Marinescu, and Christian Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theoretical Computer Science*, 2009.

[FP10] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `racket-lang.org/tr1/`.

[FUV83] R. Fagin, J.D. Ullman, and M.Y. Vardi. On the semantics of updates in databases. In *Symposium on Principles of Database Systems*, 1983.

[GdM09] Yeting Ge and Leonardo de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *International Conference on Computer Aided Verification*, pages 306–320, Berlin, Heidelberg, 2009. Springer-Verlag.

[Gia12] Theophilos John Giannakopoulos. Multi-decision policy and policy combinator specifications. Master's thesis, Worcester Polytechnic Institute, February 2012.

[GLP09] Swati Gutpa, Kristen LeFevre, and Atul Prakash. SPAN: A unified framework and toolkit for querying heterogeneous access policies. In *USENIX Workshop on Hot Topics in Security*, 2009.

[GM92] Joseph A. Goguen and José Meseguer. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[GRF13] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. In *Programming Language Design and Implementation (PLDI)*, 2013.

[GSSF12] Stephen Gutz, Alec Story, Cole Schlesinger, and Nate Foster. Splendid isolation: A slice abstraction for software-defined networks. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[Gut97] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *IEEE Symposium on Security and Privacy*, pages 120–129, 1997.

[Harpt] John Harrison. Exploiting sorts in expansion-based proof procedures, Unpublished manuscript. `http://www.cl.cam.ac.uk/~jrh13/papers/manysorted.pdf`.

[HGC$^+$09] Timothy Hinrichs, Natasha Gude, Martin Casado, John Mitchell, and Scott Shenker. Practical declarative network management. In *Workshop: Research on Enterprise Networking (WREN)*, 2009.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[HRCS02] JN Hooker, G. Rago, V. Chandru, and A. Shrivastava. Partial instantiation methods for inference in first-order logic. *Journal of Automated Reasoning*, 28(4):371–396, 2002.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2nd edition, 2012.

[Jan10] Mikoláš Janota. *SAT Solving in Interactive Configuration.* PhD thesis, University College Dublin, November 2010.

[Jer88] R. G. Jereslow. Computation-oriented reductions of predicate to propositional logic. *Decision Support Systems*, 4:183–197, 1988.

[JS09] Alan Jeffrey and Taghrid Samak. Model checking firewall policy configurations. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, 2009.

[Jus10] Just Another WordPress Weblog. WP.com downtime summary. `http://en.blog.wordpress.com/2010/02/19/wp-com-downtime-summary/`, 2010. Accessed September 23, 2011.

[KHM+07] S. Krishnamurthi, P.W. Hopkins, J. McCarthy, P.T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007.

[KL10] Amir R. Khakpour and Alex X. Liu. Quantifying and querying network reachability. In *International Conference on Distributed Computing Systems*, June 2010.

[KNFH09] Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, and Ryuzo Hasegawa. Minimal model generation with respect to an atom set. In *International Workshop on First-Order Theorem Proving*, 2009.

[KRW12] Naga Praveen Katta, Jennifer Rexford, and David Walker. Logic programming for software-defined networks. In *Workshop on Cross-Model Design and Validation (XLDI)*, 2012.

[KZCG12] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[LCG+09] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[Lew80] HR Lewis. Complexity results for classes of quantificational formulas. *J. COMP. AND SYS. SCI.*, 21(3):317–353, 1980.

[LG08] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems*, 19(8), August 2008.

[LG09] Alex X. Liu and Mohamed G. Gouda. Firewall policy queries. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):766–777, June 2009.

[LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* The MIT Press, 1992.

[LS04] S.K. Lahiri and S.A. Seshia. The UCLID decision procedure. In *International Conference on Computer Aided Verification*, pages 475–478. Springer, 2004.

[LWK08] Sihyung Lee, Tina Wong, and Hyong S. Kim. Improving dependability of network configuration through policy classification. In *IEEE/IFIP Conference on Dependable Systems and Networks*, 2008.

[LWK09] Sihyung Lee, Tina Wong, and Hyong S. Kim. Netpiler: Detection of ineffective router configurations. *IEEE Journal on Selected Areas in Communications*, 27(3):291–301, 2009.

[Mar] The Margrave Policy Analzyer. `www.margrave-tool.org/v3/`.

[McC01] William McCune. MACE 2.0 reference manual and guide. *CoRR*, 2001. cs.LO/0106042.

[MFHW12] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *Principles of Programming Languages (POPL)*, 2012.

[MK05a] Robert Marmorstein and Phil Kearns. An open source solution for testing nat'd and nested iptables firewalls. In *USENIX Large Installation System Administration Conference*, 2005.

[MK05b] Robert Marmorstein and Phil Kearns. A tool for automated iptables firewall analysis. In *USENIX Annual Technical Conference*, 2005.

[MK06] Robert Marmorstein and Phil Kearns. Firewall analysis with policy-based host classification. In *USENIX Large Installation System Administration Conference*, 2006.

[MKA$^+$11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[Mom05] Lee Momtahan. Towards a small model theorem for data independent systems in Alloy. *Electronic Notes in Theoretical Computer Science*, 128(6):37 – 52, 2005. International Workshop on Automated Verification of Critical Systems.

[MR12] Vajih Montaghami and Derek Rayside. Extending alloy with partial instances. In *International Conference on Abstract State Machines, Alloy, B, and Z*, 2012.

[MWZ00] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A firewall analysis engine. In *IEEE Symposium on Security and Privacy*, pages 177–187, 2000.

[MWZ05] Alain Mayer, Avishai Wool, and Elisha Ziskind. Offline firewall analysis. *International Journal of Information Security*, 2005.

[NBD$^+$10] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave tool for firewall analysis. In *USENIX Large Installation System Administration Conference*, 2010.

[NDFK12] Timothy Nelson, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Toward a more complete Alloy. In *International Conference on Abstract State Machines, Alloy, B, and Z*, 2012.

[Nie96] Ilkka Niemelä. A tableau calculus for minimal model reasoning. In *Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, 1996.

[NSD$^+$13] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: Principled scenario exploration through minimality. In *International Conference on Software Engineering*, 2013.

[OAS05] OASIS. Oasis extensible access control markup language, 2005. http://www.oasis-open.org/committees/xacml.

[Obe89] A. Oberschelp. Order sorted predicate logic. In *Workshop on Sorts and Types in Artificial Intelligence*, pages 1–17. Springer, 1989.

[Oel09] Oelolemy. "problem with policy based routing- urgent please !". `www.experts-exchange.com/Networking/Network_Management/Q_24113014.html`, February 2009. Access Date: July 20, 2010.

[OGP] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it?

[OLK09] Ricardo M. Oliveira, Sihyung Lee, and Hyong S. Kim. Automatic detection of firewall misconfigurations using firewall and network routing policies. In *DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance*, 2009.

[PdMB08] Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding Effectively Propositional Logic with Equality. Technical Report MSR-TR-2008-181, Microsoft Research, December 2008.

[PSY$^+$12] Phillip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofi Gu. A security enforcement kernel for openflow networks. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[Ram30] Frank P. Ramsey. On a problem in formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930.

[Rob01] A. Robinson. *Handbook of Automated Reasoning*, volume 2. Elsevier, 2001.

[Rob10] Robert Johnson. More details on today's outage. `http://www.facebook.com/note.php?note_id=431441338919`, 2010. Accessed September 23, 2011.

[Sco10] Scott Guilfoyle. Details on paypals site outage today. `https://www.thepaypalblog.com/2010/10/details-on-paypals-site-outage-today/`, 2010. Accessed September 23, 2011.

[Shl07] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 2007.

[SLBK13] Richard Skowyra, Andrei Lapets, Azer Bestavros, and Assaf Kfoury. Verifiably-safe software-defined networks for CPS. In *High Confidence Networked Systems (HiCons)*, 2013.

[TASB07] T. Tran, E. Al-Shaer, and R. Boutaba. PolicyVis: firewall security policy visualization and inspection. In *USENIX Large Installation System Administration Conference*, 2007.

[TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[VKF12] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Workshop on Hot Topics in Software Defined Networking*, 2012.

[VP05] Pavan Verma and Atul Prakash. FACE: A firewall analysis and configuration engine. In *Symposium on Applications and the Internet*, 2005.

[Wil12] R.R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing.* Academic Press, 2012.

[Woo01] Avishai Wool. Architecting the lumeta firewall analyzer. In *USENIX Security Symposium*, 2001.

[Woo04] Avishai Wool. A quantitative study of firewall configuration errors. *Computer*, 37(6):62–67, 2004.

[Woo10] Avishai Wool. Trends in firewall configuration errors: Measuring the holes in swiss cheese. *IEEE Internet Computing*, 14(4):58–65, 2010.

[YMS$^+$06] L. Yuan, J. Mai, Z. Su, H. Chen, C-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for FIREwall Modeling and ANalysis. In *IEEE Symposium on Security and Privacy*, 2006.

[ZZ95] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *International Joint Conference On Artificial Intelligence*, 1995.

First-Order Models for Configuration Analysis

by

Tim Nelson

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

May 2013

APPROVED:

[illegible]

Professor Kathi Fisler, Thesis Co-Advisor, WPI Computer Science

D J Dougherty

Professor Daniel J. Dougherty, Thesis Co-Advisor, WPI Computer Science

[illegible]

Professor Craig Wills, Head of Department, WPI Computer Science

Joshua Guttman

Professor Joshua Guttman, WPI Computer Science

[illegible]

Professor Shriram Krishnamurthi, Brown University Computer Science