Advancing Declarative Programming

by

Aleksandar Milicevic

B.S., School of Electrical Engineering, University of Belgrade (2007)
M.S., Massachusetts Institute of Technology (2010)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 15, 2015

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

Accepted by . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Advancing Declarative Programming

by

Aleksandar Milicevic

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis attempts to unite and consolidate two large and often culturally disjoint programming paradigms: *declarative* (focusing on specifying *what* a program is supposed to do, e.g., shuffle an array so that its elements are ordered) and *imperative* (detailing *how* the program intention is to be implemented, e.g., by applying the QuickSort algorithm). The ultimate result of such an effort would be a unified programming environment in which both paradigms are seamlessly integrated, specifications are fully and efficiently executable, and programs are written by freely mixing imperative statements and declarative specifications.

With the advent of automated constraint solving, executing declarative specifications as standalone programs has become feasible. A number of challenges still remain. To achieve full automation, constraint solvers often impose restrictions on specification languages and their expressiveness; compromises are also made when integrating a (typically logic-based) specification language with a traditional procedural programming language; and finally, applicability is usually limited to specialized algorithmic domains (for which constraint solving is particularly suitable) and programmers comfortable with writing formal logic.

This thesis proposes several advances to address these issues. First, a novel constraint solving framework is presented, Alloy*, the first of its kind capable of automatically and reliably solving arbitrary higher-order formulas (written in standard predicate logic) over bounded domains. Second, a new approach to integrating a specification and an implementation language is proposed, where Alloy, a relational logic-based modeling and specification language, is deeply embedded in Ruby. The resulting platform, called αRby, uses Alloy* as its back end, and serves both as an Alloy modeling environment with added Ruby scripting layer around it, and as a Ruby programming environment with added executable specifications. Third, the general idea of declarative programming (focusing on what instead of how) is applied to web programming, producing SUNNY, a model-based reactive web framework with a clear separation between data, events (business logic), and security policies. SUNNY is (1) policy-agnostic—allows security policies to be specified individually and independently from the rest of the code, (2) reactive—automatically propagates data updates to all connected clients while enforcing the security policies, (3) mostly declarative—offers a unified sequential view of the entire distributed web system, allowing events to be implemented only in terms of simple modifications to the data model.

Thesis Supervisor: Daniel N. Jackson
Title: Professor

Acknowledgments

This research was funded in part by the National Science Foundation under grants 0707612 (CRI: CRD—Development of Alloy Tools, Technology and Materials), and 1138967 (Collaborative Research: An Expedition in Computing for Compiling Printable Programmable Machines), and by 1438982 (XPS: FULL: FP: Collaborative Research: Model-based, Event Driven Scalable Programming for the Mobile Cloud).

I would like to thank my thesis advisor Prof. Daniel Jackson for his expert guidance and all the help I received from him. Everything I learned from him, ranging from formal software analysis and software design to writing papers and making beautiful presentations, has been extremely useful and incredibly gratifying. I am also grateful to my thesis committee members, Prof. Armando Solar-Lezama and Prof. Robert Miller, whose suggestions improved this thesis significantly.

I owe a big thanks to all my colleagues with whom I published together: Sasa Misailovic, Sarfraz Khurshid, Darko Marinov, Nemanja Petrovic, Derek Rayside, Zev Benjamin, Joseph P. Near, Rishabh Singh, Daniel Jackson, Kuat Yessenov, Greg Dennis, Hillel Kugler, Eunsuk Kang, Rustan Leino, Milos Gligoric, and Ido Efrati. I really appreciate the opportunity to work with and learn from each one of them.

The content of this thesis would certainly be very different had I not been influenced by a number of people I met and worked with throughout my academic career. Prof. Dragan Milicev showed me the power of UML models in software engineering back when I was an undergraduate student; the part of this thesis where I explore model-based web programming is strongly influenced by this experience. Prof. Darko Marinov soon thereafter introduced me to research opportunities in software engineering, guided me through my first publications, helped me apply for graduate studies, and remained involved in my academic progress ever since. After joining MIT, I learned about the Alloy language and the idea of lightweight formal analysis from Prof. Daniel Jackson; Alloy has been the central part of my research while at MIT, and the majority of this thesis is dedicated to its improvements and novel uses. In that process, Emina Torlak's help with some of the internals of the Alloy infrastructure was invaluable. Finally, Derek Rayside got me interested in the idea of executing Alloy-like specifications, he encouraged me to explore it further, which eventually led me to the broader and more general idea of declarative programming, the topic of this thesis.

I had the good fortune to collaborate with a number of smart undergraduate MIT students, who contributed to various parts of this thesis. Ido Efrati implemented the bridge between αRby and the Alloy Analyzer, Angel Yu explored the applicability of the event-driven paradigm behind SUNNY in robot programming, Jing Fan ported the original version of SUNNY from Ruby on Rails to Meteor, and Ebenezer Sefah developed a generic GUI for SUNNY programs. I am thankful to all of them.

Finally, a huge thanks to my friends and colleagues at MIT, who made my day-to-day work fun and enjoyable: Eunsuk Kang, Harshad Kasture, Ivan Kuraj, James Cowling, Jean Yang, Joeseph Near, Jonathan Edwards, Kuat Yessenov, Matt McCutchen, Michal Depa, Nadia Polikarpova, Rishabh Singh, Sachi Hemachandra, Santiago Perez De Roso, Sasa Misailovic, and Stelios Sidiroglou.

Contents

1 Introduction **17**

I Empowering the Solver **22**

2 Alloy*: General-Purpose Higher-Order Constraint Solving **25**
2.1 Example: Classical Graph Algorithms 26
2.2 Example: Policy Synthesis 30
2.3 Background and Key Ideas 30
2.3.1 Skolemization 30
2.3.2 CEGIS 32
2.3.3 CEGIS for a general purpose solver 32
2.4 Semantics 33
2.4.1 Translation of Formulas into Proc Objects 33
2.4.2 Satisfiability Solving 35
2.4.3 Treatment of Bounds 36
2.5 Implementation 38
2.6 Case Study: Program Synthesis 38
2.7 Optimizations 40
2.7.1 Quantifier Domain Constraints 40
2.7.2 Strictly First-Order Increments 41
2.8 Evaluation 42
2.8.1 Micro Benchmarks 42
2.8.2 Program Synthesis 43
2.8.3 Benefits of the Alloy* Optimizations 46
2.8.4 Distribution of Solving Time over Individual Candidates 46
2.8.5 Discussion 47
2.9 Related Work 49
2.10 Conclusion 50

3 Preventing Arithmetic Overflows in Alloy* **53**
3.1 Prototypical Overflow Anomalies 54
3.2 Motivating Example 56
3.3 Approach 58
3.3.1 User-Level Semantics 59

3.3.2 Implementation-Level Semantics 60
3.3.3 Correspondence Between the Two Semantics 65
3.3.4 The Law of the Excluded Middle 66
3.4 Implementation in Circuits 67
3.5 Evaluation 67
3.5.1 Exhaustive Testing of the New Translation Scheme 68
3.5.2 Effects on Models with Integer Arithmetic 69
3.6 Related Work 70
3.7 Conclusion 71

II Unifying Specification and Implementation Languages 73

4 αRby: An Embedding of Alloy in Ruby 75
4.1 Why an Imperative Shell Around a Modeling Language 76
4.2 Examples of Motivating Use Cases 77
4.3 αRby for Alloy Users 79
4.4 Beyond Standard Analysis 79
4.5 The αRby Language 82
4.5.1 Syntax 83
4.5.2 Semantics 84
4.5.3 Implementation Considerations 86
4.6 Discussion 89
4.7 Related Work 90
4.8 Discussion 91
4.9 Conclusion 92

III Declarative Programming for the Web 93

5 SUNNY: Model-Based Paradigm for Programming Reactive Web Applications 95
5.1 Motivation 97
5.2 Example 98
5.3 What is Different About SUNNY 100
5.3.1 The Java Approach 101
5.3.2 The Rails Approach 101
5.3.3 The Meteor Approach 102
5.4 The SUNNY Approach 103
5.4.1 Sample Execution 103
5.4.2 Domain-Specific Programming Language 106
5.4.3 Runtime Environment 107
5.4.4 Online Code Generator 108
5.4.5 Dynamic Template-Based Rendering Engine 112
5.5 Semantics 113
5.5.1 Policy Checking 115

5.5.2 Reactivity 118
5.5.3 Concurrency Model 120
5.6 Automated Reasoning and Analysis 120
5.6.1 Testing 120
5.6.2 Model Checking 122
5.6.3 Verification and Program Synthesis 122
5.7 Discussion 122
5.8 Evaluation 123
5.8.1 Gallery of *Sunny.js* Applications 123
5.8.2 Comparison with a Web Application in Meteor 130
5.8.3 Limitations 130
5.9 Related Work 132
5.9.1 Event-Driven Programming 132
5.9.2 Data-Centric Programming 133
5.9.3 Code Generation and Program Synthesis 134
5.9.4 Declarative Privacy Policies 134
5.9.5 GUI Builders 136
5.10 Conclusion 136

6 Conclusion **139**

List of Figures

1-1 Spectrum of the declarative programming space explored in this thesis . . . 21

2-1 An automatically generated instance satisfying *maxClique* 28
2-2 Alloy* GUI showing a trace of all explored candidate instances 29
2-3 Alloy* formalization: overview of the syntactic domains 34
2-4 Alloy* formalization: overview of semantic and built-in functions 34
2-5 Translation of boolean `Formula`s to `Proc`s 36
2-6 The higher-order model finding algorithm 37
2-7 Average (over 5 different edge densities) (a) solving times, and (b) number of explored candidates for the graph algorithms 42
2-8 Avgerage times over thresholds for graph algorithms 44
2-9 Comparison between Alloy* and Reference Solvers. 45
2-10 Distribution of total solving time over individual (sequentially explored) candidates for the three hardest benchmarks. Each candidate time is further split into times for each CEGIS phase 48

3-1 Overview of semantic domains, symbols, and stores to be used 63
3-2 Evaluation of arithmetic operations (**`aeval`**) 63
3-3 Evaluation of boolean formulas (**`beval`**) 63
3-4 Evaluation of integer predicates (**`ieval`**) 64

4-1 Core αRby syntax in BNF. Productions starting with: `ruby` are defined by Ruby. 83
4-2 Semantic domains. (`Expr` and `Decl` correspond directly to the Alloy AST) . 86
4-3 Semantic functions which translate grammar rules to semantic domains . . 86
4-4 Evaluation of αRby expressions (`expr` production rules) into Alloy expressions (`Expr`). 87
4-5 Evaluation of blocks and all declarations 88
4-6 Helper functions. 88

5-1 Internal architecture of SUNNY's runtime environment for concurrent processing of events and user requests. 103
5-2 Snippets of automatically generated code for the IRC example 109
5-3 Datatypes, global variables, built-in and framework-provided functions . . 114
5-4 Formalization of policy checking in SUNNY 116
5-5 Formalization of CRUD operations in SUNNY 117

5-6 Formalization of auto publishing in SUNNY 119
5-7 State diagram for the IRC example 121
5-8 Views of three different users of the same Chat application 126
5-9 Views of two different users of the same PartyPlanner application 127
5-10 Views of two different users of the same SocNet application 129

List of Tables

2.1 Performance on Synthesis Benchmarks 46
2.2 Performance of Alloy* (in seconds) with and without optimizations. 47

3.1 List of checked arithmetic tautologies 70
3.2 Analysis times of checks from the flash filesystem model [88] 70

4.1 Examples of differences in syntax between αRby and Alloy 84

List of listings

1 Automatic checking of *Turan's* theorem in Alloy* 27
2 Grade Assignment Policy in Alloy* 31
3 Prototypical overflow anomalies in the previous version of Alloy 55
4 Alloy model for bounded verification of Prim's algorithm that finds a minimum spanning tree for a weighted connected graph 57
5 A unit test for exhaustively checking overflow detection in elementary arithmetic formulas 68
6 *Hamiltonian Path* example 80
7 A declarative *Sudoku* solver using αRby with partial instances 81
8 A full implementation (excluding any GUI) of a simple public IRC application written in RED 99
9 ERB template views for the IRC example from Listing 8 104
10 Excerpt from the JavaScript translation of the domain model, which the client-side code can program against 112
11 *Sunny.js* `Chat` application code listing: data and security models 125
12 *Sunny.js* PartyPlanner application code listing: data and security models 127
13 *Sunny.js* SocNet application code listing: data and security models 129
14 Implementation of the IRC example in Meteor 131

Chapter 1

Introduction

This thesis attempts to unite and consolidate two large and often culturally disjoint programming paradigms: *declarative* and *imperative*. Declarative is taken to mean any programming environment that allows the programmer to focus on expressing *what* the program is supposed to do; this is in contrast to imperative programming, which requires the programmer to prescribe a step-by-step algorithm detailing *how* the program intention is to be achieved.

To illustrate the main difference between the two paradigms, consider the problem of computing a *maximum clique* in a graph. A purely declarative program would only state that the result is the largest set of nodes in which each pair of nodes is connected (i.e., the largest complete subgraph in the input graph); an imperative program, on the other hand, would express some kind of search algorithm that, might, e.g., explore all complete subgraphs and return the largest one. The former can succinctly be expressed using classical predicate logic and basic set algebra, which is arguably easier than writing a custom search algorithm implementing some form of backtracking or dynamic programming. More importantly, the declarative program directly corresponds to the problem statement, making it correct by construction.

Statements about program behavior, i.e., what the program is supposed to do, are often called *program specifications*. Traditionally, a specification is thought of as a rigorous mathematical characterization of a program, written alongside an imperative implementation. As such, it mainly serves as part of the program requirement documentation. In more advanced scenarios, advocated strongly by the *formal methods* community, a formal proof should additionally be constructed to verify that the implementation meets the specification (e.g., [21,45,46,122–124]).

In practice, however, software verification is not an easy task. It requires significant human effort, typically by highly trained personnel. Consequently, its use is often limited to safety critical systems of utmost importance. For that reason, *agile* software development approaches, which argue that software development should be driven by concrete tests and use cases instead of formal specifications [27,28], have been widely adopted in industry.

The idea of treating specifications as standalone programs and being able to execute them directly (without requiring a separate imperative implementation) is not a new one. Until recently, however, it has been widely assumed that any implementation would be hopelessly inefficient, and thus not feasible for practical applications. Back in 1987, Hoare

acknowledged the benefits that such technology would have, but also predicted that computers would never be powerful enough to carry out any interesting computation in this way [75], while Hayes and Jones argued in 1989 that direct execution of specifications would inevitably lead to a decrease in the expressive power of the specification language [74].

With the advent of automated constraint solving (e.g., [25,29,41,48,50,97,162]) and *lightweight formal methods* [77,80], executing specifications as standalone programs has become feasible [90,118,141,161]. Moreover, many modern programming languages (e.g., Ruby [105], Scala [132], Racket [56]) are highly extensible and support powerful mechanisms for embedding domain specific languages, so the expressive power of the specification language does not have to suffer either (provided, of course, that the underlying solver is powerful enough to support it).

In my previous work, I developed SQUANDER, a unified environment for executing declarative and imperative code [118], and also argued how such a unified environment might create a smooth continuum between the formal and the agile [136]. SQUANDER integrates first-order relational specifications into object-oriented Java programs. A declarative specification can be asserted against the program heap at runtime, at which point SQUANDER automatically translates the heap into relations, invokes a relational constraint solver, namely Kodkod [162], and updates the heap according to the solution returned by the solver (if one is found). In a related project, I developed JENNISYS [99], a program synthesis technique, which, for certain classes of programs, takes a similar declarative specification and generates an equivalent imperative program. The benefit of the synthesis approach is that no performance penalty is incurred while executing the generated program; the downside is, however, that the synthesis algorithm is computationally significantly more expensive, and in practice it is limited to only a small class of programs.

In this thesis, I make the following advances over my previous work on the topic of executable specifications:

(1) Empowering the Solver. To execute specifications seamlessly within a larger program, where imperative and declarative statements are mixed freely, it is mandatory that the declarative statements can be solved fully automatically. State-of-the-art constraint solvers, typically used for this purpose, limit their use to various forms of *first-order* logic (e.g., boolean satisfiability—SAT, satisfiability modulo theories—SMT, predicate logic over bounded relations—Kodkod, etc.). For many practical purposes, however, first-order logic is not sufficiently expressive. The aforementioned maximal clique problem, for instance, is not first-order, and thus could not be solved by SQUANDER or any other similar tool, specifically because of the limitations of the underlying constraint solver.

To address this issue, I designed and implemented **Alloy*** , the first general-purpose constraint solver capable of automatically solving formulas with *higher-order* quantification over bounded domains [116,117]. Existing solvers either do not admit such quantifiers, or fail to produce a solution in most cases. Alloy*, by contrast, is both sound and complete for the given bounds, and still efficient for many practical purposes. For example, the problem of software synthesis is higher-order. Existing synthesizers implement a custom search algorithm on top of a first-order constraint solver, whereas with Alloy* this problem becomes solvable directly (in one step) and purely declaratively, from a single higher-order formula. Furthermore, Alloy* scales better than JENNISYS and the standard reference syn-

thesizers[1], and is competitive with highly optimized, purpose-built synthesizers. Alloy* is built on top of Alloy [78], retaining its syntax and its input language (many-sorted predicate logic with relational algebra), and changes its semantics only by expanding the set of specifications that can be analyzed, namely to include the higher-order ones.

This thesis also addresses another deficiency, specific to bounded constraint solvers like Kodkod and Alloy, relevant for the purpose of executing specifications—that of *arithmetic overflows*. In a bounded setting, integers must also be bounded, causing all arithmetic functions to become partial. Partial functions in logic are known to be a hard problem, so a common strategy is to regard the smallest negative integer as the successor of the largest positive integer and give the arithmetic functions wraparound semantics (for example, 3 + 2 becomes -3 if integers are bounded to {-4, -3, -2, -1, 0, 1, 2, 3}). This approach, unfortunately, results in bogus solutions (when solving declarative statements) that would not hold in the unbounded context. Alloy* retains the bounded nature of integers in Alloy, but mitigates this problem (completely separately from its solution to higher-order quantifiers) by adjusting the translation from relational to propositional logic in a clever way that ensures that no solution with overflows is ever returned by the solver [113, 114].

(2) Unifying Specification and Implementation Languages. For writing specification statements, SQUANDER supports JFSL [170], an Alloy-like relational language, that supports both relational algebra and standard Java expressions. These properties make it easy to succinctly write complex relational properties in terms of a program's data structures and reachable objects on the heap. Unfortunately, however, the JFSL specification statements in SQUANDER are written as plain strings, embedded in standard Java annotations. This is suboptimal, because none of the JFSL language constructs are first-class, making it is impossible to manipulate and process them programmatically. Some other approaches (e.g., [90, 161]), in contrast, choose to use the unmodified (imperative) host language also for writing specification statements; this mitigates the first-class problem, but leads to a decrease in the expressive power of the specification language.

As part of this thesis, I designed and implemented α**Rby** [112], a single unified programming language for writing declarative specification statements and imperative object-oriented code. αRby is implemented as a deep embedding of the Alloy modeling language (backed by an automated constraint solver, namely Alloy*) in Ruby. This approach aims to bring these two distinct paradigms (imperative and declarative) together in a novel way. Having the other paradigm available within the same language is beneficial to both the modeling community of Alloy users and the object-oriented community of Ruby programmers: the Alloy users gain a full-fledged imperative shell around the modeling language (allowing them to programmatically construct their models, parameterize them, script certain tasks around their modeling workflow, etc.), while the Ruby community can still take the full advantage of executable specifications.

Additionally, I explore how the general idea of declarative programming ("saying what instead of how") can be applied to an entirely different domain, where no knowledge of formal logic is required to write specifications, and no expensive runtime calls to a constraint solver are necessary:

[1]Taken from the official SyGuS [17] synthesis competition initiative.

(3) Declarative Programming for the Web. Web application development poses a unique set of challenges. Applications are increasingly distributed and event-driven, but concurrent and distributed programming (including distributed data management) is still difficult and a significant source of software bugs. With the adoption of ubiquitous (albeit often unnecessary) online sharing, security and privacy are becoming major concerns. Implementing privacy policies, however, is still, by and large, a cross-cutting concern, requiring the implementation to be scattered across the codebase, making it fragile and unreliable. Finally, for the most part, modern three-tier web applications still require different languages and technologies to be used for each tier (e.g., relational database and SQL for the data tier, Ruby, Java, Python or some other programming language for the server-side logic tier, and Javascript for the presentation and client-side logic tier). This not only poses a cognitive burden on the developer, but more importantly, causes undesirable redundancies across the tiers (e.g., a single data model typically has three different representations: in the database, in the server-side programming language, and in Javascript for the client side).

Despite the technological complexities of building interactive web applications, most such applications are conceptually very simple. Imagine a straightforward chat application, where all rooms are public, users can freely create and join chat rooms, and once joined, they can post messages, which are then automatically shown to other participating users. Although the functionality (data model + business logic) of such an app is arguably trivial, implementing it using a standard model-view-controller web framework (e.g., Ruby on Rails, or Django), even without any scalability considerations, proves to be a tedious, time-consuming, and error-prone task, exactly because of all the accidental complexities stated above.

Applying the idea of declarative programming (saying what instead of how), I designed **SUNNY** [115], a model-based, event-driven, policy-agnostic paradigm for developing reactive web applications. SUNNY imposes a clear separation between four main (often cross-cutting) concerns of web applications: (1) data model, (2) reactive GUI, (3) event model, and (4) security model. The programmer specifies each model separately and independently from each other, and the SUNNY runtime is in charge of different technologies and ensuring that events are executed atomically, that there are no data races, that all security policies are applied consistently throughout the application, that the data model is automatically persisted (e.g., by translating everything behind the scene to a relational database), that the data is automatically replicated and appropriately sent to all clients (without violating any "read" security policies), etc. As a result, SUNNY allows a programmer to represent a distributed application if it were a simple sequential program, with atomic actions updating a single, shared global state.

I implemented SUNNY both on top of a traditional model-view-controller web framework, and a lightweight pure JavaScript web platform. The purpose of the former is to demonstrate how the SUNNY concepts can be achieved across the full web stack; I used Ruby on Rails [6] and αRby as the input modeling language. The latter implementation, made on top of Meteor [4], takes advantage of "thick client" and "client-side rendering" technologies to improve scalability and responsiveness.

Figure 1-1 graphically depicts the portion of the declarative programming spectrum explored in this thesis. The main contributions include:

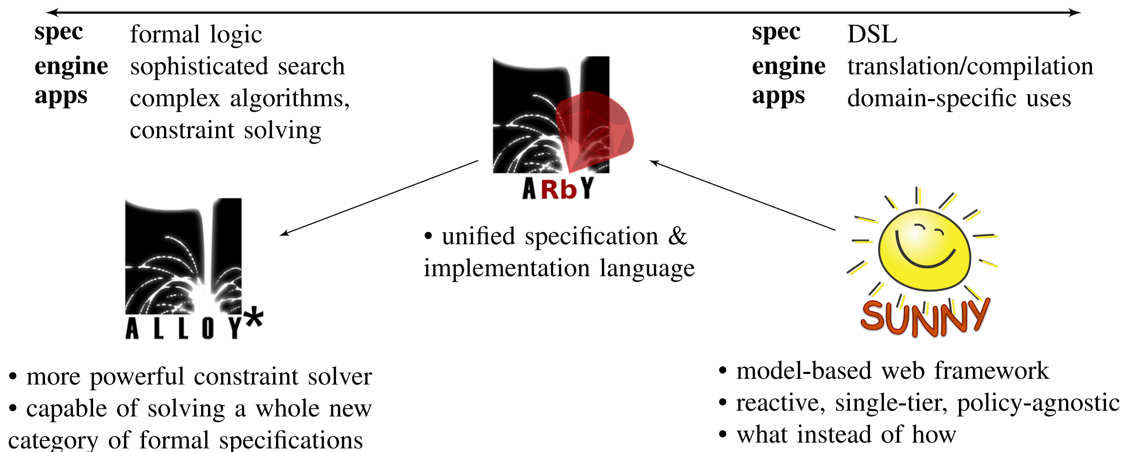


Figure 1-1: Spectrum of the declarative programming space explored in this thesis

- Alloy*: (1) a framework for extending a first-order solver to the higher-order case, including an implementation of the first general-purpose constraint solver capable of automatically solving formulas with higher-order quantification over bounded domains, and (2) a formalization of a novel treatment of partial arithmetic functions in logic, and its application in Alloy*;

- αRby: a new kind of combination of a declarative and an imperative language, justified by a collection of examples demonstrating how both imperative and declarative paradigms can benefit from having the other available in the same language;

- SUNNY: a novel declarative programming paradigm for reactive web applications, imposing a clear separation between main (often cross-cutting) concerns and allowing a simple single-tier programming model for developing distributed systems.

The technical content of this thesis is split into three chapters, one for each of the three main contributions. Part I is dedicated to the improvements to the constraint solving technologies, Part II is about the language integration challenges, and Part III focuses on declarative programming in the domain of web applications.

Part I

Empowering the Solver

The last decade has seen a dramatic growth in the use of constraint solvers as a computational mechanism, not only for analysis of software, but also at runtime. Solvers are available for a variety of logics but are generally restricted to first-order formulas. Some tasks, however, most notably those involving synthesis, are inherently higher order; these are typically handled by embedding a first-order solver (such as a SAT or SMT solver) in a domain-specific algorithm.

Using strategies similar to those used in such algorithms, Chapter 2 of this thesis shows how to extend a first-order solver (in this case Kodkod, a model finder for relational logic used as the engine of the Alloy Analyzer) so that it can handle quantifications over higher-order structures. The resulting solver is sufficiently general that it can be applied to a range of problems; it is higher order, so that it can be applied directly, without embedding in another algorithm; and it performs well enough to be competitive with specialized tools. Just as the identification of first-order solvers as reusable backends advanced the performance of specialized tools and simplified their architecture, factoring out higher-order solvers may bring similar benefits to a new class of tools.

Chapter 3 of this thesis proposes a solution to another outstanding problem in the domain of bounded constraint solving. In a bounded analysis, arithmetic operators become partial, and a different semantics becomes necessary. One approach, mimicking programming languages, is for overflow to result in wrap-around. Although easy to implement, wrap-around produces unexpected counterexamples that do not correspond to cases that would arise in the unbounded setting. This thesis describes a new approach, implemented in the latest version of the Alloy Analyzer, in which instances that would involve overflow are suppressed, and consequently, spurious counterexamples are eliminated. The key idea is to interpret quantifiers so that bound variables range only over values that do not cause overflow.

Both of the two solver-level improvements presented in this chapter have immediate benefits in the domain of executable specifications, thus contributing to the overall goal of this thesis: advancing the state of the art of declarative programming in general.

Chapter 2

Alloy*: General-Purpose Higher-Order Constraint Solving

As constraint solvers become more capable, they are increasingly being applied to problems previously regarded as intractable. Program synthesis, for example, requires the solver to find a single program that computes the correct output for all possible inputs. This "$\exists\forall$" quantifier pattern is a particularly difficult instance of higher-order quantification, and no existing general-purpose constraint solver can reliably provide solutions for problems of this form.

Instead, tools that rely on higher-order quantification use ad hoc methods to adapt existing solvers to the problem. A popular technique for the program synthesis problem is called CEGIS (counterexample guided inductive synthesis) [150], and involves using a first-order solver in a loop: first, to find a candidate program, and second, to verify that it satisfies the specification for all inputs. If the verification step fails, the resulting counterexample is transformed into a constraint that is used in generating the next candidate.

This chapter presents Alloy*, a general-purpose, higher-order, bounded constraint solver based on the Alloy Analyzer [78]. Alloy is a specification language combining first-order logic with relational algebra; the Alloy Analyzer performs bounded analysis of Alloy specifications. Alloy* admits higher-order quantifier patterns, and uses a general implementation of the CEGIS loop to perform bounded analysis. It retains the syntax of Alloy, and changes the semantics only by expanding the set of specifications that can be analyzed, making it easy for existing Alloy users to adopt.

Alloy* handles higher-order quantifiers in a generic and model-agnostic way, meaning that it allows higher-order quantifiers to appear anywhere where allowed by the Alloy syntax, and does not require any special idiom to be followed. Alloy* first creates a *solving strategy* by decomposing an arbitrary formula (possibly containing nested higher-order quantifiers) into a tree of subformulas and assigning a decision procedure to each of them. Each such tree is either (1) a higher-order "$\exists\forall$" pattern, (2) a disjunction where at least one disjunct is higher-order, or (3) a first-order formula. To solve the "$\exists\forall$" nodes, Alloy* applies CEGIS; for the disjunction leaves, Alloy* solves each disjunct separately; and for first-order formulas, Alloy* uses Kodkod [163].

To solve "$\exists\forall$" constraints, Alloy* first finds a *candidate solution* by changing the universal quantifier into an existential and solving the resulting first-order formula. Then,

it *verifies* that candidate solution by attempting to falsify the original universal formula (again, a first-order problem); if verification fails, Alloy* adds the resulting counterexample as a constraint to guide the search for the next candidate, and begins again. When verification succeeds, the candidate represents a solution to the higher-order quantification, and can be returned to the user.

To our knowledge, Alloy* is the first general-purpose constraint solver capable of solving formulas with higher-order quantification. Existing solvers either do not admit such quantifiers, or fail to produce a solution in most cases. Alloy*, by contrast, is both sound and complete for the given bounds, and still efficient for many practical purposes.

We have evaluated Alloy* on a variety of case studies taken from the work of other researchers. In the first, we used Alloy* to solve classical higher-order NP-complete graph problems like `max-clique`, and found it to scale well for uses in modeling, bounded verification, and fast prototyping. In the second, we encoded all of the SyGuS [17] program synthesis benchmarks that do not require bit vectors, and found that, while state-of-the-art purpose-built synthesizers are typically faster, Alloy* beats all the reference synthesizers provided by the competition organizers. In the third, we encoded security properties used in Margrave, and were able to synthesize new security policies consistent with the properties.

Alloy* retains Alloy's syntax, and changes its semantics only by expanding the set of specifications that can be analyzed. Our improved Alloy Analyzer is identical to the original besides the addition of higher-order solving, and will be easy for existing Alloy users to adopt. It is available [1] under the terms of the GPLv3, and we hope that Alloy users will find the additional expressive power useful.

The main contributions are as follows:

- A framework for extending a first-order solver to the higher-order case, consisting of the design of datatypes and a general algorithm comprising syntactic transformations (skolemization, conversion to negation normal form, etc.) and an incremental solving strategy;
- A collection of case study applications demonstrating the feasibility of the approach in different domains (including synthesis of code, execution and bounded verification of higher-order NP-hard algorithms), and showing encouraging performance on standard benchmarks;
- The recognition of higher-order solving as the essence of a range of computational tasks;
- The identification of the key properties of a first-order solver that are required for such a framework.

2.1 Example: Classical Graph Algorithms

Classical graph algorithms have become prototypical Alloy examples, showcasing both the expressiveness of the Alloy language and the power of the Alloy Analyzer. Many complex problems can be specified declaratively in only a few lines of Alloy, and then in a matter of

```
some sig Node  {val: one Int}
// between every two nodes there is an edge
pred clique[edges: Node -> Node, clq: set Node] {
  all disj n1, n2: clq | n1 -> n2 in edges }
// no other clique with more nodes
pred maxClique[edges: Node -> Node, clq: set Node] {
  clique[edges, clq]
  no clq2: set Node | clq2 != clq and clique[edges,clq2] and #clq2 > #clq }
// symmetric and irreflexive
pred edgeProps[edges: Node -> Node] {
  (~edges in edges) and (no edges & iden) }
// max number of edges in a (k+1)-free graph with n nodes is (k-1)n^2/2k
check Turan {
  all edges: Node -> Node | edgeProps[edges] implies
    some mClq: set Node {
      maxClique[edges, mClq]
      let n = #Node, k = #mClq, e = (#edges).div[2] |
        e <= k.minus[1].mul[n].mul[n].div[2].div[k] }
} for 7 but 0..294 Int
```

Listing 1: Automatic checking of *Turan's* theorem in Alloy*

seconds fully automatically animated (for graphs of small size) by the Alloy Analyzer. This ability to succinctly specify and quickly solve problems like these—algorithms that would be difficult and time consuming to implement imperatively using traditional programming languages—has found its use in many applications, including program verification [42,60], software testing [104, 140], fast prototyping by means of executing specifications [118, 141], teaching [54], etc.

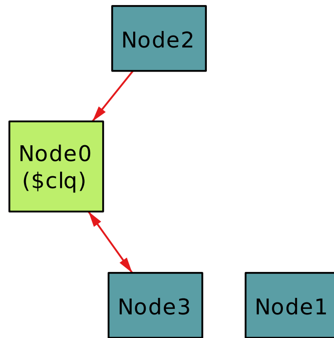
For a whole category of interesting problems, however, the current Alloy engine is not powerful enough. Those are the higher-order problems, for which the specification has to quantify over relations rather than scalars. Many well-known graph algorithms fall into this category, including finding maximal cliques, max cuts, minimum vertex covers, and various coloring problems. In this section, we show such graph algorithms can be specified and analyzed using the new engine implemented in Alloy*.

Suppose we want to check Turán's theorem, one of the fundamental results in graph theory [14]. Turán's theorem states that a $(k + 1)$-free graph with n nodes can maximally have $\frac{(k-1)n^2}{2k}$ edges. A graph is $(k+1)$-free if it contains no clique with $k+1$ nodes (a clique is a subset of nodes in which every two nodes are connected by an edge).

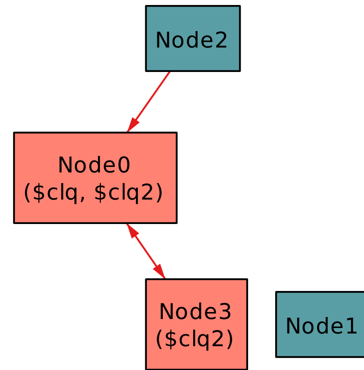
Listing 1 shows how Turán's theorem might be formally specified in Alloy. A signature is defined to represent the nodes of the graph (line 1). Next, the clique property is embodied in a predicate (lines 3–4): for a given `edges` relation and a set of nodes `clq`, every two different nodes in `clq` are connected by an edge; the `maxClique` predicate (lines 6–8) additionally asserts that no other clique contains more nodes.

Having defined maximal cliques in Alloy, we can proceed to formalize Turán's theorem. The `Turan` command (lines 13–19) asserts that for all possible `edge` relations that are

(a) A *maxClique* candidate



(b) A counterexample for (a)



(c) Final *maxClique* instance

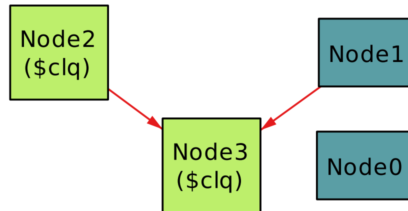


Figure 2-1: An automatically generated instance satisfying *maxClique*

symmetric and irreflexive (line 14), if the max-clique in that graph has k nodes (`k=#mClq`), the number of selected edges (`e=(#edges).div[2]`) must be at most $\frac{(k-1)n^2}{2k}$. (The number of tuples in `edges` is divided by 2 because the graph in the setup of the theorem in undirected.)

Running the `Turan` command was previously not possible. Although the specification, as given in Listing 1, is allowed by the Alloy language, trying to execute it causes the Analyzer to immediately return an error: "Analysis cannot be performed since it requires higher-order quantification that could not be skolemized". In Alloy*, in contrast, this check can be automatically performed to confirm that indeed no counterexample can be found within the specified scope. The scope we used (7 nodes, ints from 0 to 294) allows for all possible undirected graphs with up to 7 nodes. The upper bound for ints was chosen to ensure that the formula for computing the maximal number of edges ($\frac{(k-1)n^2}{2k}$) never overflows for $n \leq 7$ (which implies $k \leq 7$). The check completes in about 45 seconds.

To explain the analysis problems that higher-order quantifiers pose to the standard Alloy Analyzer, and how those problems are tackled in Alloy*, we look at a simpler task: finding an instance of a graph with a subgraph satisfying the `maxClique` predicate. The problematic quantifier in this case is the inner "`no clq2: set Node |` ..." (line 8) constraint, which requires checking that for all possible subsets of `Node`, not one of them is a clique with more nodes than the given set `clq`. A direct translation into the current SAT-based backend would require the Analyzer to explicitly, and upfront, enumerate all possible subsets of `Node`—which would be prohibitively expensive. Instead, Alloy* implements the CEGIS approach:

1. First, it finds a candidate instance, by searching for a clique `clq` and *only one* set of

nodes `clq2` that *is not* a clique larger than `clq`. A possible first candidate is given in Figure 2-1(a) (with the clique nodes are highlighted in green). At this point `clq2` could have been anything that is either not a clique or not larger than `clq`.

2. Next, Alloy* attempts to falsify the previous candidate by finding, again, *only one* set of nodes `clq2`, but this time such that `clq2` *is* a clique larger than `clq`, for the exact (concrete) graph found in the previous step. In this case, it finds one such *counterexample* clique (red nodes in Figure 2-1(b)) refuting the proposition that `clq` from the first step is a maximal clique.

3. Alloy* continues by trying to find another candidate clique, encoding the previous counterexample to prune the remainder of the search space (as explained in detail in Sections 2.3 and 2.4). After several iterations, it finds the candidate in Figure 2-1(c) which cannot be refuted, so it returns that candidate as a satisfying solution.

Once written, the `maxClique` predicate (despite containing a higher-order quantification) can be used in other parts of the model, like any other predicate, just as we used it to formulate and check Turán's theorem. In fact, the `turan` check contains another higher-order quantifier, so the analysis ends up spawning two nested CEGIS loops and exhaustively iterating over them; every candidate instance and counterexample generated in the process can be opened and inspected in the Alloy visualizer (as depicted in Figure 2-2).

```
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
              |- [Some4All] started (formula, bounds)
              |- [Some4All]  candidate found (candidate)
              |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
              |- [Some4All] started (formula, bounds)
              |- [Some4All]  candidate found (candidate)
              |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
              |- [Some4All] started (formula, bounds)
              |- [Some4All]  candidate found (candidate)
              |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) counterexample
              |- [Some4All] started (formula, bounds)
              |- [Some4All]  candidate found (candidate)
              |- [Some4All]   verifying candidate (condition, pi) success (#cand = 1)
[Some4All]   searching for next candidate  (increment)
[Some4All]  candidate found (candidate)
[Some4All]   verifying candidate (condition, pi) success (#cand = 12)
              |- [Some4All] started (formula, bounds)
Instance found. Predicate is consistent. 2311ms.
```

Figure 2-2: Alloy* GUI showing a trace of all explored candidate instances

In Section 2.8.1 we run Alloy* on concrete graph instances to compute `max clique`, `max cut`, `max independent set`, and `min vertex cover` on graphs with up to 50 nodes.

2.2 Example: Policy Synthesis

Policy design and analysis is an active area of research. A number of existing tools [58, 76, 131, 145] use a declarative language to specify policies, and a constraint-based analysis to verify them against a high-level property. In this section, we demonstrate how Alloy* can be used to automatically *synthesize* policies satisfying given properties.

Listing 2 shows an Alloy model that describes the problem of grade assignment at a university, based on the running example from [58]. A policy specification contains three basic concepts: *roles*, *actions*, and *resources*. A system consists of a set of users, each having one or more roles and performing some actions on a set of resources. A *policy* (`acl`) is a set of tuples from `Role` to `Action` to `Resource`, describing a set of allowed actions. For example, a policy containing only a single tuple `Faculty->Assign->ExtGrade` means that a user may assign an external grade only if it has the `Faculty` role.

There are two desirable properties over this system: (1) students should not be able to assign external grades, and (2) no user should be able to both assign and receive external grades. A policy is considered *valid* if and only if, when *quantified over every* possible combination of user roles and behaviors, it ensures that the properties hold. This higher-order property is encoded in the `valid` predicate.

Running Alloy* to search for an instance satisfying the `valid` predicate completes in about 0.5 seconds, and retuns an *empty* policy, which is technically valid but not very useful (since it allows no actions to be performed by anyone!). Fortunately, we can leverage the higher-order feature of Alloy* to synthesize more interesting policies. For example, 3 additional lines of Alloy are enough to describe the *most permissive* policy as a policy that is valid such that no other valid policy has more tuples in it (lines 37–41). It takes about 3.5 seconds to generate one such policy:

```
{Faculty,Receive,ExtGrade}, {Faculty,Assign,Resource},
{Student,Receive,Resource}, {Student,Assign,IntGrade},
     {TA,Receive,Resource}, {TA,Assign,IntGrade}
```

This policy provides a starting point for further exploration of the policy space. The designer may decide, for example, that students should not be able to assign `IntGrade`, add another property, and then repeat the synthesis process.

2.3 Background and Key Ideas

2.3.1 Skolemization

Many first-order constraint solvers allow some form of higher-order quantifiers to appear at the language level. Part of the reason for this is that, in certain cases, quantifiers can be eliminated in a preprocessing step called *skolemization*. In a model finding setting, every top-level existential quantifier is eliminated by (1) introducing a *skolem constant* for the quantification variable, and (2) replacing every occurrence of that variable with the newly created skolem constant. For example, solving `some s: set univ | #s > 2`, which is higher-order, is equivalent to solving `$s in univ && #$s > 2`, which is first-order and

```
/* Basic signatures */
abstract sig Resource, Action, Role {}
sig User {}

/* 'performs' describes the behavior of users */
pred enforce[acl:      Role->Action->Resource,
             roles:    User->Role,
             performs: User->Action->Resource] {
  all u: User, a: Action, r: Resource |
    /* 'u' can perform 'a' on 'r' only if allowed by 'acl' */
    u->a->r in performs => (some ro: u.roles | ro->a->r in acl)
}
/* Domain-specific concepts   */
one sig Faculty,  Student, TA extends Role {}
one sig IntGrade, ExtGrade    extends Resource {}
one sig Assign,   Receive     extends Action {}
/* Properties   */
pred prop1[roles : User->Role, performs : User->Action->Resource] {
  /* no student can assign external grade */
  no u: User | u.roles = Student and Assign->ExtGrade in u.performs
}
pred prop2[roles : User->Role, performs : User->Action->Resource] {
  /* no user can both receive and assign external grades */
  no u: User | Assign + Receive in u.performs.ExtGrade
}
/* Assumption: no user can both be a faculty and a student/TA */
pred noDualRoles[roles : User->Role] {
  no u: User | Faculty in u.roles and some (Student + TA) & u.roles
}
/* 'acl' satisfies properties over every user role and behavior */
pred valid[acl: Role->Action->Resource] {
  all roles: User->Role, performs : User->Action->Resource |
    (enforce[acl, roles, performs] and noDualRoles[roles]) implies
      (prop1[roles, performs] and prop2[roles, performs])
}
/* 'acl' allows the most number of actions while being valid */
pred mostPermissive[acl: Role->Action->Resource] {
  valid[acl]
  no acl': Role->Action->Resource |
    acl != acl' and valid[acl'] and #acl' > #acl
}
```

Listing 2: Grade Assignment Policy in Alloy*

thus solvable by general purpose constraint solvers. (Throughout, following the Alloy convention, skolem constants will be prefixed with a dollar sign.)

2.3.2 CEGIS

CounterExample-Guided Inductive Synthesis [150] is an approach for solving higher-order synthesis problems, which is extended in Alloy* to the general problem of solving higher-order formulas. As briefly mentioned before, the CEGIS strategy applies only to formulas in the form $\exists p \forall e \bullet s(p, e)$ and prescribes the following three steps:

(1) search: attempt to find a candidate value for p by solving $\exists p \exists e \bullet s(p, e)$—a first-order problem;

(2) verification: if a candidate $\$p$ is found, try to verify it by checking if it holds for all possible bindings for e. The *verification condition*, thus, becomes $\forall e \bullet s(\$p, e)$. This check is done by refutation, i.e., by satisfying the negation of the verification condition; pushing the negation through yields $\exists e \bullet \neg s(\$p, e)$, which, again, is first-order.

(3) induction: if the candidate is verified, a solution is found and the algorithm terminates. Otherwise a concrete counterexample $\$e_{cex}$ is found. The search continues by searching for another candidate which must also satisfy the counterexample, that is, solving $\exists p \exists e \bullet s(p, e) \wedge s(p, \$e_{cex})$. This strategy in particular tends to be very effective at reducing the search space and improving the overall scalability.

2.3.3 CEGIS for a general purpose solver

Existing CEGIS-based synthesis tools implement this strategy internally, optimizing for the target domain of synthesis problems. The key insight is that the CEGIS algorithm can be implemented, generically and efficiently, inside a general purpose constraint solver. For an efficient implementation, it is important that such a solver supports the following:

- *Partial Instances*. The verification condition must be solved against the previously discovered candidate; explicitly designating that candidate as a "partial instance", i.e., a part of the solution known upfront, is significantly more efficient than encoding it with constraints [163].

- *Incremental solving*. Except for one additional constraint, the induction step solves exactly the same formula as the search step. Many modern SAT solvers already allow new constraints to be added to already solved propositional formulas, making subsequent runs more efficient (because all previously learned clauses are readily reusable).

- *Atoms as expressions*. The induction step needs to be able to convert a concrete counterexample (given in terms of concrete atoms, i.e., values for each variable) to a formula to be added to the candidate search condition. All atoms, therefore, must be convertible to expressions. This is trivial for SAT solvers, but requires extra functionality for solvers offering a richer input language.

- *Skolemization*. Skolemizing higher-order existential quantifiers is necessary for all three CEGIS steps.

This approach is formalized in Section 2.4, assuming availability of a first-order constraint solver offering all the features above. In Section 2.5 an implementation is presented

as an extension to Kodkod [160], a first-order relational constraint solver already equipped with most of the required features.

2.4 Semantics

We give the semantics of our decision procedure for bounded higher-order logic in two steps. First, we formalize the translation of a boolean formula into a `Proc` datatype instance (corresponding to an appropriate *solving procedure*); next we formalize the semantics of `Proc` satisfiability solving.

Figure 2-3 gives an overview of all syntactic domains used throughout this section. We assume the datatypes in Figure 2-3(b) are provided by the solver; on top of these basic datatypes, Alloy* defines the following additional datatypes:

- `FOL`—a wrapper for a first-order formula
- `OR`—a compound type representing a disjunction of `Procs`
- `∃∀`—a compound type representing a conjunction of a first-order formula and a number of higher-order universal quantifiers (each enclosed in a `QP` datatype). The intention of the `QP` datatype is to hold the original formula, and a translation of the same formula but quantified existentially (used later to find candidate solutions).

Figure 2-4(a) lists all the semantic functions defined in this chapter. The main two are translation of formulas into `Procs` ($\mathcal{T}$, defined in Figure 2-5) and satisfiability solving ($\mathcal{S}$, defined in Figure 2-6). Relevant functions assumed to be either exported by the solver or provided by the host programming language are listed in Figures 2-4(b) and 2-4(c), respectively.

For simplicity of exposition, we decided to exclude the treatment of bounds from our formalization, as it tends to be mostly straightforward; we will, however, come back to this point and accurately describe how the bounds are constructed before every solver invocation.

Syntax notes. A language reminiscent of F# is used. In a nutshell, it is a functional language with algebraic datatypes, imperative loops, and mutable fields. The "." syntax is used to refer to field values of datatype instances. If the left-hand side in such constructs resolves to a list, we assume the operation is mapped over the entire list (e.g., $ea.qps.forAll$, is equivalent to $map\ \lambda_q \cdot q.forAll,\ \ ea.qps$). In function signatures, → is used to delimit individual argument and return types (e.g., in f : `Int` → `String` → `Int`, f is a function that takes an integer and a string and returns an integer), × is used to delimit column types in a tuple type (e.g., `String` × `Int` is a tuple type of string-integer pairs), and mutations are denoted using the ↤ symbol (e.g., $o.f \leftarrow 3$).

2.4.1 Translation of Formulas into Proc Objects

The top-level translation function ($\mathcal{T}$, Figure 2-5, line 1) ensures that the formula is converted to negation normal form (NNF), and that all top-level existential quantifiers are

(a) Alloy* syntactic domains

type QP = {*forAll*: Quant, *pExists*: Proc}
type Proc = FOL(*form*: Formula) | OR(*disjs*: Proc **list**) | ∃∀(*conj*: FOL, *qps*: QP **list**)

(b) Solver data types

type Mult = ONE | SET
type Decl = {*mult*: Mult, *var*: Expr}
type QuantOp = $\forall$ | $\exists$
type BinOp = $\land$ | $\lor$ | $\iff$ | $\implies$
type Formula = Quant(*op*: QuantOp, *decl*: Decl, *body*: Formula)
| BinForm(*op*: BinOp, *lhs*: Formula, *rhs*: Formula)
| NotForm(*form*: Formula)
type Expr = ... *// relational expressions, irrelevant here*
type Instance = {...} *// holds a concrete solution*

Figure 2-3: Alloy* formalization: overview of the syntactic domains

(a) Alloy* semantic functions

$\mathcal{T}$:	Formula → Proc	top-level formula translation
$\mathcal{S}$:	Proc → Instance	Proc evaluation (solving)
τ :	Formula → Proc	intermediate formula translation
$\curlywedge$:	Proc → Proc → Proc	Proc composition: conjunction
$\curlyvee$:	Proc → Proc → Proc	Proc composition: disjunction

(b) Functions exported by first-order sover

solve :	Formula → Instance **option**	first-order solver
eval :	Instance → Expr → Value	evaluator
replace :	Formula → Expr → Value → Formula	replacer
nnf :	Formula → Formula	NNF conversion
skolemize :	Formula → Formula	skolemization
$\land$, $\lor$:	Formula → Formula → Formula	conjunction, disjunction
TRUE, FALSE :	Formula	true and false constant formulas

(c) Built-in functions

fold :	(A → E → A) → A → E **list** → A	functional fold
reduce :	(A → E → A) → E **list** → A	fold w/o init value
map :	(E → T) → E **list** → T **list**	functional map
length :	E **list** → int	list length
hd :	E **list** → E	list head
tl :	E **list** → E **list**	list tail
+ :	E **list** → E **list** → E **list**	list concatenation
× :	E **list** → E **list** → E **list**	list cross product
fail :	String → void	runtime error

Figure 2-4: Alloy* formalization: overview of semantic and built-in functions

subsequently skolemized away, before the formula is passed to the τ function. Conversion to NNF pushes the quantifiers towards the roots of the formula, while skolemization eliminates top-level existential quantifiers (including the higher-order ones). Alloy* applies these techniques aggressively to achieve completeness in handling arbitrary formulas.

Translating a binary formula (which is either a conjunction or disjunction, since it is in NNF) involves translating both left-hand and right-hand sides and composing the resulting `Procs` using the corresponding composition operator (lines 2–4). A disjunction demands that both sides be skolemized again (thus the use of $\mathcal{T}$ instead of τ), since they were surely unreachable by any previous skolemization attempts. This ensures that any higher-order quantifiers found in a clause of a disjunction will eventually either be skolemized or converted to an `∃∀ Proc`.

A first-order universal quantifier (determined by $d.\mathit{mult}$ being equal to `ONE`) whose body is also first-order (line 6) is simply enclosed in a `FOL Proc` (line 7). Otherwise, an `∃∀ Proc` is returned, wrapping both the original formula ($\forall d \mid f$) and the translation of its existential counterpart ($p = \mathcal{T}[\![\exists d \mid f]\!]$, used later to find candidate solutions).

In all other cases, the formula is wrapped in `FOL` (line 10).

Composition of `Procs` is straightforward for the most part, directly following the distributivity laws of conjunction over disjunction and vice versa. The common goal in all the cases in lines 11–26 is to reduce the number of `Proc` nodes. For example, a conjunction of two `∃∀` nodes can be merged into a single `∃∀` node (line 17), as can a conjunction of a `FOL` and an `∃∀` node (line 14). With disjunction, however, we need to be more careful. Remember that before applying the $\vee$ operator skolemization had to be performed on both sides (line 2); since skolemization through disjunction is not sound, it would be wrong, for example, to try and recombine two `FOL Procs` into a single `FOL` (line 20). Instead, a safe optimization (which we implemented in Alloy*) would be to modify line 2 to first check if $f_1 \vee f_2$ is first-order as a whole, and if so return `FOL`($f_1 \vee f_2$).

2.4.2 Satisfiability Solving

The procedure for satisfiability solving is given in Figure 2-6.

A first-order formula (enclosed in `FOL`) is given to the solver to be solved directly, in one step (line 29).

An `OR Proc` is solved by iteratively solving its disjuncts (lines 30–34). An instance is returned as soon as one is found; otherwise, `None` is returned.

The procedure for the `∃∀ Procs` implements the CEGIS loop (lines 35–46), following the algorithm in Section 2.3. The candidate condition is a conjunction of the first-order $p.\mathit{conj}$ `Proc` and all the existential `Procs` from $p.\mathit{qps}.\mathit{pExists}$ (line 35); the verification condition is a conjunction of all original universal quantifiers within this `∃∀` (line 39). Encoding the counterexample back into the search formula boils down to obtaining a concrete value that each quantification variable has in that counterexample (by calling the *eval* function exported by the solver) and embedding that value directly in the body of the corresponding quantifier (lines 42-45).

```
𝒯 : Formula → Proc
1.  𝒯⟦f⟧          ≡ τ⟦skolemize nnf f⟧

τ : Formula → Proc
2.  τ⟦f1∨f2⟧      ≡ 𝒯⟦f1⟧ ∨ 𝒯⟦f2⟧
3.  τ⟦f1∧f2⟧      ≡ τ⟦f1⟧ ∧ τ⟦f2⟧
4.  τ⟦∃d | f⟧     ≡ fail "can't happen"
5.  τ⟦∀d | f⟧     ≡ let p = 𝒯⟦∃d | f⟧ in
6.                    if d.mult is ONE && p is FOL then
7.                      FOL(∀d | f)
8.                    else
9.                      ∃∀(FOL(TRUE), [QP(∀d | f, p)])
10. τ⟦f⟧          ≡ FOL(f)

∧ : Proc → Proc → Proc
11. p1 ∧ p2       ≡ match p1, p2 with
12.                 | FOL, FOL → FOL(p1.form ∧ p2.form)
13.                 | FOL, OR  → OR(map(λp · p1 ∧ p,  p2.disjs))
14.                 | FOL, ∃∀  → ∃∀(p1∧p2.conj,  p2.qps)
15.                 | OR , OR  → OR(map(λp,q · p ∧ q,  p1.disjs × p2.disjs))
16.                 | OR , ∃∀  → OR(map(λp · p ∧ p2,  p1.disjs))
17.                 | ∃∀ , ∃∀  → ∃∀(p1.conj ∧ p2.conj,  p1.qps + p2.qps)
18.                 | _  , _   → p2 ∧ p1

∨ : Proc → Proc → Proc
19. p1 ∨ p2       ≡ match p1, p2 with
20.                 | FOL, FOL → OR([p1, p2]) //wrong:  FOL(p1.form ∨ p2.form)
21.                 | FOL, OR  → OR([p1] + p2.disjs)
22.                 | FOL, ∃∀  → OR([p1, p2])
23.                 | OR , OR  → OR(p1.disjs + p2.disjs)
24.                 | OR , ∃∀  → OR(p1.disjs + [p2])
25.                 | ∃∀ , ∃∀  → OR([p1, p2])
26.                 | _  , _   → p2 ∨ p1
```

Figure 2-5: Translation of boolean Formulas to Procs

2.4.3 Treatment of Bounds

Bounds are a required input of any bounded analysis; for an analysis involving structures, the bounds may include not only the cardinality of the structures, but may also indicate that a structure includes or excludes particular tuples. Such bounds serve not only to finitize the universe of discourse and the domain of each variable, but may also specify a *partial instance* that embodies information known upfront about the solution to the constraint. If supported by the solver, specifying the partial instance through bounds (as opposed to enforcing it with constraints) is an important mechanism that generally improves scalability significantly.

Although essential, the treatment of bounds in Alloy* is mostly straightforward—

```
S : Proc → Instance option
27. S⟦p⟧ ≡
28.   match p with
29.   | FOL → solve p.form
30.   | OR  → if length p.disjs = 0
31.           then None
32.           else match S⟦hd p.disjs⟧ with
33.                | None       → S⟦OR(tl p.disjs)⟧
34.                | Some(inst) → Some(inst)
35.   | ∃∀  → let p_cand = fold ∧, p.conj, p.qps.pExists in
36.           match S⟦p_cand⟧ with
37.           | None       → None
38.           | Some(cand) →
39.               let f_check = fold ∧, TRUE, p.qps.forAll in
40.               match S⟦T⟦¬f_check⟧⟧ with
41.               | None      → Some(cand)
42.               | Some(cex) → let repl(q) =
43.                                 replace(q.body, q.decl.var, eval(cex, q.decl.var))
44.                             let f*_cex = map repl, p.qps.forAll in
45.                             let f_cex = fold ∧, TRUE, f*_cex in
46.                               S⟦p_cand ∧ T⟦f_cex⟧⟧
```

Figure 2-6: The higher-order model finding algorithm

including it in the above formalization (Figures 2-5 and 2-6) would only clutter the presentation and obscure the semantics of our approach. Instead, all the relevant details are precisely (albeit informally) provided next.

Bounds may change during the translation phase by means of skolemization: every time an existential quantifier is skolemized, a fresh variable is introduced and a bound for it is added. Therefore, we associate bounds with Procs, as different Procs may have different bounds. Whenever a composition of two Procs is performed, the resulting Proc gets the union of the two corresponding bounds.

During the solving phase, whenever the *solve* function is applied (line 28), bounds must be provided as an argument. We simply use the bounds associated with the input Proc (p). When supplying bounds for the translation of the verification condition ($\mathcal{T}[\![\neg f_{check}]\!]$, line 40), it is essential to encode the candidate solution (*cand*) as a partial instance, to ensure that the check is performed against that particular candidate, and not some other arbitrary one. This is done by bounding every variable from $p.bounds$ to the exact value it was given in *cand*:

```
let add_bound(b, var) = b + r ↦ eval(cand, var)
b_check = fold add_bound, p.bounds, p.bounds.variables
```

Finally, when translating the formula obtained from the counterexample (f_{cex}) to be used in a search for the next candidate (line 46), the same bounds are used as for the current candidate ($p_{cand}.bounds$).

2.5 Implementation

A concrete implementation of the decision procedure for higher-order constraint solving was done as an extension to Kodkod [160]. Kodkod, the backend engine used by the Alloy Analyzer, is a bounded constraint solver for *relational* first-order logic (thus, 'variable', as used previously, translates to 'relation' in Kodkod, and 'value' translates to 'tuple set'). It works by translating a given relational formula (together with bounds finitizing relation domains) into an equisatisfiable propositional formula and using an of-the-shelf SAT solver to check its satisfiability. The Alloy Analyzer delegates all its model finding (constraint solving) tasks to Kodkod. No change was needed to the existing translation from Alloy to Kodkod.

The official Kodkod distribution already offers most of the required features identified in Section 2.3. While efficient support for partial instances has always been an integral part of Kodkod, incremental solving was not supported until version 2.0.. Kodkod performs skolemization of top-level (including higher-order) existential quantifiers; the semantics of our translation from boolean formulas to `Procs` ensures that all quantifiers, regardless of their position in the formula, eventually get promoted to the top level, where they become subject to skolemization.

Conversion from atoms to expressions, however, was not available in Kodkod prior to this work. Being able to treat all atoms from a single domain as indistinguishable helps generate a stronger symmetry-breaking predicate. Since encoding a counterexample back to a candidate condition is crucial for CEGIS to scale, this work also extended Kodkod with the ability to create a singleton relation for each declared atom, after which converting atoms back to expressions (relations) becomes trivial. Kodkod's symmetry-breaking predicate generator was also updated to ignore all such singleton relations that are not explicitly used. This modification does not seem to incur any performance overhead; running the existing Kodkod test suite with and without the modification does not result in any observable time difference (in both cases running the 249 tests took around 230s).

Alloy* is implemented in Java directly following the semantics defined in Figures 2-5 and 2-6. Additionally, Alloy* performs the following important optimizations: (1) instead of creating an `OR` node for every disjunction (line 2, Figure 2-5), it first check if the disjunction is a first-order formula as a whole, in which case it creates a `FOL` node instead, and (2) it uses incremental solving to implement line 46 from Figure 2-6 whenever possible.

2.6 Case Study: Program Synthesis

Program synthesis is one of the most popular applications of higher-order constraint solving. The goal is to produce a program that satisfies a given (high-level) specification. The SyGuS [17] (syntax-guided synthesis) project has proposed an extension to SMTLIB for encoding such problems. The project has also organized a competition between solvers for the format, and provides three reference solvers.

We encoded a subset of the SyGuS benchmarks in Alloy* to test its expressive power and scalability. These benchmarks have a standard format, are well tested, and allow comparison to the performance of the reference solvers, making them a good target for evalu-

ating Alloy*.

To demonstrate our strategy for encoding program synthesis problems in Alloy*, we present the Alloy* specification for the problem of finding a program to compute the maximum of two numbers (the `max-2` benchmark).

We encode the `max-2` benchmark in Alloy* using signatures to represent the production rules of the program grammar, and predicates to represent both the semantics of programs and the constraints restricting the target program's semantics. Programs are composed of abstract syntax nodes, which can be integer- or boolean-typed.

```
abstract sig Node {}
abstract sig IntNode, BoolNode extends Node {}
abstract sig Var               extends IntNode {}
one      sig X, Y              extends Var {}

sig ITE extends IntNode {
  condition: BoolNode,
  then, elsen: IntNode
}

sig GTE extends BoolNode {
  left, right: IntNode
}
```

Integer-typed nodes include variables and if-then-else expressions, while boolean-typed nodes include greater-than-or-equal expressions. Programs in this space evaluate to integers or booleans; integers are built into Alloy, but we must model boolean values explicitly.

```
abstract sig Bool{}
one sig BoolTrue, BoolFalse extends Bool{}
```

The standard evaluation semantics can be encoded in a predicate that constrains the evaluation relation. It works by constraining all compound syntax tree nodes based on the results of evaluating their children, but does not constrain the values of variables, allowing them to range over all values.

```
pred semantics[eval: Node -> (Int+Bool)] {
  all n: ITE | eval[n] in Int and
    eval[n.condition] = BoolTrue implies
      eval[n.then] = eval[n] else eval[n.elsen] = eval[n]
  all n: GTE | eval[n] in Bool and
    eval[n.left] >= eval[n.right] implies
      eval[n] = BoolTrue else eval[n] = BoolFalse
  all v: Var | one eval[v] and eval[v] in Int
}
```

The specification says that the maximum of two numbers is equal to one of them, and greater than or equal to both.

```
pred spec[root: Node, eval: Node -> (Int+Bool)] {
  (eval[root] = eval[X] or eval[root] = eval[Y]) and
  (eval[root] >= eval[X] and eval[root] >= eval[Y])
}
```

Finally, the problem itself requires solving for some abstract syntax tree such that for all possible valuations for the variables, the specification holds.

```
pred synth[root: IntNode] {
  all eval: Node -> (Int+Bool) |
    semantics[eval] implies spec[root, eval]
}
run synth for 4
```
(A.1)

The evaluation results, including a performance comparison between Alloy* and existing program synthesizers, are presented in Section 2.8.2. Before that, two general purpose optimizations are presented in Section 2.7, which significantly imrove the overall scalability of Alloy*.

2.7 Optimizations

2.7.1 Quantifier Domain Constraints

As defined in Listing A.1 (Section 2.6), the `synth` predicate, although logically sound, suffers from serious performance issues. The main issue is the effect on the CEGIS loop of the implication inside the universal quantifier. To trivially satisfy the implication, the candidate search step can simply return an instance for which the semantics does not hold. Furthermore, adding the encoding of the counterexample refuting the previous instance is not going to constrain the next search step to find a program and a valuation for which the spec holds. This cycle can go on for unacceptably many iterations.

To overcome this problem, a bit of new syntax is added to identify the constraints that should be treated as part of the bounds of a quantification. The `synth` predicate, e.g., now becomes

```
pred synth[root: IntNode] {
  all eval: Node -> (Int + Bool) when semantics[eval] |
    spec[root, eval]
}
```

The existing first-order semantics of Alloy is unaffected, i.e.,

```
all  x when D[x] | P[x]  ⟺  all  x | D[x] implies P[x]
some x when D[x] | P[x]  ⟺  some x | D[x] and P[x]
```
(A.2)

The rule for pushing negation through quantifiers (used by the converter to NNF) becomes:

```
not (all  x when D[x] | P[x])  ⟺  some x when D[x] | not P[x]
not (some x when D[x] | P[x])  ⟺  all  x when D[x] | not P[x]
```

(which is consistent with classical logic).

The formalization of the Alloy* semantics needs only a minimal change. The change in semantics is caused by essentially **not** changing (syntactically) how the existential counterpart of a universal quantifier is obtained— only by flipping the quantifier, and keeping the domain and the body the same (line 5, Figure 2-5). Consequently, the candidate condition always searches for an instance satisfying both the domain and the body constraint (i.e., both the semantics and the spec). The same is automatically true for counterexamples obtained in the verification step. The only actual change to be made to the formalization is expanding q.*body* in line 43 according to the rules in Listing A.2.

Going back to the synthesis example, even after rewriting the `synth` predicate, unnecessary overhead is still incurred by quantifying over valuations for all the nodes, instead of valuations for just the input variables. Consequently, the counterexamples produced in the CEGIS loop do not guide the search as effectively. This observation leads to the following (final) formulation of the `synth` predicate (which was used in all benchmarks presented in Section 2.8.2):

```
pred synth[root: IntNode] {
  all env: Var -> Int |
    some eval: Node -> (Int+Bool)
    when env in eval && semantics[eval] |
      spec[root, eval]
}
```
(A.3)

Despite using nested higher-order quantifiers, it is the most efficient: the inner quantifier (over `eval`) always takes exactly one iteration (to either prove or disprove the current `env`), because for a fixed `env`, `eval` is uniquely determined.

2.7.2 Strictly First-Order Increments

We already pointed out the importance of implementing the induction step (line 46, Figure 2-6) using incremental SAT solving. A problem, however, arises when the encoding of the counterexample (as defined in lines 42–45) is not a first-order formula—since not directly translatable to SAT, it cannot be incrementally added to the existing SAT translation of the candidate search condition (p_{cand}). In such cases, the semantics in Figure 2-6 demands that the conjunction of p_{cand} and $\mathcal{T}[\![f_{cex}]\!]$ be solved from scratch, losing any benefits from previously learned SAT clauses.

This problem occurs in our final formulation of the `synth` predicate (Listing A.3), due to the nested higher-order quantifiers. To address this issue, we relax the semantics of the induction step by replacing $\mathcal{S}[\![p_{cand} \wedge \mathcal{T}[\![f_{cex}]\!]]\!]$ (line 39) with

```
// T_fo : Formula → FOL
let T_fo(f) = match p = T[[f]] with
              | FOL → p
              | OR  → FOL(reduce ∨, map(λ_d · T_fo(d).form, p.disjs))
              | ∃∀  → fold ∧, p.conj, map(T_fo, p.qps.pExists)
```

$\mathcal{S}[\![p_{cand} \wedge \mathcal{T}_{fo}(f_{cex})]\!]$

The $\mathcal{T}_{fo}$ function ensures that f_{cex} is translated to a first-order `Proc`, which can always be added as an increment to the current SAT translation of the candidate condition. The trade-off involved here is that this new encoding of the counterexample is potentially not as strong, and therefore may lead to more CEGIS iterations before a resolution is reached. For that reason, Alloy* accepts a configuration parameter (accessible via the "Options" menu), offering both strategies. In Section 2.8 we provide experimental data showing that for all of our synthesis examples, the strictly first-order increments yielded better performance.

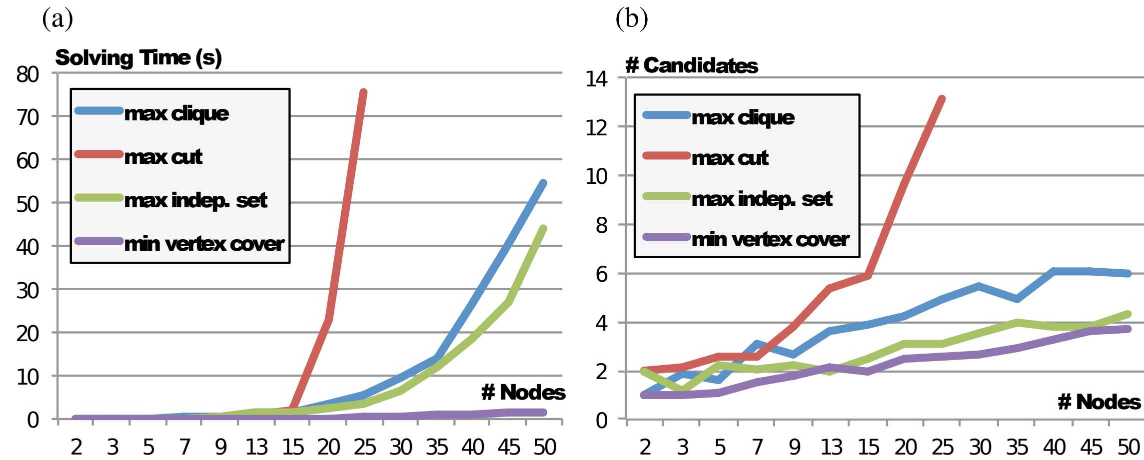


Figure 2-7: Average (over 5 different edge densities) (a) solving times, and (b) number of explored candidates for the graph algorithms

2.8 Evaluation

2.8.1 Micro Benchmarks

To assess scalability, we measure the time Alloy* takes to solve 4 classical, higher-order graph problems for graphs of varying size: `max clique`, `max cut`, `max independent set`, and `min vertex cover`.

We implemented the graph benchmarks in αRby (a unified language/environment for writing/executing declarative specifications and imperative code, presented in detail in Chapter 4). The four graph problems were specified purely declaratively, using Alloy-like higher-order relational logic.

We used the Erdős-Rényi model [52] to randomly generate graphs to serve as inputs to the benchmark problems. We generated graphs of sizes ranging from 2 to 50. To cover a wide range of edge densities, for each size we generated 5 graphs, using different probabilities of inserting an edge between a pair of nodes: 0.1, 0.3, 0.5, 0.7, 0.9. This part was written in the imperative part of αRby, which coincides with the unrestricted Ruby language.

Finally, we used the αRby runtime backed by Alloy* to automatically execute individual higher-order specifications on concrete graphs generated in the previous step. To ensure correctness of the results returned by Alloy*, we made sure they matched those of known imperative algorithms for the same problems[1]. The timeout for each Alloy* run was set to 100 seconds.

Figure 2-7 plots two graphs: (a) the average solving time across graphs size, and (b) the average number of explored candidates per problem. More detailed results, including individual times for each of the 5 densities, can be found in our technical-report [116].

The performance results show that for all problems but `max cut`, Alloy* was able to han-

[1]For `max clique` and `max independent set`, we used the Bron-Kerbosch heuristic algorithm; for the other two, no good heuristic algorithm is known, and so we implemented enumerative search. In both cases, we used Java.

dle graphs of sizes up to 50 nodes in less than a minute (`max cut` started to time out at around 25 nodes). The original target goal for these benchmarks was to be able to solve graphs with 10-15 nodes, and claim that Alloy* can be effectively used for teaching, specification animation, and small scope bounded verification, all within the Alloy Analyzer IDE (which is one of the most common uses of the Alloy technology). These results, however, suggest that executing higher-order specifications may be feasible even for declarative programming approaches such as αRby (where a constraint solver is embedded in a programming language, e.g., [112, 118, 161]), which is very encouraging.

The average number of explored candidates (Figure 2-7(b)) confirms the effectiveness of the CEGIS induction step at pruning the remainder of the search space. Even for graphs of size 50, in most cases the average number of explored candidates is around 6; the exception is, again, `max cut`, where this curve is closer to being linear. We further analyze how the total time is split over individual candidates in Section 2.8.4.

Figure 2-8 shows average solving times for individual probability thresholds used to generate graphs for the micro benchmark experiments. Lower the threshold (T), denser the graph is (similarly, higher T values lead to sparser graphs). In general, the solving time tends to be greater for denser graphs, as the search space is bigger. This trend is especially evident in the max cut problem (Figure 2-8(b)), where the solving time increases drastically for T=0.1 until Alloy* times out at graphs of size 20.

On the other hand, sparsity does not necessarily lead to faster performance. For example, in the max clique and max independent set, note how Alloy* takes longer on graphs with T=0.9 (Listings 2-8(a) and (c)) than on those with lower threshold as the graph size increases. We believe that this is because sparser graphs tend to permit fewer cliques (and independent sets) than more densely connected graphs.

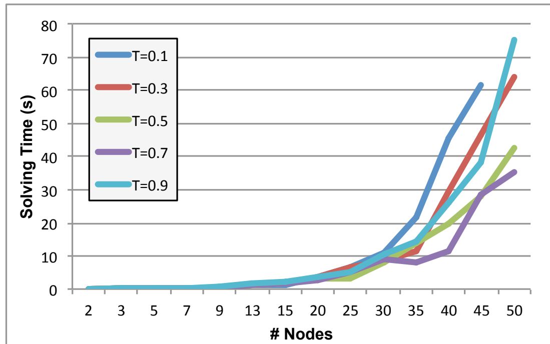
Figure 2-8 shows average solving times over different probability thresholds used to generate graphs for the micro benchmark experiments. Lower the threshold (T), denser a graph is (similarly, higher T values lead to sparser graphs). In general, the solving time tends to be greater for denser graphs, since Alloy* needs to explore a larger number of edges than in sparser graphs. This trend is especially evident in the max cut problem (Figure (b)), where the solving time increases drastically for T=0.1 until Alloy* times out at graphs of size 20.

On the other hand, sparsity does not necessarily lead to faster performance. For example, in the max clique and max independent set, note how Alloy* takes longer on graphs with T=0.9 (Figures (a) and (c)) than on those with lower threshold as the graph size increases. We believe that this is because large, sparser connected graphs tend to permit fewer cliques (and independent sets) than more densely connected graphs. As the figures show, Alloy* tends to perform best on graphs that lie in the middle of the spectrum (T=0.5 or T=0.7).

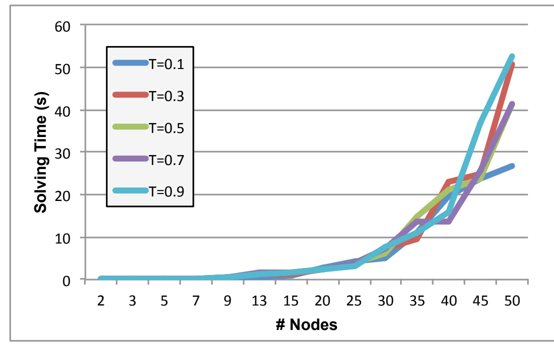
2.8.2 Program Synthesis

To demonstrate the expressive power of Alloy*, we encoded 123 out of 173 benchmarks available in the SyGuS 2014 GitHub repository [8]—all except those from the "icfp-problems" folder. We skipped the 50 "icfp" benchmarks because they all use large (64-bit) bit vectors, which are not supported by Alloy; none of them could be solved anyway by any of the

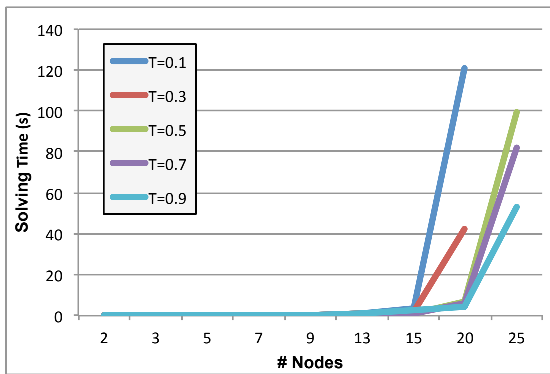
(a) max clique



(c) max independent set



(b) max cut



(d) min vertex

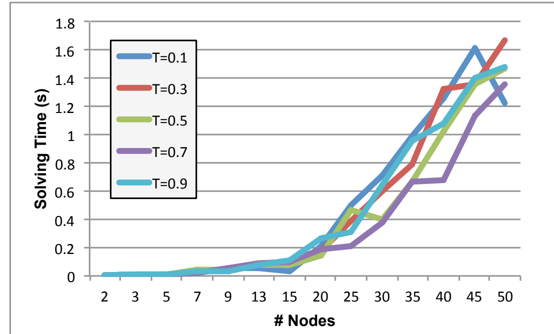


Figure 2-8: Avgerage times over thresholds for graph algorithms

solvers that entered the SyGuS 2014 competition. All of our encoded benchmarks come with the official Alloy* distribution [1] and are accessible from the main menu.

Some SyGuS benchmarks require synthesizing multiple functions at once, and some require multiple applications of the synthesized functions. All of these cases can be handled with small modifications to our encoding presented in Section 2.6. For example, to synthesize multiple functions at once, we add additional root node pointers to the signature of the `synth` predicate; to allow for multiple applications of the same function we modify the `synth` predicate to compute multiple `eval` relations (one for each application). Finally, to support SyGuS benchmarks involving bit vectors, we exposed the existing Kodkod bitwise operators over integers at the Alloy level, and also made small changes to the Alloy grammar to allow for a more flexible integer scope specification (so that integer atoms can be specified independently of integer bitwidth).

To evaluate Alloy*'s performance, we ran the same benchmarks on the same computer using the three reference solvers. We limit the benchmarks to those found in the "integer-benchmarks" folder because they: (1) do not use bit vectors (which Alloy does not support natively), and (2) allow for the scope to be increased arbitrarily, and thus are suitable for performance testing. Our test machine had an Intel dual-core CPU, 4GB of RAM, and ran Ubuntu GNU/Linux and Sun Java 1.6. We set Alloy*'s solver to be MiniSAT.

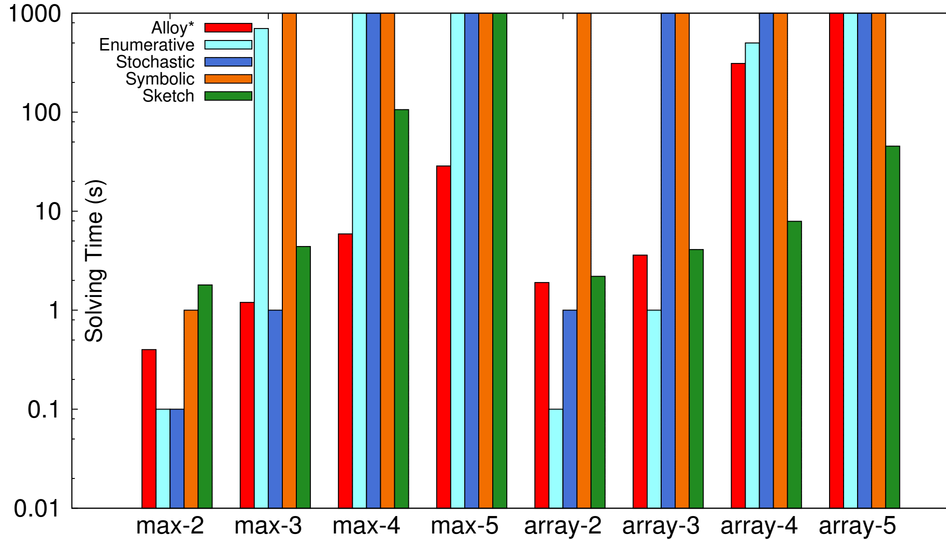


Figure 2-9: Comparison between Alloy* and Reference Solvers.

Figure 2-9 compares the performance of Alloy* against the three SyGuS reference solvers and Sketch [150], a highly-optimized, state-of-the-art program synthesizer. According to these results, Alloy* scales better than the three reference solvers, and is even competitive with Sketch. On the `array-search` benchmarks, Sketch outperforms Alloy* for larger problem sizes, but on the `max` benchmarks, the opposite is true. Both solvers scale far more predictably than the reference solvers, but Alloy* has the additional advantage, due to its generality, of a flexible encoding of the target language's semantics, while Sketch relies on the semantics of the benchmark problems being the same as its own.

Other researchers have reported [82] that benefits can be gained by specifying tighter bounds on the abstract syntax tree nodes considered by the solver. Table 2.1 confirms that for `max` and `array` significant gains can be realized by tightening the bounds. In the case of `max`, tighter bounds allow Alloy* to improve from solving the 5-argument version of the problem to solving the 7-argument version. For these experiments, Scope 1 specifies the exact number of each AST node required; Scope 2 specifies exactly which types of AST nodes are necessary; and Scope 3 specifies only how many total nodes are needed. Other solvers also ask the user to bound the analysis—Sketch, for example, requires both an integer and recursion depth bound—but do not provide the same fine-grained control over the bounds as Alloy*. For the comparison results in Figure 2-9, we set the most permissive scope (Scope 3) for Alloy*.

These results show that Alloy*, in certain cases, not only scales better than the reference solvers, but can also be competitive with state-of-the-art solvers based on years of optimization. Such cases are typically those that require a structurally complex program AST (adhering to complex relational invariants) be discovered from a large search space. When the size of the synthesized program is small, however, the results of the SyGuS 2014 competition show [18] that non-constraint based techniques, such as enumerative and stochastic search, tend to be more efficient.

Finally, Alloy* requires only the simple model presented here—which is easier to produce than even the most naive purpose-built solver. Due to its generality, Alloy* is also,

Problem	Scope 1		Scope 2		Scope 3	
	Steps	Time (s)	Steps	Time (s)	Steps	Time (s)
max-2	3	0.3	3	0.4	3	0.4
max-3	6	0.9	7	0.9	8	1.2
max-4	8	1.5	8	3.0	15	5.9
max-5	25	4.2	23	36.3	19	28.6
max-6	29	16.3	n/a	t/o	n/a	t/o
max-7	34	256.5	n/a	t/o	n/a	t/o
array-2	8	1.6	8	2.4	8	1.9
array-3	13	4.0	9	8.1	7	3.6
array-4	15	16.1	11	98.0	15	310.5
array-5	19	386.9	n/a	t/o	n/a	t/o

Table 2.1: Performance on Synthesis Benchmarks

in some respects, a more flexible program synthesis tool—it makes it easy, for example, to experiment with the semantics of the target language, while solvers like Sketch have their semantics hard-coded.

2.8.3 Benefits of the Alloy* Optimizations

We used the program synthesis benchmarks (with the tightest, best performing scope), and the bounded verification of Turán's theorem from Section 2.2 to evaluate the optimizations introduced in Section 2.7 by running the benchmarks with and without them. The baseline was a specification written without using domain constraints and an analysis without first-order increments. The next two columns correspond to (1) adding exactly one optimization (rewriting the specification to use the domain constraints for the synthesis benchmarks, and using first-order increments for Turán's theorem[2]), and (2) adding both optimizations. Table 2.2 shows the results. Across the board, writing domain constraints makes a huge difference; using first-order increments often decreases solving time significantly, and, in the synthesis cases, causes the solver to scale to slightly larger scopes.

2.8.4 Distribution of Solving Time over Individual Candidates

In Figure 2-10 we take the three hardest benchmarks (`max-7`, `array-search-5`, and `turan-10`, with tightest bounds and both optimizations applied) and show how the total solving time is distributed over individual candidates (which are sequentially explored by Alloy*). Furthermore, for each candidate we show the percentage of time that went into each of the three CEGIS phases (search, verification, induction). Instead of trying to draw strong conclusions, the main idea behind this experiment is to illustrate the spectrum of possible behaviors the Alloy* solving strategy may exhibit at runtime.

[2] If we first rewrote Turán's theorem to use domain constraints, there would be no nested CEGIS loops left, so increments would be first-order even without the other optimization.

	base	base + 1 optimization	base + both optimizations
`max2`	0.4	0.4	0.3
`max3`	7.6	1.0	0.9
`max4`	t/o	4.7	1.5
`max5`	t/o	10.3	4.2
`max6`	t/o	136.4	16.3
`max7`	t/o	t/o	163.6
`max8`	t/o	t/o	987.3
`array-search2`	140	2.9	1.6
`array-search3`	t/o	6.3	4.0
`array-search4`	t/o	76.9	16.1
`array-search5`	t/o	t/o	485.6
`turan5`	3.5	1.1	0.5
`turan6`	12.8	5.1	2.1
`turan7`	235	43	3.8
`turan8`	t/o	t/o	15
`turan9`	t/o	t/o	45
`turan10`	t/o	t/o	168

Table 2.2: Performance of Alloy* (in seconds) with and without optimizations.

The problems seen early in the search tend to be easy in all cases. The results reveal that later on, however, at least three different scenarios can happen. In the case of `max-7`, the SAT solver is faced with a number of approximately equally hard problems, which is where the majority of time is spent. In `array-search-5`, in contrast, saturation is reached very quickly, and the most time is spent solving very few hard problems. For `turan-10`, the opposite is true: the time spent solving a large number of very easy problems dominates the total time.

2.8.5 Discussion

Alloy* can solve arbitrary higher-order formulas and is sound and complete for the given bounds. The current Alloy language, however, is not necessarily the most intuitive way to express certain higher-order properties. Its semantics for encoding a candidate solution as a partial instance during the verification step, on the other hand, might not always be what the user wants.

Alloy users are used to using sigs and fields for representing relations (which are always implicitly existentially quantified over), but to express certain higher-order properties, explicit quantification is necessary. For example, to verify Turán's theorem from Listing 1, for most Alloy users it would be more natural to represent edges as a field of sig `Node` of type `Node`, and then, in the `Turan` predicate, somehow say "for all possible graph structures, assert the property of Turán's theorem". But in Alloy, `all g: Graph` already has a very different semantics than the one needed for this example [78, 91]; in Alloy*, we wanted

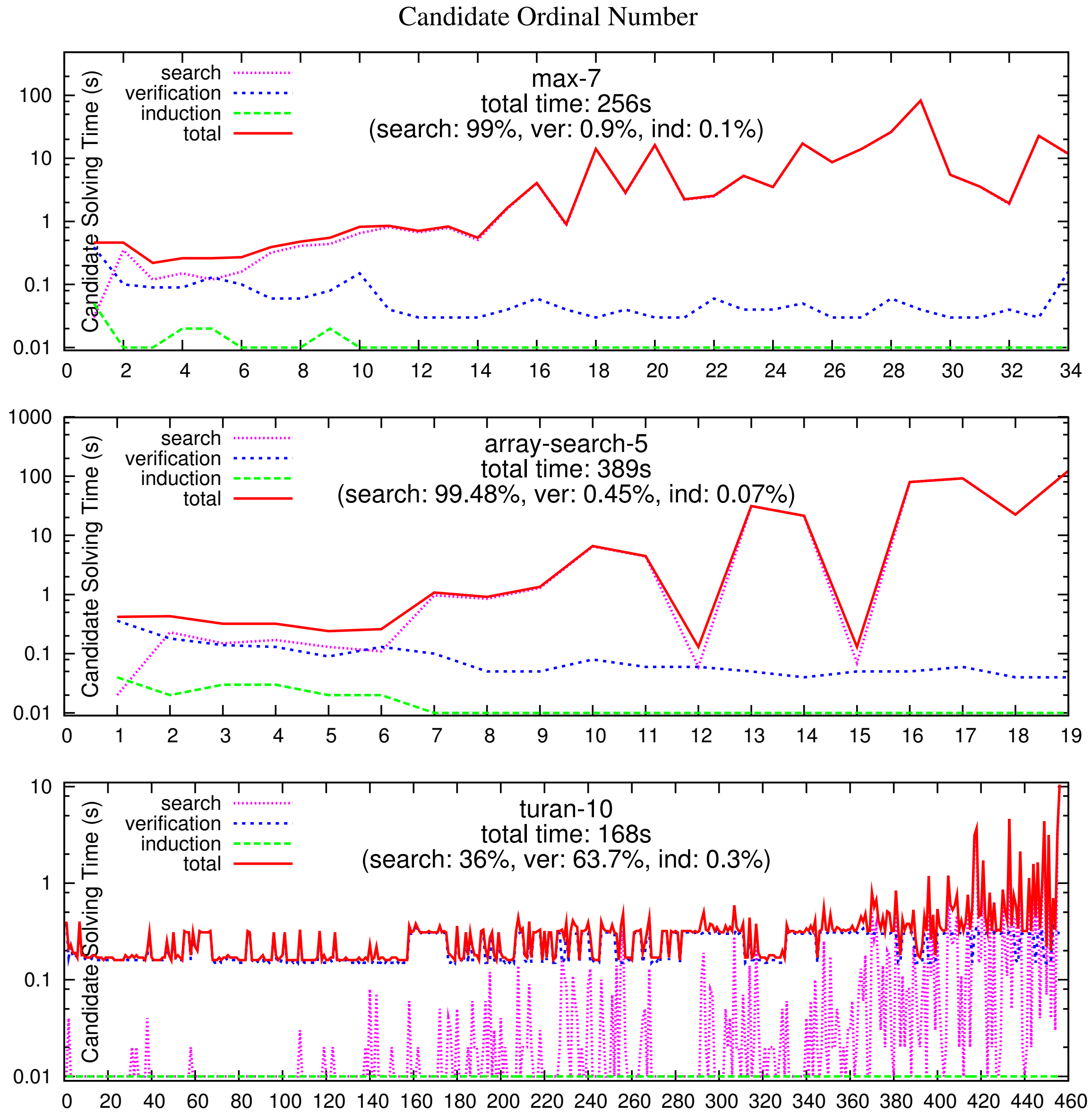


Figure 2-10: Distribution of total solving time over individual (sequentially explored) candidates for the three hardest benchmarks. Each candidate time is further split into times for each CEGIS phase

to preserve backward compatibility, as well as not introduce any significant changes to the language, so the user has to explicitly quantify over all possible `edges` relations (line 14, Listing 1) to achieve the desired behavior.

Several higher-order idioms can be solved more efficiently than by applying CEGIS. Minimization/maximization are probably the most obvious such idioms. To find a maximal clique in a graph, rather than finding a clique and asserting that there is no other clique that is larger than it, it is more efficient to start with one (arbitrary) clique, and then iteratively keep searching for a larger clique, until one cannot be found. We believe that many such

special higher-order idioms can be implemented in a general-purpose solver, but would require adding special new quantifiers, which would conflict with our initial decision to retain the existing Alloy language.

Regarding Alloy*'s semantics for encoding a candidate solution as a partial instance during the verification step, because it has no domain-specific knowledge of the problem being solved, to proceed from candidate search to verification, Alloy* always encodes all relations except the higher-order quantification variable as a partial instance. As said, Alloy users, mostly for convenience reasons, write sigs to represent relations that are implicitly existentially quantified over, so it is possible that it is not always desirable to include all of them in the partial instance.

2.9 Related Work

The ideas and techniques used in this chapter span a number of different areas of research including constraint solvers, synthesizers, program verifiers, and executable specification tools. A brief discussion of how a more powerful analysis engine for Alloy (as offered by Alloy*) may affect the plethora of existing tools built on top of Alloy is also in given.

Constraint solvers. SMT solvers, by definition, find satisfying interpretations of first-order formulas over unbounded domains. In that context, only quantifier-free fragments are decidable. Despite that, many solvers (e.g., Z3 [41]) support certain forms of quantification by implementing an efficient matching heuristic based on patterns provided by the user [40]. Certain non-standard extension allow quantification over functions and relations for the purpose of checking properties over recursive predicates [30], as well as model-based quantifier instantiation [62]. In the general case, however, this approach often leads to "unknown" being return as the result. Tools that build on top of SMT raise the level of abstraction of the input language and provide quantification patterns that work more reliably in practice (e.g., [23, 98]), but are limited to first-order forms.

SAT solvers, on the other hand, are designed to work with bounded domains. Tools built on top may support logics richer than propositional formulas, including higher-order quantifiers. One such tool is Kodkod [160]. At the language level, it allows quantification over arbitrary relations, but the analysis engine, however, is not capable of handling those that are higher-order. Rosette [161] builds on top of Kodkod a suite of tools for embedding constraint solvers into programs for a variety of purposes, including synthesis. It implements a synthesis algorithm internally, so at the user level, unlike Alloy*, this approach enables only one predetermined form of synthesis, namely, finding an instantiation of a user-provided grammar that satisfies a specified property.

Synthesizers. State-of-the-art synthesizers today are mainly purpose-built. Domains of application include program synthesis (e.g., [92, 99, 148, 150, 153]), automatic grading of programming assignments [147], synthesis of data manipulation regular expressions [69], and so on, all using different ways for the user to specify the property to be satisfied. Each such specialized synthesizer, however, would be hard to apply in a domain different than its own. A recent effort has been made to establish a standardized format for program synthesis problems [17]; this format is syntax-guided, similar to that of Rosette, and thus less general than the language (arbitrary predicate logic over relations) offered by Alloy*.

Program Verifiers. Program verifiers benefit directly from expressive specification languages equipped with more powerful analysis tools. In recent years, many efforts have been made towards automatically verifying programs in higher-order languages. Liquid types [139] and HMC [83] respectively adapt known techniques for type inference and abstract interpretation for this task. Bjørner et al. examine direct encodings into Horn clauses, concluding that current SMT solvers are effective at solving clauses over integers, reals, and arrays, but not necessarily over algebraic datatypes. Dafny [98] is the first SMT-based verifier to provide language-level mechanisms specifically for automating proofs by co-induction [100].

Executable Specifications. Many research projects explore the idea of extending a programming language with symbolic constraint-solving features (e.g., [90, 118, 141, 161, 168]). Limited by the underlying constraint solvers, none of these tools can execute a higher-order constraint. In contrast, we used αRby [112] (our most recent take on this idea where we embed the entire Alloy language directly into Ruby), equipped with Alloy* as its engine, to run all our graph experiments (where αRby automatically translated input partial instances from concrete graphs, as well as solutions returned from Alloy back to Ruby objects), demonstrating how a higher-order constraint solver can be practical in this area.

Existing Alloy Tools. Certain tools built using Alloy already provide means for achieving tasks similar to those we used as Alloy* examples. Aluminum [130], for instance, extends the Alloy Analyzer with a facility for minimizing solutions. It does so by using the low-level Kodkod API to selectively remove tuples from the resulting tuple set. In our graph examples, we were faced with similar tasks (e.g., minimizing vertex covers), but, in contrast, we used a purely declarative constraint to assert that there is no other satisfying solution with fewer tuples. While Aluminum is likely to perform better on this particular task, we showed (Section 2.8.1) that even the most abstract form of specifying such minimization/maximization tasks scales reasonably well.

Rayside et al. used the Alloy Analyzer to synthesize iterators from abstraction functions [137], as well as complex (non-pure) AVL tree operations from abstract specifications [94]. In both cases, they target a very specific categories of programs, and their approach is based on insights that hold only for those particular categories. In another instance, Montaghami et al. recognize a particular Alloy idiom involving a universal quantifier, and change the semantics to force all atoms in the domain of that quantifier to be always included in the analysis [121], effectively supporting only a small class of higher-order models.

2.10 Conclusion

Software analysis and synthesis tools have typically progressed by the discovery of new algorithmic methods in specialized contexts, and then their subsequent generalization as solutions to more abstract mathematical problems. This trend—evident in the history of dataflow analysis, symbolic evaluation, abstract interpretation, model checking, and constraint solving—brings many benefits. First, the translation of a class of problems into a single, abstract and general formulation allows researchers to focus more sharply, resulting

in deeper understanding, cleaner APIs and more efficient algorithms. Second, generalization across multiple domains allows insights to be exploited more widely, and reduces the cost of tool infrastructure through sharing of complex analytic components. And third, the identification of a new, reusable tool encourages discovery of new applications. This thesis argues that general-purpose higher-order constraint solving can, and should, be viewed in this context.

Chapter 3

Preventing Arithmetic Overflows in Alloy*

A popular approach to the analysis of undecidable logics artificially bounds the universe, making a finite search possible. In model checking, the bounds may be imposed by setting parameters at analysis time, or even hardcoded into the system description. The Alloy Analyzer [16] is a model finder for the Alloy language that follows this approach, with the user providing a 'scope' for an analysis that sets the number of elements for each basic type.

Such an analysis is not sound with respect to proof; just because a counterexample is not found (in a given scope) does not mean that no counterexample exists (in a larger scope). But it is generally sound with respect to counterexamples. That is, no spurious counterexamples are generated, so if a counterexample is found, the putative theorem does not hold. Equivalently, in the context of model finding (e.g., for the purpose of executing a declarative specification), if an instance (e.g., a solution to the specification) is found, that instance is a valid interpretation of the specification even in the unbounded context.

The soundness of Alloy's counterexamples is a consequence of the fact that the interpretation of a formula in a particular scope is always a valid interpretation for the unbounded model. There is no special semantics for interpreting formulas in the bounded case. This is possible because the relational operators are closed, in the sense that if two relations draw their elements from a given universe of atoms, then any relation formed from them (for example, by union, intersection, composition, and so on) can be expressed with the same universe.

Arithmetic operators, in contrast, are not closed. For example, the sum of two integers drawn from a given range may fall outside that range. So the arithmetic operators, when interpreted in a bounded context, appear to be partial and not total functions, and call for special treatment. One might therefore consider applying the standard strategies that have been developed for handling logics of partial functions.

A common strategy is to make the operators total functions by selecting appropriate values when the function is applied out of domain. In some logics (e.g., [67]) the value is left undetermined, but this approach is not easily implemented in a search-based model finder. Alternatively, the value can be determined. In the previous version of the Alloy Analyzer, arithmetic operators were totalized in this way by giving them wraparound semantics, so

that the smallest negative integer is regarded as the successor of the largest positive integer. This matches the semantics in some programming languages (e.g., Java), and is relatively easy to implement. Unfortunately, however, it results in counterexamples that would not arise in the unbounded context, so the soundness of counterexamples is violated. This approach leads to considerable confusion among users, and imposes the burden of having to filter out the spurious cases.

Another common strategy is to introduce a notion of undefinedness—at the value, term or formula level—and extend the semantics of the operators accordingly. However this is done, its consequence will be that formulas expressing standard properties will not hold. The associativity of addition, for example, will be violated, because the definedness of the entire expression may depend on the order of summation. In logics that take this approach, the user is expected to insert explicit guards that ensure that desired properties do not rely on undefined values. In our setting, however, where the partiality arises not from any feature of the system being described, but from an artifact of the analysis, demanding that such guards be written would be unreasonable, and would violate Alloy's principle of separating description from analysis bounds.

This thesis provides a different solution to the dilemma. Roughly speaking, counterexamples that would result in arithmetic overflow are excluded from the analysis, so that any counterexample that is returned by the analysis is guaranteed not to be spurious (while guaranteeing that no non-overflowing counterexamples are ever excluded from the analysis). This is achieved by redefining the semantics of quantifiers in the bounded setting so that the models of a formula are always models of the formula in the unbounded setting. This solution has been implemented in Alloy* (and also Alloy4.2) and it is by default turned on; it can be deactivated via the "Prevent Overflows" option.

Integers are an important part of Alloy, because they enable various program analysis tools that build on top of it; examples include tools for testing [104], verification [42], and specification execution [118]. To motivate the problem of arithmetic overflow further, a number of toy examples are presented to illustrate some of the typical anomalies that arise from treating overflow as wraparound (Section 3.1), as well as a more realistic example of modeling a minimum spanning tree algorithm, in which case none of the known (previously commonly used) "integer tricks" can be applied to fully eliminate spurious counterexamples (Section 3.2). The new semantics is formalized (Section 3.3) and an argument is made that the concrete implementation in Alloy*, realized in boolean circuits, ensures the desired semantics (Section 3.4).

To evaluate this approach, (1) a case study is shown where the analysis time is cut by 33% due to the reduced search space imposed by the new semantics, and (2) an exhaustive test suite is designed and presented to ensure that the implementation in Alloy* meets the specification.

3.1 Prototypical Overflow Anomalies

While a wraparound semantics for integer overflow is consistent and easily explained, its lack of correspondence to unbounded arithmetic produces a variety of anomalies. Most obviously, the expected properties of arithmetic do not necessarily hold: for example, that

(a) Sum of two positive integers is not necessarily positive.

```
check {
  all a, b: Int |
    a > 0 && b > 0 => a.plus[b] > 0
} for 3 Int
```

counterexample

```
Int = {-4, -3, ..., 2, 3}
a = 3; b = 1;
a.plus[b] = - 4
```

(b) Overflow anomaly involving cardinality of sets.

```
check {
  all s: set univ |
    some s iff #s > 0
} for 4 but 3 Int
```

counterexample

```
Int = {-4, -3, ..., 2, 3}
s = {S0, S1, S2, S3}
#s = -4
```

(c) Overflow anomaly involving cardinality of relations.

```
check {
  all p, q: univ -> univ |
    p in q => #p <= #q
} for 3 but 3 Int
```

counterexample

```
Int = {-4, -3, ..., 2, 3}
p = {}
q = {S0->S0, S0->S1, S1->S0, S1->S1}
#p = 0; #q = -4
```

Listing 3: Prototypical overflow anomalies in the previous version of Alloy

the sum of two positive integers is positive (Listing 3(a)). More surprisingly, expected properties of the cardinality operator may not hold. For example, the Alloy formula `some s` is defined to be true when the set `s` contains some elements. One would expect this to be equivalent to stating that the set has a cardinality greater than zero (Listing 3(b)). And yet this property will not hold if the cardinality expression `#s` overflows, since it may wrap around, so that a set with enough elements is assigned a negative cardinality.

One might imagine that this problem could be eliminated by requiring that the scope of any analysis always assign a bitwidth to integers that can measure, without overflow, the cardinality of any signature. But this is not practical, since the cardinality operator by definition counts tuples, and can be applied to any relational expression—including one of higher arity (whose cardinality rises exponentially with the number of columns). An example of the use of the cardinality operator for non-set relations is the claim that a binary relation `p` has no more tuples than a binary relation `q` if `p` is a subrelation of `q` (Listing 3(c)).

In practice, Alloy is more often used for analyzing software designs than for exploring mathematical theorems, and so properties of this kind are rarely stated explicitly. But such properties are often relied upon implicitly, and consequently, when they fail to hold, the spurious counterexamples that are produced are even harder to comprehend. Such a case arises in the the example discussed in the next section, where a test for an undirected graph being treelike is expressed by saying that there should be one fewer edge than nodes. Clearly, when using such a formulation, the user would rather not consider the effects of wraparound in counting nodes or edges.

3.2 Motivating Example

Consider checking Prim's algorithm [37, §23.2], a greedy algorithm that finds a minimum spanning tree (MST) for a connected graph with positive integral weights. Alloy is for the most part well-suited to this task, since it makes good use of Alloy's quantifiers and relational operators, including transitive closure. The need to sum integer weights, however, is potentially problematic, due to Alloy's bounded treatment of integers.

An alternative approach would be to use an analysis that includes arithmetic without imposing bounds. It is not clear, however, whether such an approach could be fully automated, since the logics that are sufficiently expressive to include both arithmetic and relational operators do not have decision procedures, and those (such as SMT) that do offer decision procedures for arithmetic are not expressive enough. We are not arguing that such an approach cannot work, and indeed, experts in these other approaches may find a suitable encoding of the problem that makes it tractable. But, either way, exploring ways to mitigate the effects of bounding arithmetic has immediate benefit for users of Alloy, and may prove useful for other tools that impose ad hoc bounds.

Listing 4 shows an Alloy representation of the problem. The sets (signatures in Alloy) `Node` and `Edge` (lines 3–10) represent the nodes and edges of a graph. Each edge has a weight (line 5) and connects a set of nodes (line 6); weights are non-negative and edges connect exactly two nodes (line 9).

This model uses the *event-based idiom* [78, §6.2.4] to model sequential execution. The `Time` signature (line 2) is introduced to model discrete time instants, and fields `covered` (line 3) and `chosen` (line 7) track which nodes and edges have been covered and selected respectively at each time. Initially (line 25) an arbitrary node is covered and no edges have been chosen. In each subsequent time step (line 27), the state changes according to the algorithm. The algorithm terminates (line 29) when the set of all nodes has been covered.

At each step, a 'cutting edge' (that is, one that connects a covered and a non-covered node) is selected such that there is no other cutting edge with a smaller weight (line 19). The edge is marked as chosen (line 20), and its nodes as covered (line 21)[1]. If the node set has already been covered (line 16), instead no change is made (line 17), and the algorithm stutters. An implementation would, of course, terminate rather than stuttering. In Alloy, however, ensuring that traces can be extended to a fixed length allows the Alloy Analyzer to employ a better symmetry breaking strategy, dramatically improving performance.

Correctness entails two properties, namely that: (1) at the end, the set of covered edges forms a spanning tree (line 39), and (2) there is no other spanning tree with lower total weight (lines 40–44). The auxiliary predicate (`spanningTree`, lines 31–38) defines whether a given set of edges forms a spanning tree, and states that, unless the graph has no edges and only one node, the edges cover all nodes of the graph (line 33), the number of given edges is one less than the number of nodes (line 35), and that all nodes are connected by the given set of edges (lines 36–37).

If we run the previous version of the Alloy Analyzer (v4.1.2) to check these two properties, the `smallest` check fails. In each of the reported counterexamples, the expression

[1] For a field `f` modeling a time-dependent state component, the expression `f.t` represents the value of `f` at time `t`.

```
open util/ordering[Time]
sig Time {}
sig Node {covered: set Time}
sig Edge {
  weight: Int,
  nodes: set Node,
  chosen: set Time
} {
  weight >= 0 and #nodes = 2
}
pred cutting (e: Edge, t: Time) {
  (some e.nodes & covered.t) and (some e.nodes & (Node - covered.t))
}
pred step (t, t': Time) {
  -- stutter if done, else choose a minimal edge from a covered to an uncovered node
  covered.t = Node =>
    chosen.t' = chosen.t and covered.t' = covered.t
  else some e: Edge {
    cutting[e,t] and (no e2: Edge | cutting[e2,t] and e2.weight < e.weight)
    chosen.t' = chosen.t + e
    covered.t' = covered.t + e.nodes }
}
fact prim {
  -- initially just one node marked
  one covered.first and no chosen.first
  -- steps according to algorithm
  all t: Time - last | step[t, t.next]
  -- run is complete
  covered.last = Node
}
pred spanningTree (edges: set Edge) {
  -- empty if only 1 node and 0 edges, otherwise covers set of nodes
  (one Node and no Edge) => no edges else edges.nodes = Node
  -- connected and a tree
  #edges = (#Node).minus[1]
  let adj = {a, b: Node | some e: edges | a + b in e.nodes} |
    Node -> Node in *adj
}
correct: check { spanningTree[chosen.last] } for 5 but 10 Edge, 5 Int
smallest: check {
  no edges: set Edge {
    spanningTree[edges]
    (sum e: edges | e.weight) < (sum e: chosen.last | e.weight)}
} for 5 but 10 Edge, 5 Int
```

Listing 4: Alloy model for bounded verification of Prim's algorithm that finds a minimum spanning tree for a weighted connected graph

`sum e: edges | e.weight` (representing the sum of weights in the alternative tree, line 43) overflows and wraps around, and thus appears (incorrectly) to have a lower total weight than the tree constructed. One might think that this overflow could be avoided by adding guards, for example that the total computed weight in the alternative tree is not negative. This does not work, since the sum can wrap around all the way back into positive territory. In Alloy*, which implements the approach described in this section, the check, as expected, yields no counterexamples for a scope of up to 5 nodes, up to 10 edges and integers ranging from -16 to 15.

3.3 Approach

This section defines a formal semantics to formulas whose arithmetic expressions might involve out-of-domain applications, such as the addition of two integers that ideally would require a value that cannot be represented. In contrast to traditional approaches to the treatment of partial functions, the out-of-domain applications arise here not from any intrinsic property of the system being modeled, but rather from a limitation of the analysis.[2] Consequently, whereas it would be appropriate in more traditional settings to produce a counterexample when an out-of-bounds application occurs, in this setting, we aim to mask such counterexamples, since they do not indicate problems with the model per se.

First, a standard three-valued logic [86] is adopted, in which elementary formulas involving out-of-bounds arithmetic applications are given the third logical value of 'undefined' ($\perp$), and undefinedness is propagated through the logical connectives in the expected way (so that, for example, 'false and undefined' evaluates to false). But the semantics of quantifiers diverges from the standard treatment: the meaning of a quantified formula is adjusted so that the bound variable ranges only over values that would yield a body that is *not undefined* (i.e., evaluates to true or false)[3]. Thus bindings that would result in an undefined quantification are masked (never presented to the user), and quantified formulas are never undefined. Since every top level formula in an Alloy model is quantified (the fields and signatures of an Alloy model are always implicitly bound in an outermost existential quantifier) this means that counterexamples (and, in the case of specification execution, instances) never involve undefined terms.

This semantics cannot be implemented directly, since the Alloy Analyzer does not explicitly enumerate values of bound variables, but instead uses a translation to boolean satisfiability (SAT) [162]. A scheme is therefore needed in which the formula is translated compositionally to a SAT formula. To achieve this, a boolean formula is created to represent whether or not an arithmetic expression is undefined. This is then propagated to

[2] Note that this discussions concern only the partial function applications arising from arithmetic operators; partial functions over uninterpreted types are treated differently in Alloy, and counterexamples involving their application are never masked.

[3] One might wonder at this point how an automated solver for this logic can possibly know in advance which bindings will not yield an overflow (without explicitly enumerating and checking every single combination); indeed, our compilation to SAT does not modify the ranges of the bound variables, rather, it uses a clever translation (as explained in Section 3.3.2) that make the associated bindings irrelevant whenever an overflow occurs.

elementary subformulas in an unconventional way that ensures the high-level semantics of quantifiers given above.

This section therefore gives two semantics: the *user-level* (high level) semantics that the user needs to understand, and the *implementation-level* (low level) semantics that justifies the analysis. This lower level semantics is then implemented by a straightforward translation to boolean circuits.

3.3.1 User-Level Semantics

As explained above, the key idea of our approach is to change the semantics of quantifiers so that the quantification domain is restricted to those values for which the body of the quantifier is defined (determined by the **dfn** predicate). For the universal quantifier, that means that the body must be satisfied for all bindings for which the body does not overflow; similarly for the existential quantifier, there must exist at least one binding for which the body does not overflow and evaluates to true:

$$\begin{aligned} [\![\texttt{all}\ r\texttt{:}\ \mathcal{R}\ \texttt{|}\ \phi(r)]\!] &\equiv \forall r \in \mathcal{R} \setminus \{i \mid r \to i \text{ causes overflow in } \phi(r)\} \bullet [\![\phi(r)]\!] \\ [\![\texttt{some}\ r\texttt{:}\ \mathcal{R}\ \texttt{|}\ \phi(r)]\!] &\equiv \exists r \in \mathcal{R} \setminus \{i \mid r \to i \text{ causes overflow in } \phi(r)\} \bullet [\![\phi(r)]\!] \end{aligned}$$

An important subtlety to note about this definition is that it is more strict than simply saying that, for a given binding, some subexpression in the body of the quantifier is undefined (e.g., because of an arithmetic overflow); additionally, it is crucial to ensure that the undefinedness is *caused* by this particular binding and not something else (e.g., an addition of two integer constants that overflows). Formally defining this causation relation at this level would only clutter this semantics and defeat its main purpose—namely to be intuitive and easy to understand. Instead, we formalize here how formulas are evaluated (denoted with $[\![\,]\!]$ brackets) and what it means for a formula/expression to be undefined (embodied in the **dfn** function); the implementation-level semantics (Section 3.3.2), of course, provides a complete formalization. A concrete example of applying the user-level semantics to evaluate quantifiers can be found in Section 3.3.3.

We have already given the quantifier evaluation semantics; all other formulas are either undefined or evaluate to the same value they do in the standard Alloy Analyzer (denoted as $\mathcal{A}[\phi]$, which is always defined):

$$[\![\phi]\!] \equiv \begin{cases} \bot, & \text{if } \neg\textbf{dfn}[\phi] \\ \mathcal{A}[\phi], & \text{otherwise} \end{cases}$$

Quantifiers are always defined. This simply follows from the idea to restrict quantification domains to bindings for which the quantifier body is defined—if every instantiation of the body is defined, the quantifier as a whole must also be defined:

$$\textbf{dfn}[\texttt{all}\ r : \mathcal{R} \mid \phi(r)] \equiv \texttt{true} \qquad \textbf{dfn}[\texttt{some}\ r : \mathcal{R} \mid \phi(r)] \equiv \texttt{true}$$

Integer expressions (i.e., those using Alloy's arithmetic operators) are defined if all arguments are defined and the evaluation does not result in overflow:

$$\textbf{dfn}[\alpha(i_1, \ldots, i_n)] \equiv \textbf{dfn}[i_1] \wedge \cdots \wedge \textbf{dfn}[i_n] \wedge \neg(\alpha[i_1, \ldots, i_n] \text{ overflows})$$

Other expressions supported in Alloy include: (1) relational algebra operators (e.g., union, intersection, etc.), (2) operators that take a relation and produce an integer (e.g., the cardinality operator), and (3) operators that take an integer and produce a relation (e.g., the int-to-expression cast operator). They are all defined if all arguments are defined:

$$\textbf{dfn}[\psi(r_1,\ \ldots,\ r_n)] \equiv \textbf{dfn}[r_1] \wedge \cdots \wedge \textbf{dfn}[r_n]$$

Predicates are boolean formulas that relate one or more (either integer or relational) expressions. In Alloy, predicates that relate integer expressions correspond directly to integer comparison operators (e.g., less than, greater than, equal to, etc.), and predicates that relate relational expressions correspond to standard boolean operators in relational algebra (e.g., subset, equality, etc.). Predicates are also defined if all arguments are defined:

$$\textbf{dfn}[\phi(r_1,\ \ldots,\ r_n)] \equiv \textbf{dfn}[r_1] \wedge \cdots \wedge \textbf{dfn}[r_n]$$

A constant is defined unless it is equal to $\bot$:

$$\textbf{dfn}[c] \equiv c \neq \bot$$

A formula is defined if it evaluates to either true or false when three-valued logic truth tables (e.g., [86, Table A.1]) of propositional operators are used (denoted here as $\wedge_3$, $\vee_3$, $\neg_3$, $\Rightarrow_3$, and $\Leftrightarrow_3$). Before the three-valued propositional operators can be applied, the operands must first be evaluated to determine their definedness:

$$\begin{array}{lcl}
\textbf{dfn}[\texttt{and}(p,q)] & \equiv & ([\![p]\!] \wedge_3 [\![q]\!]) \neq \bot \\
\textbf{dfn}[\texttt{or}(p,q)] & \equiv & ([\![p]\!] \vee_3 [\![q]\!]) \neq \bot \\
\textbf{dfn}[\texttt{implies}(p,q)] & \equiv & ([\![p]\!] \Rightarrow_3 [\![q]\!]) \neq \bot \\
\textbf{dfn}[\texttt{iff}(p,q)] & \equiv & ([\![p]\!] \Leftrightarrow_3 [\![q]\!]) \neq \bot \\
\textbf{dfn}[\texttt{not}(p)] & \equiv & (\neg_3 [\![p]\!]) \neq \bot
\end{array}$$

The semantics of the rest of the Alloy logic (in particular, of the relational operators) remains unchanged.

3.3.2 Implementation-Level Semantics

A direct implementation of the user-level semantics in Alloy would entail a three-valued logic, and the translation to SAT would thus require 2 bits for a single boolean variable (to represent the 3 possible values), a substantial change to the existing Alloy engine. Furthermore, such a change would likely adversely affect the analysis performance of models that do not use integer arithmetic. In this section, we show how the same semantics can be achieved using the existing Alloy engine, merely by adjusting the evaluation of elementary integer functions and integer predicates.

We call this semantics "implementation-level", not because it shows how boolean formulas and relational expressions are translated (rewritten) to propositional formulas (to be solved by a SAT solver), but because it is directly implementable on top of Kodkod, the solver used by the Alloy Analyzer. This section, thus, shows the mathematical semantics of evaluating formulas to boolean constants in the presence of arithmetic overflows; Section 3.4 explains how to modify Kodkod to achieve this semantics.

Syntax notes. For semantic function definitions, the expression $\texttt{fun_name}[args...]\,\sigma$ is used whenever the content of the store is irrelevant; otherwise, $\texttt{fun_name}[args...](x, i, q, b, \sigma_{\mathsf{p}})$ is written to assign concrete variable names to store fields. The same square brackets are used to explicitly designate cases where a built-in function or predicate (e.g., α, ρ, β) is to be applied to a number of constant arguments to produce a concrete (constant) result.

To make all formulas denote (and thus avoid the need for a third boolean value), a truth value must be assigned to an integer predicate even when some of its arguments are undefined. The key idea behind the approach proposed in this thesis is that in such cases a logic value can be assigned to make the subformula irrelevant in the context of the entire (enclosing) formula (i.e., the Alloy specification as a whole). This is different from common approaches (e.g., [53, 135]) which in those cases simply assign the value `false`. For example, the sentence e_1`<`e_2 will be true *iff* both e_1 and e_2 are defined and e_1 is less than e_2 (and similarly for e_1`>=`e_2):

$$\begin{aligned} [\![\ \texttt{lt}(e_1,e_2)\ ]\!] &\equiv [\![e_1]\!] < [\![e_2]\!] \ \wedge\ \textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2] \\ [\![\ \texttt{gte}(e_1,e_2)\ ]\!] &\equiv [\![e_1]\!] \geq [\![e_2]\!] \ \wedge\ \textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2] \end{aligned}$$

Negation presents a challenge. Following the user-level semantics, negation of an integer predicate (e.g., `!(`e_1`<`e_2`)`) is still undefined if any argument is undefined. Therefore, under the implementation-level semantics, `!(`e_1`<`e_2`)` must also, despite the negation, evaluate to `false` if either e_1 or e_2 is undefined (and thus have exactly the same semantics as $e_1 \geq e_2$). To achieve this behavior, the *polarity* [81] of each expression must be known (which is, loosely speaking, the number of enclosing negations). Evaluation of a binary integer predicate can be then formulated (ignoring the stack of enclosing quantifiers for the moment) as:

$$[\![\rho(e_1, e_2)]\!] \equiv \begin{cases} \rho[[\![e_1]\!], [\![e_2]\!]] \ \wedge\ (\textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2]), & \text{if polarity is positive;} \\ \rho[[\![e_1]\!], [\![e_2]\!]] \ \vee\ \neg(\textbf{dfn}[e_1] \ \wedge\ \textbf{dfn}[e_2]) & \text{otherwise.} \end{cases}$$

The polarity approach is not *compositional*, since the meaning of the negation of a formula is not simply the logical negation of the meaning of that formula. For that reason, this approach violates the law of the excluded middle, which, fortunately, will not be problematic, since the violation would only be observable for variable bindings that result in overflow and such bindings are excluded by the semantics (see Section 3.3.4).

In the presence of quantifiers, to achieve the goal of restricting quantification domains to values that do not cause overflows, the key idea is to assign truth values to formulas that overflow such that the *associated bindings* to quantification variables *become irrelevant*. For example, consider the following formula:

```
some x: Int | x > 0 and x+1 < 0
```

Kodkod unrolls this existential quantifier to a disjunction with as many clauses as there are integers in the given scope. Assuming that the bitwidth is set to 4 (integers ranging from -8 to 7), the clause in which `x` is bound to 7 will overflow. To make a clause in a disjunction of clauses irrelevant, the truth value `false` must be assigned to it; in this context, therefore, we define `x+1 < 0` to be false when `x` is bound to 7.

Now consider an example involving a universal quantifier:

```
all x: Int | x+1 > x
```

Universal quantifiers get unrolled to a conjunction of clauses; making a binding irrelevant in this case means assigning the value `true` to the associated clause. Assuming the same scope for integers, when `x` is bound to 7, we define `x+1 > x` in this context to be true.

The semantics is formally defined in Figures 3-1–3-4. Expressions and formulas are interpreted in the context of a store (defined in Figure 3-1(a)) which for each variable (`var`) bound in an enclosing quantifier holds: (1) the value of the variable in the particular binding (`val`), (2) whether the quantifier is universal or existential (`quant`), and (3) its current polarity (`polarity`). Here we only focus on handling integers, as the semantics of the relational operators remains the same.

Evaluation of integer expressions (**`aeval`**) and boolean formulas (**`beval`**) has the same effect as evaluation in the user-level semantics; it is elaborated differently here simply to account for the need to pass the store. Every time a negation is seen, the inner formula is interpreted in a store in which the polarity is negated. Quantifiers are unfolded, with the body interpreted in a new nested store (depending on the current polarity, the quantifier is adjusted according to De Morgan's laws). For the evaluation of top-level formulas, an empty existential environment is presented.

The crucial differences lie in the evaluation of integer predicates (**`ieval`**). Whereas in the user-level semantics predicates evaluate to `true`, `false` and `undefined`, in this implementation semantics predicates evaluate only to `true` or `false`. When a predicate would have been undefined in the user-level semantics, its meaning will be either `true` or `false`, chosen in such a way as to ensure that the associated binding becomes irrelevant. This choice is represented by the auxiliary function **`ensureDfn`**, which determines the truth value based on the current polarity and the stack of enclosing quantifiers.

As explained before, to make bindings resulting in overflow irrelevant, it is enough to make predicates containing existentially quantified variables evaluate to `false` and predicates containing universally quantified variables evaluate to `true`. Therefore, all expressions with universally quantified variables are identified first ($e_{\texttt{univ}}$) and a definedness condition for them ($b_{\texttt{undef}}$) is computed as a disjunction of either being undefined. For all other arguments ($e_{\texttt{ext}}$) the definedness condition ($b_{\texttt{def}}$) is a conjunction of all being defined (as before). Finally, based on the value of the polarity flag ($b_{\texttt{pol}}$), the two conditions are attached to the base result (b).

Following the formalization in Figure 3-4, evaluating an integer predicate ρ boils down to evaluating its arguments (e_1, and e_2), applying ρ to obtained integer values, and appending the definedness condition to the previous result. Since ρ is a built-in integer comparison predicate, it can be applied concretely to two given constants to obtain a concrete boolean constant (b) corresponding to the result of the comparison. If any of the arguments evaluates to $\perp$, the definedness condition appended in the next step will ensure that the concrete value of b becomes irrelevant.

The definedness condition function (**`ensureDfn`**) takes a boolean constant (b), a set of integer expressions ($e_{\texttt{in}}$), and a store. If all received expressions are defined, the result is b; otherwise the result is computed so that the associated binding in the enclosing quantifiers becomes irrelevant. Concretely, the **`ensureDfn`** function splits $e_{\texttt{in}}$ into two sets, $e_{\texttt{univ}}$ and $e_{\texttt{ext}}$, first containing expressions with universally quantified integer variables, and the

(a) Syntactic Domains

```
Formula     = BoolConst
            | IntPred(op: IntPredOp, operands: IntExpr*)
            | BoolPred(op: BoolPredOp, operands: Formula*)
            | QuantFormula(quant: Quant, var: VarDecl, body: Formula)
IntExpr     = IntConst
            | IntVar
            | IntFunc(fun: IntFuncOp, args: IntExpr*)
BoolConst   = true | false
IntConst    = ⊥ | 0 | -1 | 1 | -2 | 2 | ...
Quant       = all | some
BoolPredOp  = not₁ | and₂ | or₂ | implies₂ | iff₂
IntPredOp   = eq₂ | neq₂ | gt₂ | gte₂ | lt₂ | lte₂
IntFuncOp   = neg₁ | plus₂ | minus₂ | times₂ | div₂ | mod₂ |
              shl₂ | shr₂ | sha₂ | bitand₂ | bitor₂ | bitxor₂
Store       = {var: IntVar, val: IntConst, quant: Quant, polarity: BoolConst, parent: Store}
```

(b) Symbols

$\bot$	$\in$ IntConst	(undefined value)	b_i	$\in$ BoolConst	(boolean constants)
i_i	$\in$ IntConst	(integer constants)	p_i	$\in$ Formula	(formulas)
e_i	$\in$ IntExpr	(integer expressions)	β_i	$\in$ BoolPred	(boolean predicates)
ρ_i	$\in$ IntPred	(integer predicates)	x_i	$\in$ IntVar	(integer variables)
α_i	$\in$ IntFunc	(arithmetic functions)	q_i	$\in$ QuantFormula	(quantified formula)

(c) Stores

σ : Store (environment of nested quantifiers and variable bindings)

Figure 3-1: Overview of semantic domains, symbols, and stores to be used

aeval : IntExpr → Store → IntConst

$$\textbf{aeval}[i]\,\sigma \equiv i$$

$$\textbf{aeval}[x](x_\sigma, i_\sigma, q, b, \sigma_{\mathsf{p}}) \equiv \textbf{if } x_\sigma = x \textbf{ then } i_\sigma \textbf{ else } \textbf{aeval}[x]\sigma_{\mathsf{p}}$$

$$\textbf{aeval}[\alpha(i_1, \ldots, i_n)]\,\sigma \equiv \begin{cases} \bot, & \textbf{if } i_i = \bot \text{ or ... or } i_n = \bot; \\ \bot, & \textbf{if } \alpha[i_1, \ldots, i_n] \text{ overflows;} \\ \alpha[i_1, \ldots, i_n] & \text{otherwise.} \end{cases}$$

$$\textbf{aeval}[\alpha(e_1, \ldots, e_n)]\,\sigma \equiv \textbf{aeval}[\alpha(\textbf{aeval}[e_1]\,\sigma, \ldots, \textbf{aeval}[e_n]\,\sigma)]\,\sigma$$

Figure 3-2: Evaluation of arithmetic operations (**aeval**)

beval : Formula → Store → BoolConst

$$\textbf{beval}[b]\,\sigma \equiv b$$

$$\textbf{beval}[\rho(e_1, e_2)]\,\sigma \equiv \textbf{ieval}[\rho(e_1, e_2)]\,\sigma$$

$$\textbf{beval}[\mathsf{not}(p)](x, i, q, b, \sigma_{\mathsf{p}}) \equiv \neg\, \textbf{beval}[p](x, i, q, \neg b, \sigma_{\mathsf{p}})$$

$$\textbf{beval}[\beta(p_1, \ldots, p_2)]\,\sigma \equiv \beta\,[\textbf{beval}\,[p_1]\,\sigma, \ldots, \textbf{beval}\,[p_2]\,\sigma]$$

$$\textbf{beval}[\mathsf{all}\ x\colon \mathsf{Int}\ |\ p]\,\sigma \equiv \textbf{let } q = (\sigma.\mathsf{polarity})\ ?\ \mathsf{all} : \mathsf{some}\ \textbf{in} \equiv \bigwedge_{i \in \mathrm{Int}} \textbf{beval}\,[p]\,(x, i, q, \sigma.\mathsf{polarity}, \sigma)$$

$$\textbf{beval}[\mathsf{some}\ x\colon \mathsf{Int}\ |\ p]\,\sigma \equiv \textbf{let } q = (\sigma.\mathsf{polarity})\ ?\ \mathsf{some} : \mathsf{all}\ \textbf{in} \equiv \bigvee_{i \in \mathrm{Int}} \textbf{beval}\,[p]\,(x, i, q, \sigma.\mathsf{polarity}, \sigma)$$

Figure 3-3: Evaluation of boolean formulas (**beval**)

$\mathbf{ieval}: \ \texttt{IntPred} \rightarrow \texttt{Store} \rightarrow \texttt{BoolConst}$

$$\mathbf{ieval}[\rho(e_1, e_2)]\,\sigma \equiv \mathbf{let}\ b = \rho[\mathbf{aeval}[e_1]\,\sigma, \mathbf{aeval}[e_2]\,\sigma]\ \mathbf{in}\ \ \mathbf{ensureDfn}[b, \{e_1, e_2\}]\,\sigma$$

$\mathbf{ensureDfn}: \ \texttt{BoolConst} \rightarrow \texttt{set IntExpr} \rightarrow \texttt{Store} \rightarrow \texttt{BoolConst}$

$$\begin{aligned}
&\mathbf{ensureDfn}[b, e_{\texttt{in}}]\,(x, i, q, b_{\texttt{pol}}, \sigma_{\texttt{p}}) \equiv\\
&\quad \mathbf{let}\ \sigma = (x, i, q, b_{\texttt{pol}}, \sigma_{\texttt{p}})\ \mathbf{in}\\
&\quad\ \ \mathbf{let}\ e_{\texttt{univ}} = \{e \mid e \in e_{\texttt{in}} \wedge \mathbf{isUnivQuant}[e]\,\sigma\}\ \mathbf{in}\\
&\quad\ \ \ \ \mathbf{let}\ e_{\texttt{ext}} = e_{\texttt{in}} \setminus e_{\texttt{univ}}\ \mathbf{in}\\
&\quad\ \ \ \ \ \ \mathbf{let}\ b_{\texttt{def}} = (e_{\texttt{ext}} = \varnothing) \vee \bigwedge_{e \in e_{\texttt{ext}}} (\mathbf{aeval}[e]\,\sigma \neq \bot)\ \mathbf{in}\\
&\quad\ \ \ \ \ \ \ \ \mathbf{let}\ b_{\texttt{undef}} = (e_{\texttt{univ}} \neq \varnothing) \wedge \bigvee_{e \in e_{\texttt{univ}}} (\mathbf{aeval}[e]\,\sigma = \bot)\ \mathbf{in}\\
&\quad\ \ \ \ \ \ \ \ \ \ \mathbf{if}\ b_{\texttt{pol}}\ \mathbf{then}\ (b \vee b_{\texttt{undef}}) \wedge b_{\texttt{def}}\\
&\quad\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \mathbf{else}\ (b \vee \neg b_{\texttt{def}}) \wedge \neg b_{\texttt{undef}}
\end{aligned}$$

$\mathbf{isUnivQuant}: \ \texttt{IntExpr} \rightarrow \texttt{Store} \rightarrow \texttt{BoolConst}$

$$\begin{aligned}
&\mathbf{isUnivQuant}[e]\,(\{\}) &&\equiv \texttt{false}\\
&\mathbf{isUnivQuant}[e]\,(x, i, q, b, \sigma_{\texttt{p}}) &&\equiv \mathbf{if}\ x \in \mathbf{vars}[e]\ \mathbf{then}\ q = \texttt{all}\\
& && \quad \mathbf{else}\ \mathbf{isUnivQuant}[e]\,\sigma_{\texttt{p}}
\end{aligned}$$

$\mathbf{vars}: \ \texttt{IntExpr} \rightarrow \texttt{set IntVar}$

$$\begin{aligned}
&\mathbf{vars}[i] &&\equiv \varnothing\\
&\mathbf{vars}[x] &&\equiv \{x\}\\
&\mathbf{vars}[\alpha(e_1, \ldots, e_n)] &&\equiv \mathbf{vars}[e_1] \cup \ldots \cup \mathbf{vars}[e_n]
\end{aligned}$$

Figure 3-4: Evaluation of integer predicates (**ieval**)

second all others (which are implicitly considered as existentially quantified). The conditions associated with the existentially and universally quantified expressions ($b_{\texttt{def}}$ and $b_{\texttt{undef}}$) state that none of the expressions are undefined, or at least one is undefined, respectively. If the polarity is positive (the easiest way to think about it is the case when there are no enclosing negations), the final result is computed as $(b \vee b_{\texttt{undef}}) \wedge b_{\texttt{def}}$. Intuitively, if any existentially quantified expression is undefined ($b_{\texttt{def}}$ equal to `false`), the result must be `false` (since `false` is the value that makes a binding irrelevant inside an existential quantifier); given no existentially quantified expression is undefined, if any universally quantified expression is undefined ($b_{\texttt{undef}}$ equal to `true`), the result must be `true` (since `true` makes a binding irrelevant inside a universal quantifier); otherwise, the result is exactly equal to the original value b. When the polarity is negative, the same reasoning applies, except that the conditions need to be flipped, so $b_{\texttt{def}}$ becomes $\neg b_{\texttt{undef}}$, and $b_{\texttt{undef}}$ becomes $\neg b_{\texttt{def}}$.

The helper functions used in the definition of **ensureDfn**, **isUnivQuant** and **vars** are straightforward. An expression is universally quantified if any of its variables (computed by a simple top-down algorithm embodied in the **vars** function) is quantified over by a universal quantifier (information about the enclosing quantifiers is directly accessible from the store).

Finally, the user-level semantics also allows relational expressions to be undefined, for example, when the int-to-expression cast operator is applied to an undefined integer expression. Under the user-level semantics, whenever a boolean predicate is applied to a number

of relational expressions, the result is undefined if at least one of its arguments is undefined; here, however, a truth value must be assigned for all cases. The way this is done is analogous to evaluating integer predicates (**ieval**), which we already formally defined.

3.3.3 Correspondence Between the Two Semantics

To show that our low-level semantics correctly implements the high-level user semantics, it is enough to establish a correspondence between the two definitions of quantifiers (the low-level semantics only introduced a change to the semantics of quantifiers). Following directly from the two definitions, this is equivalent to proving that whenever an expression $p(x)$ is undefined by the laws of three-valued logic (i.e., **dfn**$[p(x)]$ is `false`), if x is *universally* quantified then **beval**$[p(x)]$ evaluates to `true`, else it evaluates to `false`.

This hypothesis could be proved by a structural induction on expressions. Instead of giving a complete proof, several interesting cases are explained instead.

The low-level evaluation of integer predicates is where the crucial differences lie. Let us therefore consider the case when $p(x)$ is an integer predicate, $\rho(e_1(x), e_2(x))$. Furthermore, let us assume that $e_1(x)$ is undefined, which makes $p(x)$ undefined as well. In this context, polarity is positive, and the value of **beval**$[\rho(e_1(x), e_2(x))]$ becomes the value of **ensureDfn**. There are two cases to consider: (1) if x is *universally* quantified, $e_{\texttt{univ}}$ contains both e_1 and e_2, $b_{\texttt{undef}}$ becomes `true`, $b_{\texttt{def}}$ is `true` by default, so the result is also `true` regardless of the base value b; (2) if x is *existentially* quantified, $e_{\texttt{ext}}$ contains both e_1 and e_2, $b_{\texttt{def}}$ becomes `false`, $b_{\texttt{undef}}$ is `false` by default, so the result is also `false`, as expected.

Let us now assume that $p(x)$ is a negation of an integer predicate, $p(x) = \neg\rho(e_1(x), e_2(x))$, and that $e_1(x)$ is again undefined. Despite the negation, $p(x)$ is *still* undefined, so the low-level evaluation should behave exactly as in the previous case. The result of **beval**$[p(x)]$ now becomes a negation of the value returned by **ensureDfn**, which, in contrast, now evaluates in a context where the polarity is negative. Following exactly the same derivation as before, it can be shown that **ensureDfn** now returns `false` for the universal case, and `true` for the existential case (because of the negative polarity), so the end result of **beval**$[p(x)]$ remains the same, as expected.

Another class of interesting examples is those with nested quantifiers. Consider the following formula:

```
run {
  all x: Int | some y: Int | y = 3 and (x = 3 implies plus[x,x] = plus[y,y])
} for 3 Int
```

Applying the user-level semantics, this formula evaluates to

$$\begin{aligned}&[\![\texttt{all x:Int | some y:Int | f[x,y]}]\!]\\ &\quad = \forall \texttt{x} \in \texttt{Int} \setminus \{\texttt{x}_{\texttt{OF}}\} \bullet [\![\texttt{some y:Int | f[x,y]}]\!]\\ &\quad = \forall \texttt{x} \in \texttt{Int} \setminus \{\texttt{x}_{\texttt{OF}}\} \bullet \exists \texttt{y} \in \texttt{Int} \setminus \{\texttt{y}_{\texttt{OF}}\} \bullet [\![\texttt{f[x,y]}]\!]\end{aligned}$$

where `f[x,y]` is `y = 3 and (x = 3 implies plus[x,x] = plus[y,y])`. The set of excluded bindings for variable `x`, $\{\texttt{x}_{\texttt{OF}}\}$, is a set of all integers for which `plus[x,x]` overflows (which is the only subexpression containing variable `x` that can possibly be undefined); similarly, $\{\texttt{y}_{\texttt{OF}}\}$ is a set of all integers for which `plus[y,y]` overflows. In both cases, the excluded

bindings are equal to $\{-3, 2, 3\}$. Since the only binding that satisfies `f[x,y]` is $x \to 3$, $y \to 3$, the formula as a whole is unsatisfiable. Now following the implementation-level semantics (the definition of the **`ensureDfn`** function), the e_{ext} set contains `plus[y,y]`; given the binding $x \to 3$, $y \to 3$, b_{def} evaluates to false, and since the polarity is positive (b_{pol} is true), **`ensureDfn`** returns false, thus, the formula as a whole is, again, unsatisfiable.

As an exercise, the reader may want to check that if the quantifiers in the previous formula swap places, the result does not change. When `plus[x,y] = plus[x,y]` is used instead of `plus[x,x] = plus[y,y]`, the order of quantification does matter: the formula

```
all x: Int | some y: Int | y = 3 and (x = 3 implies plus[x,y] = plus[x,y])
```

is not satisfiable, but

```
some y: Int | all x: Int | y = 3 and (x = 3 implies plus[x,y] = plus[x,y])
```

is, because the two arithmetic expressions are both treated as "existentially" quantified in the former case, and "universally" in the latter. This behavior is not simply an artifact of the presented formalization. Rather, it is by design, as the intention was to have

```
all x: Int | some y: Int | plus[x,y] > x
```

be false (indeed, in a bounded setting, for `x = MAXINT`, there is no integer that can be added to it to obtain an integer greater than `MAXINT`), and

```
some y: Int | all x: Int | plus[x,y] > x
```

be true (for, e.g., `y = 1`, every integer `x` when added to it can only produce a number greater than `x`).

3.3.4 The Law of the Excluded Middle

As mentioned earlier, the non-compositional rule for negation breaks the law of the excluded middle. Usually, this is not a problem.

Consider checking the theorem that all integers (within the bitwidth of 3) when multiplied by two are either less than zero or not less than zero:

```
check { all x: Int | x.mul[2] < 0 or !(x.mul[2] < 0) } for 3 Int
```

If we run the Alloy Analyzer with overflow prevention turned on, this sentence is interpreted as "*for all integers x s.t. x times two does not overflow, x times two is either less than zero or not less than zero*", and thus no counterexample is found, which is consistent with classical logic.

Similarly, if we ask the Alloy Analyzer to find all instances of `x` where `x` multiplied by two is either less or not less than zero, we will not get all integers from the domain, but only those that do not overflow when multiplied by two.

```
run {
  some x: Int | x.mul[2] < 0 or !(x.mul[2] < 0)
} for 3 Int
```

instances

```
x = -2, x = 0
x = -1, x = 1
```

In a sense, however, the violation of the law of the excluded middle is visible if truth is associated with whether or not a check yields a counterexample at all. For example, a check of whether 4 plus 5 is *equal* to 6 plus 3 for the bitwidth of 4 (`Int = {-8, ..., 7}`)

does not return a counterexample, but neither does a check of whether 4 plus 5 is *different* from 6 plus 3.

```
check { 4.plus[5]  = 6.plus[3] } for 4 Int -- no counterexample found
check { 4.plus[5] != 6.plus[3] } for 4 Int -- no counterexample found
```

Though this might at first appear confusing, it is consistent with the main design decision behind this approach: indeed, for a bitwidth of 4, there is no non-overflowing instance in which 4 plus 5 is either equal to or different from 6 plus 3.

3.4 Implementation in Circuits

Detecting arithmetic overflows at the level of relational logic would be difficult, and probably inefficient. We therefore implemented our approach at the level of the translation to propositional logic, as an extension to Kodkod.

The Alloy Analyzer delegates the core task of finding satisfying models to Kodkod, a bounded constraint solver for relational first-order logic. Kodkod works by translating a given relational formula (together with bounds) into an equisatisfiable propositional formula and using an of-the-shelf SAT solver to check its satisfiability.

Even though the goal here is to translate the input formula into a digital circuit (instead of evaluating it to a boolean constant), the denotational semantics defined in Section 3.3.1 still applies, simply because all logic operators used in our formalization are also available at the level of digital gates. Only Kodkod's translation of appropriate terms had to be modified, directly following the formal semantics presented in this thesis. The list of changes made to Kodkod is as follows:

- *the translation of arithmetic operations* was changed to generate an additional one-bit overflow circuit which is set if and only if the operation overflows. Textbook overflow circuits were used for all arithmetic operations supported by Kodkod, and the definition in Figure 3-2 was used to propagate this information from the operands to the operation result;

- *the way the store gets updated* was modified so that it additionally keeps track of the polarity and the quantification stack (the store is defined in Figure 3-1, and how it gets updated in Figure 3-3);

- *the translation of boolean predicates* was also updated so that the original circuit representing the predicate result is extended to include the definedness conditions, exactly as defined in Figure 3-4;

3.5 Evaluation

The goal of this evaluation is twofold: (1) test the new semantics as embodied in code within Alloy* and make sure that all the anomalies presented in Section 3.1 are fixed, as well as spurious counterexamples caused by integer overflows in several other prototypical cases are eliminated, and (2) provide some evidence about potential effects (in terms of

```java
public void testArithmeticOverflows() {
  int bw = 5, l = -(1 << (bw - 1)), h = (1 << (bw - 1));
  IntOperator[] ops = new IntOperator[]{PLUS, MINUS, MULTIPLY, DIVIDE, MODULO};
  for (IntOperator op: ops) for (int i=l; i<h; i++) for (int j=l; j<h; j++) {
    // f: ret = Int[(i op j)]
    Formula f = ret.eq(compose(op, constant(i), constant(j)).toExpression());
    int javaRes = -1;
    try { javaRes = exeJava(op, i, j); } catch(ArithmeticException e){ continue; }
    if (javaRes >= h || javaRes < l) {
      try { exeKodkod(f); fail("Overflow not detected"); } catch(NoSolution e){}
    } else {
      assertEquals("Wrong result", javaRes, exeKodkod(f));
    }
  }
}
```

Listing 5: A unit test for exhaustively checking overflow detection in elementary arithmetic formulas

scalability of the analysis) the new semantics might have on existing models already using integer arithmetics.

3.5.1 Exhaustive Testing of the New Translation Scheme

To ensure the correctness of the new semantics, as well as the implementation of the new translation scheme, we ran a series of exhaustive tests (up to a finite bound) and verified that the results were as expected.

Basic Arithmetic Tests

The unit test in Listing 5 exhaustively checks whether overflows are detected in all supported binary arithmetic functions. It dynamically constructs all possible expressions in the form of

```
ret = {i op j},
```

where `i` and `j` are integers drawn from {`-16`, ..., `15`}, and `op` is an arithmetic operator drawn from {`+`, `-`, `*`, `/`, `%`}. For each case, it first computes the expected result using Java built-in arithmetic operators (which certainly will not overflow in this small scope). If the computed result falls outside of the `[-16, 15]` range, an overflow is expected, that is, no satisfying instance is expected to be found by Kodkod; otherwise, the value of `ret` returned by Kodkod is expected to be equal to the value obtained in Java.

A slightly modified version of this test uses the same ideas to check all expressions in the form of

```
ret = (some {i op j} => {i op j} else {-1}).
```

This test shows that a constraint can be written to check whether an integer expression overflows. The formula above uses that feature to assign a default value (-1) to `ret` whenever `i`

op j overflows. One might (wrongly) expect this entire formula to be unsatisfiable when {i op j} overflows; recalling the semantics, however, applying the int-to-expression cast operator to an overflowing integer expression (i op j) results in an undefined relational expression, then applying the **some** boolean predicate to it yields false (since the polarity is positive), which finally selects the else branch (regardless of the evaluation of the then branch).

Yet another variation of the same test uses relations in place of integer constants i and j, so that it can ask Kodkod to enumerate all valid solutions to ret = {i op j}. It then checks that the result exactly matches the set of all non-overflowing solutions computed in Java.

Testing Tautologies

Table 3.1 shows the list of arithmetic tautologies that were checked for counterexamples using Kodkod.

For the purpose of exercising various polarity cases (that is, nestings of negations and quantifiers), for each row from Table 3.1 we ran the test on the following equivalent formulas:

```
all decl | pre => post              !(some decl | !(pre => post))
!!(all decl | pre => post)          !!!(some decl | !(pre => post))
all decl | !!(pre => post)          !(some decl | pre && !post)
all decl | !(pre && !post)          !(some decl | !!(pre && !post))
all decl | !pre || post             !(some decl | !(!pre || post))
```

The cardinality operator (#) returns the number of elements in a given relation. If that number is greater than the largest integer in the scope, the operation overflows, often causing anomalies especially difficult to debug. To ensure that the new semantics correctly prevents overflows in those cases, we checked the following tautologies:

```
all s:    set univ | #s >= 0
no  s:    set univ | #s < 0
all s:    set univ | (some s) iff #s > 0
all s, t: set univ | #(s + t) >= #s && #(s + t) >= #t
all s, t: set univ | s in t => #s <= #t
all s, t: set univ | (no s & t && some s) => #(s + t) > #t
```

As expected, none of the tautologies could be refuted given the integer bitwidth of 5.

3.5.2 Effects on Models with Integer Arithmetic

We took a previously published model of a flash filesystem [88], which uses arithmetic operations and whose analysis is non-trivial, and compared its execution under the old (Alloy4) and new (Alloy*) analysis schemes. This model involves both assertions (that certain properties hold) and simulations (that produce sample scenarios). First, we checked that there are no new spurious counterexamples, and that none of the expected valid scenarios are lost. This was not the focus of our evaluation, however, since the design of the analysis ensures it. Rather, our concern was that the addition of new clauses to the SAT formula generated by the Analyzer might increase translation and solving time.

decl	precondition	postcondition
`a, b: Int`	`a > 0 && b > 0`	`a + b > 0 && a + b > a && a + b > b`
`a, b: Int`	`a < 0 && b < 0`	`a + b < 0 && a + b < a && a + b < b`
`a, b: Int`	`a > 0 && b < 0`	`a - b > 0 && a - b > a && a - b > b`
`a, b: Int`	`a < 0 && b > 0`	`a - b < 0 && a - b < a && a - b < b`
`a, b: Int`	`a > 0 && b > 0`	`a * b > 0 && a * b >= a && a * b >= b`
`a, b: Int`	`a < 0 && b < 0`	`a * b > 0 && a * b >= -a && a * b >= -b`
`a, b: Int`	`a > 0 && b < 0`	`a * b < 0 && -(a * b) > a && -(a * b) > -b`
`a, b: Int`	`a < 0 && b > 0`	`a * b < 0 && -(a * b) > -a && -(a * b) > b`

Table 3.1: List of checked arithmetic tautologies

The new translation always results in a larger SAT formula, because extra clauses are needed to rule out models that overflow. One might imagine that adding clauses would cause the solving time to increase. On the other hand, the additional clauses might result in a smaller search space, and thus potentially reduce the search time.

We ran all checks that were present in the "concrete" module of the model. The first 11 (`run1` through `run11`) are simulations (which all find an instance), and the remaining 5 (`check1` through `check5`) are checks, which, with the exception of `check5`, produce no counterexamples. For each check, we measured both the translation and solving time, as shown in Table 3.2. As expected, in some cases the analysis runs faster, and sometimes it takes longer. In total, with the overflow prevention turned on, the entire analysis finished in about 8 hours, as opposed to almost 12 hours that the same analysis took otherwise.

	`run1`		`run2`		`run3`		`run4`		`run5`		`run6`		`run7`		`run8`		`run9`	
old `[s]`	1.2	0.9	2.1	0.4	0.8	0.2	12.9	2.3	5.9	0.5	12.7	1.0	11.9	1.1	9.0	1.0	12.5	1.0
new `[s]`	1.2	0.8	1.6	0.4	0.8	0.3	13.4	8.7	6.2	0.5	12.6	0.8	12.1	1.5	9.1	1.0	12.7	2.6
diff `[s]`	0	0.1	0.5	0	0	-0.1	-0.5	-6.4	-0.3	0	0.1	0.2	-0.2	-0.4	-0.1	0	-0.2	-1.6
x `[%]`	0	11.1	23.8	0	0	-50.0	-3.9	-278.3	-5.1	0	0.8	20.0	-1.7	-36.4	-1.1	0	-1.6	-160.0

	`run10`		`run11`		`check1`		`check2`		`check3`		`check4`		`check5`		**`total`**
old `[s]`	25.7	14.8	20.0	39.6	12.1	2190.7	12.0	30673.3	12.5	3713.2	12.3	3.0	74.3	5782.6	42663.5
new `[s]`	25.9	12.5	20.2	12.6	12.2	1670.4	12.2	16741.9	12.7	3526.9	12.5	1.3	73.9	7083.5	29304.5
diff `[s]`	-0.2	2.3	-0.2	27	-0.1	520.3	-0.2	13931.4	-0.2	186.3	-0.2	1.7	0.4	-1300.9	13359.0
speedup `[%]`	-0.8	15.5	-1.0	68.2	-0.8	23.8	-1.7	45.4	-1.6	5.0	-1.6	56.7	0.5	-22.5	31.3

Table 3.2: Analysis times of checks from the flash filesystem model [88]

3.6 Related Work

The problem addressed in this thesis is an instance of the more general problem of handling partial functions in logic. The most important difference, however, is that, in this case, the out-of-bound function applications arise due to deficiencies in the analysis, rather than from the inherent semantics of the logic. Requiring the user to introduce guards in the formal description itself to mitigate the effects of undefinedness is therefore not acceptable.

Despite this fundamental difference, our approach shares some features of several previously explored approaches.

The *Logic of Partial Functions* (LPF) was proposed for reasoning about the development of programs [86, 87], and was adopted in VDM [85]. In this approach, not only integer predicates but also boolean formulas may be non-denoting, so truth tables extended to a three-valued logic are needed. This allows guards for definedness to be treated intuitively; thus, for example, even when "x" is equal to zero, formula `x!=0 => x/x=1`, evaluates to `true` in spite of `x/x=1` being undefined. Our approach uses this three-valued logic for determining whether the body of a quantified formula is undefined, but the meaning of the formula as a whole is treated differently—masking the binding that produces undefinedness rather than interpreting the quantification in the same three-valued logic.

Our implementation-level semantics adopts the *traditional approach to partial functions* (a term coined by Farmer [53]), in which all formulas must be denoting but functions may be partial. Farmer's approach, however, leaves open whether, given an undefined `a`, `!(a=a)` and `a!=a` have different meanings—an issue that in the standard setting is hard to resolve because of the competing concerns of compositionality and preserving complementarity of predicates. In our case, the non-compositional choice fits nicely with the user-level semantics.

Like the Alloy Analyzer, SMT [26] solvers can also be used for model finding. They all support unbounded integer arithmetic, so the problem of overflows does not arise. However, using Alloy over SMT-based tools has certain benefits, most notably the expressiveness of the Alloy relational language. There are higher-level languages that build on SMT technologies (e.g. Dafny [98]), but for a task similar to verifying Prim's algorithm, such tools are typically not fully automatic, and demand that the user provide intermediate lemmas.

Model-based languages such as B [13] and Z [151], being designed for specifying programs, make extensive use of partial functions. Both are based on set theory, and model functions as relations. Whereas in Alloy out-of-bounds applications of partial functions over uninterpreted types result in the empty set, in B such an application results in an unknown value [155] (consequently, propositions containing unknown values cannot be proved). The initial specification of the Z notation [151] left the handling of partial functions open.

Several different approaches have been proposed (see [20] for a survey); in the end, it appears that the same approach as in B has evolved to be the norm [155]. In both Z and B, integers are unbounded, and so the problems of integer overflow do not arise. On the other side, the tools for discharging proof obligations (e.g. Rodin [12]) are typically less automated than the Alloy Analyzer.

3.7 Conclusion

A new logic has been presented, a logic that provides a special treatment of quantifiers to eliminate models yielding out-of-domain applications of partial functions. The main motivation for the new logic was making an automated analysis based on it sound with respect to counterexamples (i.e., instances), even in the presence of partial functions. This thesis focuses on applying this approach to integer functions (which in a bounded setting

become partial), with the goal of excluding the models containing arithmetic overflows. The proposed approach has been implemented in the Alloy Analyzer, and thus eliminated its only source of spurious counterexamples, making it more practical for use in fully automated tools that are built on top of it (like Alloy* [117], αRby [112], SQUANDER [118], Forge [43, 44], TestEra [104], etc.).

Part II

Unifying Specification and Implementation Languages

Chapter 4

αRby: An Embedding of Alloy in Ruby

Part I focused on improving automated constraint solving—a technology often used as the main engine in many declarative programming systems—and presented Alloy* and its key advancements: (1) full support for higher-order formulas, and (2) a novel treatment of partial arithmetic function for the purpose of preventing integer overflows. The next step when developing a specification-based declarative programming system is designing an interface between the user (programmer) and the underlying solver. This chapter discusses the set of challenges that arise in this phase, offers a novel approach to address them, and presents a tool implementing the approach.

Probably the most important characteristic of any specification language is its expressive power. One of the main benefits of writing a specification is to describe the behavior of a program *succinctly* and *accurately*, in terms that are easier to understand and comprehend than any possible imperative implementation of the same program. To achieve this, specification languages often resort to using various logics and formalisms that are not commonly found in traditional programming languages. This poses a language design challenge when developing a programming environment which allows declarative specifications and imperative statements be arbitrarily mixed.

To write specifications, previous approaches either employ the same imperative constructs provided by the host language (e.g., [90,161]) or simply use plain strings (e.g., [118, 141]); the former often has to sacrifice expressive power, while the latter tends to be inconvenient to use and manipulate (because the specification language is not first-class). This thesis takes a different approach: instead of adding specification statements to an imperative programming language, we propose that, at least conceptually, an imperative shell is created around a specification language. We recognize that extending an existing specification language to support imperative construct may turn to be very impractical from the implementation standpoint, so this chapter also presents how the same idea can be implemented more conveniently using a modern programming language equipped with powerful metaprogramming features.

We present αRby, an embedding of the Alloy language in Ruby, as well as the benefits of having a declarative specification and modeling language (backed by an automated solver) embedded in a traditional object-oriented imperative programming language. This approach aims to bring these two distinct paradigms (imperative and declarative) together in a novel way. We argue that having the other paradigm available within the same lan-

guage is beneficial to both the modeling community of Alloy users and the object-oriented community of Ruby programmers. Our embedding is specific in that one of its main objectives is to preserve as much as possible the original character of the chosen specification language (Alloy), in terms of both syntax and semantics.

The main contributions include:

- An argument for a new kind of combination of a declarative and an imperative language, justified by a collection of examples of functionality implemented in a variety of tools, all of which are subsumed by this combination, becoming expressible by the end-user;
- An embodiment of this combination in αRby, a deep embedding of Alloy in Ruby, along with a semantics, a discussion of design challenges, and an implementation for readers to experiment with;
- An illustration of the use of the new language in a collection of small but non-trivial examples.

4.1 Why an Imperative Shell Around a Modeling Language

A common approach in formal modeling and analysis is to use (1) a declarative language (based on formal logic) for writing *specifications*, and (2) an automated constraint solver for finding valid *models* of such specifications[1]. Such models are most often either examples (of states or execution traces), or counterexamples (of correctness claims).

In many practical applications, however, the desired analysis involves more than a single model finding step. At the very least, a tool must convert the generated model into a form suitable for showing to the user; in the case of Alloy [78], this includes projecting higher-arity relations so that the model can be visualized as a set of snapshots. In some cases, the analysis may involve repeating the model finding step, e.g., to find a minimal model by requesting a solution with fewer tuples [130].

To date, these additional analyses have been hard-coded in the analysis tool. The key advantage of this approach is that it gives complete freedom to the tool developer. The disadvantage is that almost no freedom is given to the modeler, who must make do with whatever additional processing the tool developer chose to provide.

This thesis explores a different approach, in which, rather than embellishing the analysis in an ad hoc fashion in the implementation of the tool, the modeling language itself is extended so that the additional processing can be expressed directly by the end user. An imperative language seems most suitable for this purpose, and the challenge therefore is to find a coherent merging of two languages, one declarative and one imperative. We show how this has been achieved in the merging of Alloy and Ruby.

[1] Throughout this chapter, we use the term 'model' in its mathematical sense, and never to mean the artifact being analyzed, for which we use the term 'specification' instead.

This challenge poses two questions, one theoretical and one practical. Theoretically, a semantics is needed for the combination: what combinations are permitted, and what is their meaning? Practically, a straightforward way to implement the scheme is needed. In particular, can a tool be built without requiring a new parser and engine that must handle both languages simultaneously?

Roughly speaking, αRby answers these questions as follows. Execution consists of interleaved phases of imperative program execution and declarative model finding. The imperative phases construct specifications as abstract objects, which are then solved by the model finder; subsequent imperative phases can process the resulting models, and construct new specifications, based not only on previous specifications but also on the models found. To implement this, the modeling language is embedded as a domain-specific language in the imperative programming language. The keywords of the specification are implemented as functions that construct abstract syntax trees. Thus no special parsing is required, and the model finding phase is handled by passing a tree representing the entire specification to an existing model finder.

This project focuses on the combination of Alloy and Ruby. In some respects, these choices are significant. Alloy's essential structuring mechanism, the signature [77], allows a relational logic to be viewed in an object-oriented way (in just the same way that instance variables in an object-oriented language can be viewed as functions or relations from members of the class to members of the instance variable type). So Alloy is well suited to an interpretation scheme that maps it to object-oriented constructs. Ruby is a good choice because, in addition to being object-oriented and providing (like most recent scripting languages) a powerful metaprogramming interface, it offers a syntax that is flexible enough to support an almost complete embedding of Alloy with very few syntactic modifications.

At the same time, these key ideas could be applied to other languages; there is no reason, in principle, that similar functionality might not be obtained by combining the declarative language B [13] with the programming language Racket, for example.

4.2 Examples of Motivating Use Cases

Analysis of a declarative specification typically involves more than just model finding. In this section, we outline the often needed additional steps.

Preprocessing. The specification or the analysis command may be updated based on user input. For example, in an analysis of Sudoku, the size of the board must be specified. In Alloy, this size would be given as part of the 'scope', which assigns an integer bound to each basic type. For Sudoku, we would like to ensure that the length of a side is a perfect square; this cannot be specified directly in Alloy.

Postprocessing. Once a model has been obtained by model finding, some processing may be needed before it is presented to the user. A common application of model finding in automatic configuration is to cast the desired configuration constraints as the specification, then perform the configuration steps based on the returned solution.

Partial instances. A partial instance is a partial solution that is known (given) upfront. In solving a Sudoku problem, for example, the model finder must be given not only the rules of Sudoku but also the partially filled board. It is easy to encode a partial solution

as a formula that is then just conjoined to the specification. But although this approach is conceptually simple, it is not efficient in practice, since the model finder must work to reconstruct from this formula information (namely the partial solution) that is already known, thus needlessly damaging performance.

Kodkod (the back-end solver for Alloy) explicitly supports partial instances: it allows the user to specify relation *bounds* in terms of tuples that must (*lower bound*) and may (*upper bound*) be included in the final value. Kodkod then uses the bounds to shrink the search space, often leading to significant speedups [160]. At the Alloy level, however, this feature is not directly available[2].

Staged model finding. Some analyses involve repeated calls to the model finder. In the simplest case, the bounds on the analysis are iteratively increased (when no counterexample has been found, to see if one exists within a larger scope), or decreased (when a counterexample has already been found, to see if a smaller one exists).

Mixed execution. Model finding can be used as a step in a traditional program execution. In this case, declarative specifications are executed 'by magic', as if, in a conventional setting, the interpreter could execute a program assertion by making it true despite the lack of any explicit code to establish it [90, 118]. Alternatively, flipping the precedence of the two paradigms, the interpreter can be viewed as a declarative model finder that uses imperative code to setup a declarative specification to be solved. In this chapter, we are primarily concerned with the latter direction, which has not been studied in the literature as much.

The Alloy Analyzer—the official and the most commonly used IDE for Alloy—does not currently provide any scripting mechanisms around its core model finding engine. Instead, its Java API must be used to automate even the most trivial scripting tasks. Using the Java API, however, is inconvenient; the verbosity and inflexibility of the Java language leads to poor transparency between the API and the underlying Alloy specification, making even the simplest scripts tedious and cumbersome to write. As a result, the official API is rarely used in practice, and mostly by expert users and researchers building automated tools on top of Alloy. This is a shame, since a simple transparent scripting shell would be, in many respects, beneficial to the typical Alloy user—the user who prefers to stay in an environment conceptually similar to that of Alloy and not have to learn a second, foreign API.

This is exactly what αRby provides—an embedding of the Alloy language in Ruby. Thanks to Ruby's flexibility and a very liberal parser, the αRby language manages to offer a syntax remarkably similar to that of Alloy, while still being syntactically correct Ruby. To reduce the gap between the two paradigms further, instead of using a separate AST, αRby maps the core Alloy concepts are onto the corresponding core concepts in Ruby (e.g., sigs are classes, fields are instance variables, atoms are objects, functions and predicates are methods, most operators are the same in both languages). αRby automatically interoperates with the Alloy back end, so all the solving and visualization features of the Alloy Analyzer can be easily invoked from within an αRby program. Finally, the full power of the Ruby language is at the user's disposal for other tasks unrelated to Alloy.

[2]The Alloy Analyzer recognizes certain idioms as partial instances; some extensions (discussed in Section 4.7) support explicit partial instance specification.

4.3 αRby for Alloy Users

A critical requirement for embedding a modeling language in a programming language is that the embedding should preserve enough of the syntax of the language for users to feel comfortable in the new setting. We first introduce a simple example to illustrate how αRby achieves this for Alloy. Next, we address the new features brought by αRby, highlighted in Section 4.2, which are the primary motivation for the embedding.

Consider using Alloy to specify directed graphs and the *Hamiltonian Path* algorithm. *Signatures* are used to represent unary sets: `Node`, `Edge`, and `Graph`. *Fields* are used to represent relations between the signatures: `val` mapping each `Node` to an integer value; `src` and `dst` mapping each `Edge` to the two nodes (source and destination) that it connects; and `nodes` and `edges` mapping each `Graph` to its sets of nodes and edges.

A standard Alloy model for this is shown in Listing 6(b), lines 2–4; the same declarations are equivalently written in αRby as shown in Listing 6(a), lines 2–4.

To specify a Hamiltonian path (that is, a path visiting every node in the graph exactly once), a predicate is defined; lines 6–12 in Listings 6(b) and 6(a) show the Alloy and αRby syntax, with equivalent semantics. This predicate asserts that the result (`path`) is a sequence of nodes, with the property that it contains all the nodes in the graph, and that, for all but the last index `i` in that sequence, there is an edge in the graph connecting the nodes at positions `i` and `i+1`. A *run* command is defined for this predicate (line 18), which, when executed, returns a satisfying instance.

Just as a predicate can be run for examples, an assertion can be checked for counterexamples. Here we assert that starting from the first node in a Hamiltonian path and transitively following the edges in the graph reaches all other nodes in the graph (lines 13–17). We expect this check (line 19) to return no counterexample.

From the model specification in Listing 6(a), αRby dynamically generates the class hierarchy in Listing 6(c). The generated classes can be used to freely create and manipulate graph instances, independent of the Alloy model.

In Alloy, a command is executed by selecting it in the user interface. In αRby, execution is achieved by calling the `exe_cmd` method. Listing 6(d) shows a sample program that calls these methods, which includes finding an arbitrary satisfying instance for the `hampath` predicate and checking that the `reach` assertion indeed cannot be refuted.

This short example is meant to include as many different language features as possible and illustrate how similar αRby is to Alloy, despite being embedded in Ruby. We discuss syntax in Section 4.5.1; a summary of main differences is given in Table 4.1.

4.4 Beyond Standard Analysis

Sudoku has become a popular benchmark for demonstrating constraint solvers. The solver is given a partially filled $n \times n$ grid (where n must be a square number, so that the grid is perfectly divided into n times $\sqrt{n} \times \sqrt{n}$ sub-grids), and is required to fill the empty cells with integers from $\{1, \ldots, n\}$ so that all cells within a given row, column, and sub-grid have distinct values.

Implementing a Sudoku solver directly in Alloy poses a few problems. A practical one

(a) Graph specification in αRby

```
1  alloy :GraphModel do
2   sig Node [val: (lone Int)]
3   sig Edge [src, dst: (one Node)] {src != dst}
4   sig Graph[nodes:(set Node), edges:(set Edge)]
5
6   pred hampath[g: Graph, path: (seq Node)] {
7    path[Int] == g.nodes and
8    path.size == g.nodes.size and
9    all(i: 0...path.size-1) |
10    some(e: g.edges) {
11     e.src == path[i] && e.dst == path[i+1] }
12  }
13  assertion reach {
14   all(g: Graph, path: (seq Node)) |
15    if hampath(g, path)
16     g.nodes.in? path[0].*((~src).dst)
17    end }
18  run :hampath, 5, Graph=>exactly(1), Node=>3
19  check :reach, 5, Graph=>exactly(1), Node=>3
20 end
```

⇔

(b) Equivalent Alloy specification

```
1  module GraphModel
2  sig Node {val: lone Int}
3  sig Edge {src, dst: one Node}{src != dst}
4  sig Graph{nodes: set Node, edges: set Edge}
5
6  pred hampath[g: Graph, path: seq Node] {
7   path[Int] = g.nodes
8   #path = #g.nodes
9   all i: Int | i >= 0 && i < minus[#path,1] => {
10   some e: g.edges |
11    e.src = path[i] && e.dst = path[plus[i,1]] }
12  }
13  assert reach {
14   all g: Graph, path: seq Node |
15    hampath[g, path] =>
16     g.nodes in path[0].*(~src.dst)
17  }
18 run hampath for 5 but exactly 1 Graph, 3 Node
19 check reach for 5 but exactly 1 Graph, 3 Node
20
```

⇝

```
module GraphModel
 class Node;  attr_accessor :val end
 class Edge;  attr_accessor :src, :dst end
 class Graph; attr_accessor :nodes, :edges end

 def self.hampath(g, path) #same as above end
 def self.reach()          #same as above end
 def self.run_hampath() exe_cmd :hampath end
 def self.check_reach() exe_cmd :reach end
end
```

(c) Automatically generated Ruby classes

```
1  # find an instance satisfying the :hampath pred
2  sol = GraphModel.run_hampath
3  assert sol.satisfiable?
4  g, path = sol["$hampath_g"], sol["$hampath_path"]
5  puts g.nodes # => e.g., {<Node$0>, <Node$1>}
6  puts g.edges # => e.g., {<Node$1, Node$0>}
7  puts path    # => {<0, Node$1>, <1, Node$0>}
8  # check that the "reach" assertion holds
9  sol = GraphModel.check_reach
10 assert !sol.satisfiable?
```

(d) Running `hampath`, checking `reach`

Listing 6: *Hamiltonian Path* example

is that such an implementation cannot easily be used as a stand-alone application, e.g., to read a puzzle from some standard format and display the solution in a user-friendly grid. A more fundamental problem is the inability to express the information about the pre-filled cell values as a partial instance; instead, the given cell values have to be enforced with logical constraints, resulting in significant performance degradation [160]. The αRby solution in Listing 7 addresses both of these issues: on the left is the formal αRby specification, and on the right is the Ruby code constructing bounds and invoking the solver for a concrete puzzle.

Mixed execution. The imperative statements (lines 2, 7, 8) used to dynamically produce a Sudoku specification for a given size would not be directly expressible in Alloy. A concrete Ruby variable `N` is declared to hold the size, and can be set by the user before the specification is symbolically evaluated. Another imperative statement calculates the square root of `N` (line 7); that value is later embedded in the symbolic expression specifying uniqueness within sub-grids (line 13). For illustration purposes, a lambda function is defined (line 8) and used to compute sub-grid ranges (line 15).

Partial instances. Listing 7(b) shows how the bounds are computed for a given Sudoku puzzle (embodied in a Ruby function `pi`, for "partial instance"). Remember that bounds are just tuples (sequences of atoms) that a relation must or may include; since sig-

(a) Sudoku specification in αRby

```
 1 alloy :SudokuModel do
 2  SudokuModel::N = 9
 3
 4  sig Sudoku[grid: Int ** Int ** (lone Int)]
 5
 6  pred solved[s: Sudoku] {
 7   m   = Integer(Math.sqrt(N))
 8   rng = lambda{|i| m*i...m*(i+1)}
 9
10   all(r: 0...N) {
11    s.grid[r][Int] == (1..N) and
12    s.grid[Int][r] == (1..N)
13   } and
14   all(c, r: 0...m) {
15    s.grid[rng[c]][rng[r]] == (1..N)
16   }
17  }
18 end
```

(b) Solving the specification for a partial instance

```
 1 class SudokuModel::Sudoku
 2  def pi
 3   bnds = Arby::Ast::Bounds.new
 4   inds = (0...N)**(0...N) - self.grid.project(0..1)
 5   bnds[Sudoku]         = self
 6   bnds.lo[Sudoku.grid] = self ** self.grid
 7   bnds.hi[Sudoku.grid] = self ** inds ** (1..N)
 8   bnds.bound_int(0..N)
 9  end
10  def solve() SudokuModel.solve :solved, self.pi end
11  def display() puts grid end
12  def self.parse(s) Sudoku.new grid:
13    s.split(/;\s*/).map{|x| x.split(/,/).map(&:to_i)}
14  end
15 end
16 SudokuModel.N = 4
17 s = Sudoku.parse "0,0,1; 0,3,4; 3,1,1; 2,2,3"
18 s.solve(); s.display(); # => {<0,0,1>, <0,1,3>, ...}
```

Listing 7: A declarative *Sudoku* solver using αRby with partial instances

nature definitions in αRby are turned into regular Ruby classes, instances of those classes will be used as atoms. The `Sudoku` signature is bounded by a singleton set containing only the `self` Sudoku object (line 5). Tuples that must be included in the `grid` relation are the values currently present in the puzzle (line 6); additional tuples that may be included are values from `1` to `N` for the empty cells (line 7; empty cell indexes computed in line 4). We also bound the set of integers to be used by the solver; Alloy, in contrast, only allows a cruder bound, and would include all integers within a given bitwidth. Finally, a Sudoku instance can be parsed from a string, and the solver invoked to find a solution satisfying the `solved` predicate (lines 17–18). When a satisfying solution is found, if a partial instance was given, fields of all atoms included in that partial instance are automatically populated to reflect the solution (confirmed by the output of line 18). This particular feature makes for seamless integration of *executable specifications* into otherwise imperative programs, since there is no need for any manual back and forth conversion of data between the program and the solver.

Staged model finding. Consider implementing a Sudoku puzzle generator. The goal is now to find a partial assignment of values to cells such that the generated puzzle has a unique solution. Furthermore, the generator must be able to produce various difficulty levels of the same puzzle by iteratively decrementing the number of filled cells (while maintaining the uniqueness property). With αRby, it takes only the following 8 lines to achieve this with a simple search algorithm on top of the already implemented solver:

```
1 def dec(sudoku, order=Array(0...sudoku.grid.size).shuffle)
2  return nil if order.empty? # all possibilities exhausted
3  s_dec = Sudoku.new grid: sudoku.grid.delete_at(order.first) # delete a tuple at random position
4  sol   = s_dec.clone.solve() # clone so that "s_dec" doesn't get updated if a solution is found
5  (sol.satisfiable? && !sol.next.satisfiable?) ? s_dec : dec(sudoku, order[1..-1])
6 end
7 def min(sudoku) (s1 = dec(sudoku)) ? min(s1) : sudoku end
8 s = Sudoku.new; s.solve(); s = min(s); puts "local minimum found: #{s.grid.size}"
```

The strategy here is to generate a solved puzzle (line 8), and keep removing one tuple

from its `grid` at a time until a local minimum is reached (line 7); the question is which one can be removed without violating the uniqueness property. The algorithm first generates a random permutation of all existing grid tuples (line 1) to determine the order of trials. It then creates a new Sudoku instance with the chosen tuple removed (line 3) and runs the solver to find a solution for it. It finally calls `next` on the obtained solution (line 5) to check if a different solution exists; if it does not, a decremented Sudoku is found, otherwise moves on to trying the rest of the tuples. On a commodity machine, on average it takes about 8 seconds to minimize a Sudoku of size 4 (generating 13 puzzles in total, number of filled cells ranging from 16 down to 4), and about 6 minutes to minimize a puzzle of size 9 (55 intermediate puzzles), number of filled cells ranging from 81 down to 27).

4.5 The αRby Language

αRby is implemented as a domain-specific language in Ruby, and is (in standard parlance) "deeply embedded". *Embedded* means that all syntactically correct αRby programs are syntactically correct Ruby programs; *deeply* means that αRby programs exist as an AST that can be analyzed, interpreted, and so on. Ruby's flexibility makes it possible to create embedded languages that look quite different from standard Ruby. αRby exploits this, imitating the syntax of Alloy as closely as possible. Certain differences are unavoidable, mostly because of Alloy's infix operators that cannot be defined in Ruby.

The key ideas behind our approach are: (1) mapping the core Alloy concepts directly to those of object-oriented programming (OOP), (2) implementing keywords as methods, and (3) allowing mixed (concrete and symbolic) execution in αRby programs.

Mapping Alloy to OOP is aligned with the general intuition, encouraged by Alloy's syntax, that signatures can be understood as classes, atoms as objects, fields as instance variables, and all function-like concepts (functions, predicates, facts, assertions, commands) as methods [15].

Implementing keywords as methods works because Ruby allows different formats for specifying method arguments. αRby defines many such methods (e.g., **`sig`**, **`fun`**, **`fact`**, etc.) that (1) mimic the Alloy syntax and (2) dynamically create the underlying Ruby class structure using the standard Ruby metaprogramming facilities. For an example of the syntax mimicry, compare Listings 6(a) and 6(b)); for an example of metaprogramming, see Listing 6(c).

Note that the meta information that appears to be lost in Listing 6(c) (for example, the types of fields) is actually preserved in separate *meta* objects and made available via the *meta* methods added to each of the generated modules and classes (for example, the following returns the type of the "Graph.nodes" field: `Graph.meta.field("nodes").type`).

Mixed execution, implemented on top of the standard Ruby interpreter, translates αRby programs into symbolic Alloy models. Using the standard interpreter means adopting the Ruby semantics of name resolution and operator precedence (which is inconvenient when it conflicts with Alloy's); a compensation, however, is the benefit of being able to mix symbolic and concrete code. We override all the Ruby operators in our symbolic expression classes to match the semantics of Alloy, and using a couple of other tricks (Section 4.5.3), are able to keep both syntactic (Section 4.5.1) and semantic (Section 4.5.2) differences to

```
spec      ::= "alloy" cname "do" [open*] paragraph* "end"
open      ::= "open" cnameID
paragraph ::= factDecl   | funDecl | cmdDecl | sigDecl
sigQual   ::= "abstract" | "lone"  | "one"   | "some"  | "ordered"
sigDecl   ::= sigQual* "sig"  cname,+ ["extends" cnameID] ["[" rubyHash "]"] [block]
factDecl  ::= "fact"         [fname] block
funDecl   ::= "fun"           fname  "[" rubyHash "]"  "[" expr "]" block
            | "pred"          fname ["[" rubyHash "]"]              block
cmdDecl   ::= ("run"|"check") fname "," scope
            | ("run"|"check") "(" scope ")" block
expr      ::= ID | rubyInt | rubyBool | "(" expr ")"
            | unOp expr         | unMeth "(" expr ")"
            | expr binOp expr   | expr "[" expr "]"    | expr "if" expr
            | expr "." "(" expr ")"                   // relational join
            | expr "." (binMeth | ID) "(" expr,* ")"  // function/predicate call
            | "if" expr "then" expr ["else" expr] "end"
            | quant "(" rubyHash ")" block
quant     ::= "all" | "no" | "some" | "lone" | "one" | "sum" | "let" | "select"
binOp     ::= "||" | "or" | "&&" | "and" | "**" | "&" | "+" | "-" | "*" | "/" | "%"
            | "<<" | ">>" | "==" | "<=>" | "!=" | "<" | ">" | "<=" | ">="
binMeth   ::= "closure" | "rclosure" | "size" | "in?" | "shr" | "<" | ">" | "*" | "^"
unOp      ::= "!" | "~" | "not"
unMeth    ::= "no" | "some" | "lone" | "one" | "set" | "seq"
block     ::= "{" stmt* "}" | "do" stmt* "end"
stmt      ::= expr | rubyStmt
scope     ::= rubyInt "," rubyHash  // global scope, individual sig scopes
ID        ::= cnameID | fnameID
cname     ::= cnameID | '"'cnameID'"' | "'"cnameID"'" | ":"cnameID
fname     ::= fnameID | '"'fnameID'"' | "'"fnameID"'" | ":"fnameID
cnameID   ::= constant identifier in Ruby (starts with upper case)
fnameID   ::= function identifier in Ruby (starts with lower case)
```

Figure 4-1: Core αRby syntax in BNF. Productions starting with: `ruby` are defined by Ruby.

a minimum.

4.5.1 Syntax

A grammar of αRby is given in Figure 4-1 and examples of principal differences in Table 4.1. In a few cases (e.g., function return type, field declaration, etc.) Alloy syntax has to be slightly adjusted to respect the syntax of Ruby (e.g., by requiring different kind of brackets). More noticeable differences stem from the Alloy operators that are illegal or cannot be overridden in Ruby; as a replacement, either a method call (e.g., `size` for cardinality) or a different operator (e.g., ** for cross product) is used.

The difference easiest to overlook is the equality sign: == versus =. Alloy has no assignment operator, so the single equals sign always denotes boolean equality; αRby, in contrast, supports both concrete and symbolic code, so we must differentiate between assignments and equality checks, just as Ruby does.

The tokens for the join and the two closure operators (`.`, ^ and *) exist in Ruby, but have

fundamentally different meanings than in Alloy (object dereferencing and an infix binary operator in Ruby, as opposed to an infix binary and a prefix unary operator in Alloy). Despite this, αRby preserves Alloy syntax for many idiomatic expressions. Joins in Alloy are often applied between an expression and a field whose left-hand type matches the type of the expression (ie, in the form `e.f`, where `f` is a field from the type of `e`). This corresponds closely to object dereferencing, and is supported by αRby (e.g., `g.nodes` in Listing 6(a)). In other kinds of joins, the right-hand side must be enclosed in parentheses. Closures are often preceded by a join in Alloy specifications. Those constructs yield *join closure* expressions of the form `x.*f`. In Ruby, this translates to calling the $*$ method on object `x` passing `f` as an argument, so we simply override the $*$ method to achieve the same semantics (e.g., line 15, Listing 6(a)).

This grammar is, for several reasons, an under-approximation of programs accepted by αRby: (1) Ruby allows certain syntactic variations (e.g., omitting parenthesis in method calls, etc.), (2) αRby implements special cases to enable the exact Alloy syntax for certain idioms (which do not always generalize), and (3) αRby provides additional methods for writing expression that have more of a Ruby-style feel.

<table>
<tr><th>description</th><th>Alloy</th><th>αRby</th></tr>
<tr><td>equality</td><td>x = y</td><td>x == y</td></tr>
<tr><td>sigs and fields</td><td>sig S {
  f: lone S -> Int
}</td><td>sig S [
  f: lone(S) ** Int
]</td></tr>
<tr><td>fun return type declaration</td><td>fun f[s: S]: set S {}</td><td>fun f[s: S][set S] {}</td></tr>
<tr><td>set comprehension</td><td>{s: S | p1[s]}</td><td>S.select{|s| p1(s)}</td></tr>
<tr><td>quantifiers</td><td>all s: S {
  p1[s]
  p2[s]
}</td><td>all(s: S) {
  p1(s) and
  p2(s)
}</td></tr>
<tr><td>illegal Ruby operators</td><td>x in y, x !in y
x !> y
x -> y
x . y
#x
x => y
x => y else z
S <: f, f >: Int</td><td>x.in?(y), x.not_in?(y)
not x > y
x ** y
x.(y)
x.size
y if x
if x then y else z
S.< f, f.> Int</td></tr>
<tr><td>operator arity mismatch</td><td>^x, *x</td><td>x.closure, x.rclosure</td></tr>
<tr><td>fun/pred calls</td><td>f1[x]</td><td>f1(x)</td></tr>
</table>

Table 4.1: Examples of differences in syntax between αRby and Alloy

4.5.2 Semantics

This section formalizes the translation of αRby programs into Alloy. We provide semantic functions (summarized in Figure 4-3) that translate the syntactic constructs of Figure 4-1 to Alloy AST elements defined in Figure 4-2. A store, binding names to expressions or declarations, is maintained throughout, representing the current evaluation context.

Expressions. The evaluation of the αRby expression production rules (`expr`) into Alloy expressions (`Expr`) is straightforward for the most part (Figure 4-4). Most of the

unary and binary operators have the same semantics as in Alloy; exceptions are `**` and **`if`**, which translate to `->` and `=>` (lines 5–11). For the operators that do not exist in Ruby, an equivalent substitute method is used (lines 12–20). A slight variation of this approach is taken for the `^` and `*` operators (lines 21–22), to implement the "join closure" idiom (explained in Section 4.5.1).

The most interesting part is the translation of previously undefined method calls (lines 23–27). We first apply the τ function to obtain the type of the left-hand side expression, and then the $\oplus$ function to extend the current store with that type (line 23). In a nutshell, this will create a new store with added bindings for all fields and functions defined for the range signature of that type (the $\oplus$ function is formally defined in Figure 4-6 and discussed in more detail shortly). Afterward, we look up `meth` as an identifier in the new store (line 24) and, if an expression is found (line 25), the expression is interpreted as a join; if a function declaration is found (line 26), it is interpreted a function call; otherwise, it is an error.

For quantifiers (lines 29-30), quantification domains are evaluated in the context of the current store (using the δ helper function, defined in Figure 4-6) and the body is evaluated (using the β function, defined in Figure 4-5) in the context of the new store with added bindings for all the quantified variables (returned previously by δ).

Blocks. The semantics of αRby blocks differs from Alloy's. An Alloy block (e.g., a quantifier body) containing a sequence of expressions is interpreted as a conjunction of all the constituent constraints (a feature based on Z [151]). In αRby, in contrast, such as sequence evaluates to the meaning of the last expression in the sequence. This was a design decision, necessary to support mixed execution (as in Listing 7(a), lines 7–16). Since Ruby is not a pure functional language, previous statements can affect the result of the last statement by mutating the store, which effectively gives us the opportunity to easily mix concrete and symbolic execution.

This behavior is formally captured in the β function (Figure 4-5). Statements (s_1, ..., s_n) are evaluated in order (line 32). If a statement corresponds to one of the expression rules from the αRby grammar (line 34), it is evaluated using the previously defined $\mathcal{E}$ function; otherwise (line 35), it is interpreted by Ruby (abstracted as a call to the $\mathcal{R}$ functions). Statements interpreted by Ruby may change the store, which is then passed on to the subsequent statements.

Function declarations. The evaluation function (ϕ, Figure 4-5, lines 37–38) is similar to quantifier evaluation, except that the return type is different. The semantics of other function-like constructs (predicates, facts, etc.) is analogous.

Signature declarations. The evaluation function (function ξ, Figure 4-5, lines 39–45) is conceptually straightforward: as before, functions δ and β can be reused to evaluate the field name-domain declarations and the appended facts block, respectively. The caveat is that appended facts in Alloy must be evaluated in the context of Alloy's *implicit this*, meaning that the fields from the parent signature should be implicitly joined on the left with an implicit `this` keyword. To achieve this, we create a variable corresponding to `this` and a new list of fields with altered domains (lines 42–43). A temporary `SigDecl` containing those fields is then used to extend the current store (line 44). A binding for `this` is also added and the final store is used to evaluate the body (line 45). The temporary signature is created just for the convenience of reusing the semantics of the $\oplus$ operator (explained

shortly).

Top-level specifications. Evaluation of an αRby specification (function $\mathcal{A}$, Figure 4-5, lines 46–52) uses the previously defined semantic functions to evaluate the nested signatures and functions. Since declaration order does not matter in Alloy, multiple passes may be needed until everything is resolved or a fixed point is reached (lines 51–52).

Name-domain declaration lists. Name-domain lists are used in several places (for fields, method parameters, and quantification variables); common functionality is extracted and defined in the δ function (Figure 4-6). It simply maps the input list into a list of `VarExpr` expressions, each having $name$ the same as in the declaration list and $domain$ equal to the evaluation of the declared domain against the current store (line 53). It returns that list and the current store extended with those variables (line 54).

Store extension. The $\oplus$ operator (Figure 4-6) is used to extend a store with one or more `VarExpr` or `FunDecl`, a `Type`, and a `Spec`. If a `VarExpr` or a `FunDecl` is given, its name is bound to itself. If a list is given, the operation is folded over the entire list. Extending with a `Type` reduces to extending with the range of that type. Extending with a `SigDecl` means recursively adding bindings for its parent signature, adding a binding for the name of that signature, bindings for all the functions that take that signature as the first argument (an auxiliary function `funs(x)` discovers such functions), and bindings for all its fields. Extending with a `Spec` adds bindings for all the sigs and functions defined in it, including those from all opened specifications.

```
Expr  = VarExpr(name: String, domain: Expr | Type)
      | IntExpr(value: Int)
      | BoolExpr(value: Bool)
      | UnExpr(sub: Expr)
      | BinExpr(lhs: Expr, rhs: Expr)
      | CallExpr(target: Expr, fun: FunDecl, args: Expr*)
      | QuantExpr(kind: String, vars: VarExpr*, body: Expr)
Decl  = Spec(name: String, opens: Spec*, sigs: SigDecl*, funs: FunDecl*)
      | SigDecl(name: String, parent: SigDecl, fields: VarExpr*, inv: FunDecl)
      | FunDecl(name: String, params: VarExpr*, ret: Expr, body: Expr)
Type  = Univ | None | Int | SigDecl | ProductType(lhs: Type, rhs: Type)
Store = {name: String; binding: Expr | Decl}
```

Figure 4-2: Semantic domains. (`Expr` and `Decl` correspond directly to the Alloy AST)

$\mathcal{A}$:	spec → Store→ Spec	$\mathcal{E}$:	expr → Store→ Expr
ξ :	sigDecl → Store→ SigDecl	β:	block → Store→ Expr
ϕ :	funDecl → Store→ FunDecl	δ :	decl∗ → Store→ (VarExpr*, Store)

Figure 4-3: Semantic functions which translate grammar rules to semantic domains

4.5.3 Implementation Considerations

Symbolic Execution. Using the standard Ruby interpreter to symbolically execute αRby programs relieves us from having to keep an explicit representation of the store; instead, the

$\mathcal{E}$: expr → Store→ Expr

1.	$\mathcal{E}[\![\texttt{ID}]\!]\sigma$	≡	$\sigma[\texttt{ID}]$
2.	$\mathcal{E}[\![\texttt{rubyInt}]\!]\sigma$	≡	IntExpr(rubyInt)
3.	$\mathcal{E}[\![\texttt{rubyBool}]\!]\sigma$	≡	BoolExpr(rubyBool)
4.	$\mathcal{E}[\![(e)]\!]\sigma$	≡	$\mathcal{E}[\![e]\!]\sigma$
5.	$\mathcal{E}[\![\texttt{unOp}\ e]\!]\sigma$	≡	UnExpr(unOp, $\mathcal{E}[\![e]\!]\sigma$)
6.	$\mathcal{E}[\![\texttt{unMeth}(e)]\!]\sigma$	≡	UnExpr(unMeth, $\mathcal{E}[\![e]\!]\sigma$)
7.	$\mathcal{E}[\![e_1\ \texttt{**}\ e_2]\!]\sigma$	≡	BinExpr("->", $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
8.	$\mathcal{E}[\![e_1\ \texttt{binOp}\ e_2]\!]\sigma$	≡	BinExpr(binOp, $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
9.	$\mathcal{E}[\![e_1[e_2]]\!]\sigma$	≡	BinExpr("[]", $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
10.	$\mathcal{E}[\![e_1\ \textbf{if}\ e_2]\!]\sigma$	≡	BinExpr("=>", $\mathcal{E}[\![e_2]\!]\sigma$, $\mathcal{E}[\![e_1]\!]\sigma$)
11.	$\mathcal{E}[\![e_1.(e_2)]\!]\sigma$	≡	BinExpr(".", $\mathcal{E}[\![e_1]\!]\sigma$, $\mathcal{E}[\![e_2]\!]\sigma$)
12.	$\mathcal{E}[\![e.\texttt{binMeth}]\!]\sigma$	≡	**match** binMeth **with**
13.			| closure → UnExpr("^", $\mathcal{E}[\![e]\!]\sigma$)
14.			| rclosure → UnExpr("*", $\mathcal{E}[\![e]\!]\sigma$)
15.			| size → UnExpr("#", $\mathcal{E}[\![e]\!]\sigma$)
16.	$\mathcal{E}[\![e.\texttt{binMeth}(a_1)]\!]\sigma$	≡	**match** binMeth **with**
17.			| in? → BinExpr("in", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
18.			| shr → BinExpr(">>>", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
19.			| < → BinExpr("<:", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
20.			| > → BinExpr(":>", $\mathcal{E}[\![e]\!]\sigma$, $\mathcal{E}[\![a_1]\!]\sigma$)
21.			| ^ → $\mathcal{E}[\![e.(a_1.\text{closure})]\!]\sigma$
22.			| * → $\mathcal{E}[\![e.(a_1.\text{rclosure})]\!]\sigma$
23.	$\mathcal{E}[\![e.\texttt{ID}(a_1,\ \ldots)]\!]\sigma$	≡	**let** $\sigma_{sub} = \sigma \oplus \tau(e)$ **in**
24.			**match** σ_{sub}[ID] **as** x **with**
25.			| Expr → BinExpr(".", $\mathcal{E}[\![e]\!]\sigma$, x)
26.			| FunDecl → CallExpr($\mathcal{E}[\![e]\!]\sigma$, x, $\mathcal{E}[\![a_1]\!]\sigma$, ...)
27.			| → $fail$
28.	$\mathcal{E}[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{end}]\!]\sigma$	≡	$\mathcal{E}[\![(e_2\ \textbf{if}\ e_1)\ \textbf{and}\ (e_3\ \textbf{if}\ !e_1)]\!]\sigma$
29.	$\mathcal{E}[\![\texttt{quant}(d_*)\ \texttt{block}]\!]\sigma$	≡	**let** v_*, $\sigma_b = \delta(d_*)\sigma$ **in**
30.			QuantExpr(quant, v_*, $\beta[\![\texttt{block}]\!]\sigma_b$)

Figure 4-4: Evaluation of αRby expressions (`expr` production rules) into Alloy expressions (`Expr`).

store is implicit in the states of the object in which the execution takes place. Having signatures, fields, and functions represented directly as classes, instance variables, and methods, means having most of the bindings (as defined in Section 4.5.2) already in place for all sigs and atoms; for all other expressions, missing methods are dynamically forwarded to the signature class corresponding to the expression's type.

One technical challenge is that the semantics of quantifiers requires a new scope to be created, which, for our syntax, Ruby does not already ensure. Consider the following αRby code: `all(s: S){some s}`. This is just a hash and a block passed to the **`all`** DSL method. When the block is eventually executed (to obtain the symbolic body for this universal quantifier), `s` must be available as a symbolic variable inside of that block. We do that by first dynamically defining a method with the same name in the context of that block, then calling the block, and, finally, redefining the same method to call `super`:

```
ctx = block.binding.eval("self")
ctx.define_singleton_method :s, lambda{VarExpr.new(:s, S)}
begin block.call ensure ctx.define_singleton_method(:s) do super() end end
```

β : block → Store → Expr

31. $\beta[\![$**do** $s_1; \ldots; s_n$ **end**$]\!]\sigma \equiv \beta[\![\{ s_1; \ldots; s_n \}]\!]\sigma \equiv \sigma_{curr} = \sigma, res =$ nil
32. **for** s_i: $\{s_1, \ldots, s_n\}$ **do**
33. **match** s_i **with**
34. | expr → $res \leftarrow \mathcal{E}[\![s_i]\!]\sigma_{curr}$
35. | → $res, \sigma_{curr} \leftarrow \mathcal{R}(s_i)\sigma_{curr}$
36. **return** res

ϕ : funDecl → Store → FunDecl

37. $\phi[\![$**fun** fname$[d_*][e_{ret}]$ block$]\!]\sigma \equiv$ **let** v_*, $\sigma_b = \delta(d_*)\sigma$ **in**
38. FunDecl(fname, v_*, $\mathcal{E}[\![e_{ret}]\!]\sigma$, $\beta[\![$block$]\!]\sigma_b$)

ξ : sigDecl → Store → SigDecl

39. $\xi[\![$**sig** cname **extends** sup $[d_*]$ block$]\!]\sigma \equiv$
40. **let** $s_p = \tau(\sigma[$sup$])$ **in**
41. **let** fld_*, $= \delta(d_*)\sigma$ **in**
42. **let** $this$ = VarExpr("this", cname) **in**
43. **let** $tfld_* = map(\lambda f_i \cdot$ BinExpr(".", $this$, f_i)$, fld_*)$ **in**
44. **let** $\sigma_s = \sigma \oplus$SigDecl(cname, s_p, $tfld_*$, BoolExpr(true)) **in**
45. SigDecl(cname, s_p, fld_*, $\beta[\![$block$]\!]\sigma_s[$"this" $\mapsto this])$

$\mathcal{A}$: spec → Store → Spec

46. $\mathcal{A}[\![$**alloy** cname **do** open∗ paragraph∗ **end**$]\!]\sigma \equiv$
47. **let** $opn_* = map(\lambda$cnameID$\cdot\sigma[$cnameID$]$, open∗$)$ **in**
48. **let** $sig_* = map(\xi, filter($sigDecl, paragraph∗$))$ **in**
49. **let** $fun_* = map(\phi, filter($funDecl, paragraph∗$))$ **in**
50. **let** a = Spec(cname, opn_*, sig_*, fun_*) **in**
51. **if** resolved(a) **then** a
52. **elsif** $\sigma \oplus a \neq \sigma$ **then** $\mathcal{A}[\![$**alloy** cname **do** open∗ paragraph∗ **end**$]\!]\sigma \oplus a$ **else** $fail$

Figure 4-5: Evaluation of blocks and all declarations

δ : decl∗ → Store → (VarExpr∗, Store)

53. $\delta(v_1: e_1, \ldots, v_n: e_n)\sigma \equiv$ **let** $vars = \bigcup_{1<=i<=n}$ VarExpr(v_i, $\mathcal{E}[\![e_i]\!]\sigma$) **in**
54. $[vars, \sigma \oplus vars]$

$\oplus$: Store → *Any* → Store

55. $\sigma \oplus x \equiv$ **match** x **with**
56. | VarExpr(n,_) | FunDecl(n,_) → $\sigma[n \mapsto x]$)
57. | VarExpr∗ | FunDecl∗ → $fold(\oplus, \sigma, x)$
58. | ProductType(_, rhs) → $\sigma \oplus rhs$
59. | SigDecl(n, s_p, fld_*,_) → **let** $\sigma_p = \sigma \oplus s_p$ **in**
60. **let** $\sigma_s = \sigma_p[n \mapsto$ VarExpr$(n, x)]$ **in**
61. $fold(\oplus, \sigma_s,$ funs$(x) + fld_*)$
62. | Spec(n, opn_*, sig_*, fun_*) → $fold(\oplus, \sigma, opn_* + fun_* + sig_*)$
63. | → σ

$\mathcal{R}$	:	rubyStmt → Store → (*Object* → Store)	Executes arbitrary Ruby code
τ	:	Expr → Type	Type of the given expression
funs	:	SigDecl → FunDecl∗	Functions where given sig is first param
resolved	:	Spec → Bool	Whether all references are resolved

Figure 4-6: Helper functions.

Responding to Missing Methods and Constants. To avoid requiring strings instead of identifiers for every new definition (e.g., `sig :Graph` instead of `sig Graph`, where `Graph` is previously undefined), αRby overrides `const_missing` and `method_missing` and instead of failing returns a `MissingBuilder` instance. Furthermore, `MissingBuilder` instances also accept a block at creation time, and respond to several operator methods, making constructs like `fun f[s: S][set S] {}` possible. To guard against unintended conversions (e.g., typos), αRby raises a syntax error every time a `MissingBuilder` is not "consumed" (by certain DSL methods, like `sig` and `fun`) by the end of its scope block.

Online Source Instrumentation. For the purpose of symbolic evaluation, the source code of every αRby function/predicate is instrumented before it is turned into a Ruby method. The need for instrumentation arises because certain operators and control structures, which we would like to treat symbolically, cannot be overridden; examples include all the if-then-else variants, as well as the logic operators. Our instrumentation uses an off-the-shelf parser and implements a visitor over the generated AST to replace these constructs with appropriate αRby expressions (e.g., `x if y` gets translated to `BinExpr.new(IMPLIES, proc{y}, proc{x})`). This "traverse and replace" algorithm is far simpler than implementing a full parser for the entire Alloy grammar.

Distinguishing Equivalent Ruby Constructs. Ruby allows different syntactic constructs for the same underlying operation. For example, some built-in infix operators can be written with or without a dot between the left-hand side and the operator (e.g., `a*b` is equivalent to `a.*b`). Since αRby already performs online source instrumentation, it additionally detects the following syntactic nuances for the purpose of assigning different semantics: (1) in Ruby, "<b2> **if** <b1>" is equivalent to "`b1 and b2`", but our instrumenter always rewrites `and` and `or` to boolean conjunction and disjunction; (2) when prefixed with a dot, operators `*`, `<` and `>` are translated to join closure, domain restriction, and range restriction, respectively (`.*`, `<:`, and `:>` in Alloy).

αRby to Alloy Bridge. All model-finding tasks are delegated to a slightly modified version of our Alloy* Java implementation. The main modification made was adding an extra API method, which additionally accepts a partial instance (represented in a simple textual format independent of the Alloy language). The Alloy Analyzer already has a complex heuristic for computing bounds from the scope specification and certain (automatically detected) idioms; all those features are retained, and on top of them the αRby-provided partial instance is used to shrink the bounds further. To interoperate between Ruby and Java, RJB [138] is used, which conveniently automates most of the process.

4.6 Discussion

The technical part of this chapter described a number of Ruby techniques for designing convincingly looking DSLs. In addition to syntax, however, some common ground has to be found between the DSL and the host language to make the integration smooth, and thus, the whole embedding transparent—that in particular is the key reason we argued that αRby is more than just a neat API to Alloy. Furthermore, we believe that much of the discussions generalize to broader domains, and could be applied to a different pair of specification and implementation language.

Unfortunately, however, this integration is not perfect. We have mentioned several cases where, because of the limitations of the Ruby parser, a generic syntax concept in Alloy is implemented only as a special case idiom in αRby (e.g., the ~, ^, * Alloy relational operators). This often creates confusion among users who are already used to the Alloy syntax. Other small peculiarities of the Ruby parser, which come handy in some cases (like allowing in certain cases parenthesis around function call arguments to be omitted), in the general case break the uniformity of the language, meaning that the user has to learn many special cases instead of a few general rules.

αRby relies heavily on Ruby's metaprogramming facilities to imitate Alloy's pure relational nature. That means that all field values of Ruby classes representing αRby sigs are wrapped to appear as sets of tuples. While an Alloy user might not object this behavior, a Ruby user is probably more used to scalar values and thus prefers to think about αRby sigs as regular Ruby classes. For that reason, αRby performs implicit coercion whenever possible to meet expectations of both sides; again, however, this cannot be implemented uniformly so corner cases where manual conversion is necessary typically come as a big surprise. Finally, the heavy use of metaprogramming sometimes incurs unexpected performance penalties (only during the translation to Alloy phase, not during solving), especially when dealing with large αRby models.

4.7 Related Work

Montaghami and Rayside extended the Alloy language with special syntax for specifying partial instances [120]. They argue convincingly for the importance of having partial instances for Alloy, giving use cases such as test-driven model development, regression testing of models, modeling by example etc. They also provide experimental evidence that staged model finding can lead to better scalability. Their approach is limited to partial instances only, and it does not provide any scripting mechanisms for automating such tasks. Thus to carry out their staged model finding experiments, after obtaining an instance in the first stage, they manually inspected it (e.g., in the visualizer), rewrote it using the new syntax, and then solved in the second stage. Using αRby would automate the whole process, since an αRby instance can provide a set of exact bounds for all included relations, and can handle all the use cases discussed.

The same authors provide another extension of the Alloy language to allow (for performance reasons) certain idiomatic specifications to be evaluated in two stages [121]. Their extension implements additional semantics, specific to their syntax extension and the chosen idiom, so it is not entirely subsumed by αRby's out-of-the-box staged model finding functionality. However, we argue that it is much simpler to implement such an extension in αRby than on top of the Alloy Java API.

A number of tools built on top of Alloy have implemented (often in an ad hoc fashion) one or more features that can now be provided by αRby. Aluminum [130] implements an interesting heuristic for minimizing Alloy instances and by default showing the minimal one first. It also allows the user to augment the current instance by selecting one or more tuples to be included in the next instance. αRby provides a more generic mechanism that lets the user provide an arbitrary formula (possibly involving atoms from the current in-

stance) to be satisfied in the next solution. TACO [60] is a bounded verifier for Java that achieves scalability by relying heavily on the Alloy Analyzer to recognize certain idioms as partial instances; we believe αRby would have made their implementation much simpler.

Our mixed execution was inspired by Rubicon's [129] symbolic evaluator, which also uses the standard Ruby interpreter. Unlike αRby, Rubicon stubs the library code with custom expressions in order to symbolically execute and verify existing web apps.

Many research projects explore the idea of extending a programming language with symbolic constraint-solving features (e.g., [90,118,141,149,161,169]). αRby can be understood as a kind of dual, with the opposite goal. While these efforts aim to bring declarative features in imperative programming, αRby aims to bring imperative features to declarative modeling. Although the basic idea of combining declarative model finding and imperative model finding is shared, the research challenges are very different. As this chapter has explained, αRby addresses the challenge of embedding an entire modeling language in a programming language, whereas these related projects instead tend to use a constraint language that is only a modest extension of the programming language's existing expression sublanguage. αRby also addresses the challenge of reconciling two different views of a data structure: one as objects on a heap, and the other as relations (and in this respect is related to work on relational data representation, such as [73]).

4.8 Discussion

The technical part of this chapter described a number of Ruby techniques for designing convincingly looking DSLs. In addition to syntax, however, some common ground has to be found between the DSL and the host language to make the integration smooth, and thus, the whole embedding transparent—that in particular is the key reason we argued that αRby is more than just a neat API to Alloy. Furthermore, we believe that much of the discussions generalize to broader domains, and could be applied to a different pair of specification and implementation language.

Unfortunately, however, this integration is not perfect. We have mentioned several cases where, because of the limitations of the Ruby parser, a generic syntax concept in Alloy is implemented only as a special case idiom in αRby (e.g., the ~, ^, * Alloy relational operators). This often creates confusion among users who are already used to the Alloy syntax. Other small peculiarities of the Ruby parser, which come handy in some cases (like allowing in certain cases parenthesis around function call arguments to be omitted), in the general case break the uniformity of the language, meaning that the user has to learn many special cases instead of a few general rules.

αRby relies heavily on Ruby's metaprogramming facilities to imitate Alloy's pure relational nature. That means that all field values of Ruby classes representing αRby sigs are wrapped to appear as sets of tuples. While an Alloy user might not object this behavior, a Ruby user is probably more used to scalar values and thus prefers to think about αRby sigs as regular Ruby classes. For that reason, αRby performs implicit coercion whenever possible to meet expectations of both sides; again, however, this cannot be implemented uniformly so corner cases where manual conversion is necessary typically come as a big surprise. Finally, the heavy use of metaprogramming sometimes incurs unexpected perfor-

mance penalties (only during the translation to Alloy phase, not during solving), especially when dealing with large αRby models.

4.9 Conclusion

On the one hand, αRby addresses a collection of very practical problems in the use of a model finding tool. The contribution can thus be regarded as primarily architectural, in demonstrating a different way to build an analysis tool that uses a DSL embedding to allow end-user scripting, rather than a closed compiler-like tool that can be extended only by one of the tool's developers.

On the other hand, αRby suggests a new way to think about a modeling language. The constructs of the language are not treated as functions that generate abstract syntax trees only in a mathematical sense, but are implemented as these functions in a manner that the end user can exploit. This leads us to wonder whether it might be possible to use this style of embedding in the very design of the modeling language. Perhaps, had this approach been available when Alloy was designed, an essential core might have been more cleanly separated from a larger collection of structuring idioms, implemented as functions on top of the core's functions.

Practically speaking, the hope is that the developers of tools that use Alloy as a backend will be able to use αRby in their implementations, at the very least making it easier to prototype new functionality. And perhaps the implementors of tools for other declarative languages will find ideas here that they can exploit in similar embeddings.

Part III

Declarative Programming for the Web

Chapter 5

SUNNY: Model-Based Paradigm for Programming Reactive Web Applications

Parts I and II were primarily concerned with different aspects of declarative programming based on executable specifications, where, at any point in a program, a full functional specification (often expressed in a formal logic) can be asserted against the program state and be treated like any other executable program statement. Automatically solving for such a declarative statement necessitates a computationally complex algorithm; when this is done via a translation to SAT (as in Alloy*), every such execution is NP-complete. Despite the high complexity and the associated runtime cost, we argued that for many practical purposes this approach is not only adequate, but also competitive with hand-optimized solutions. In many other cases, however, this approach still, unfortunately, turns out to be intractable, hindering its wider adoption. Another (unrelated) criticism of the idea of executable specifications is that, for most programmers, it is hard to learn formal logic and effectively write full functional specifications.

In this chapter we explore an alternative form of declarative programming, applied to a specific domain, which avoids functional logic-based specifications and does not require an expensive solver at runtime, but nevertheless, aims to provide similar benefits to the programmer (saying what instead of how). The chosen domain is that of web application programming. This is a particularly suitable domain, since building even the simplest web applications seems to be more difficult than necessary, mostly due to a multitude of involved technologies and big abstraction gaps between them, as well as a number of common programming problems arising from concurrency, data distribution/consistency/serialization, event handling, synchronizations, etc.

We propose SUNNY, a model-based, event-driven, policy-agnostic paradigm for developing reactive web applications. SUNNY allows a programmer to represent a distributed application as if it were a simple sequential program, with atomic actions updating a single, shared global state. SUNNY imposes a clear separation between four main (often cross-cutting) concerns of web applications: (1) data model, (2) network model, (3) event model, and (4) security model. The programmer specifies each model, in a style similar to object-oriented programming, separately and independently from each other. A run-

time environment executes the program on a collection of clients and servers, automatically handling (and hiding from the programmer) complications such as network communication (including server push), serialization, concurrency and races, persistent storage of data, and queuing and coordination of events.

5.1 Motivation

Today's era of social networks, online real-time user collaboration, and distributed computing brings new demands for application programming. Interactiveness and multi-user experience are essential features of successful and popular applications. However, programming such inherently complex software systems, especially when the interactive (real-time) multi-user component is needed, has not become much easier. Reasons for this complexity are numerous and include:

- the *distributed architecture* of multiple servers running on the cloud (server farms) interacting with clients running on different platforms (e.g., smartphones, web browsers, desktop widgets, etc.);
- the *abstraction gap* between the problem-domain level (high-level, often event-driven) and the implementation-level (low-level messages, queues, schedulers, asynchronous callbacks);
- shared *data consistency*;
- *concurrency* issues such as data races, atomicity violations, deadlocks, etc.

Problems of this kind are known as *accidental complexity* [31], since they arise purely from abstraction mismatches and are not essential to the actual problem being solved. Carefully managing accidental complexity, however, is absolutely crucial to developing a correct and robust system. Although thoroughly studied in the literature, these problems not only pose serious challenges even for experienced programmers, but also distract the programmer from focusing on essential problems, i.e., designing and developing the system to achieve its main goals.

This thesis proposes a new *model*-based programming paradigm for designing and developing interactive event-driven systems, accompanied by a runtime environment for monitored execution of programs written in that language. Our paradigm is structured around models (mostly declarative, but fully executable) using concepts from the domain of interactive web applications, (e.g., shared data, system events, interactions and interconnections between clients, etc.), and also explicitly separating concerns like data, core business logic, user interface, privacy and security rules, etc. This allows the programmer to think and write code at a high-level, close to the actual problem domain, directly addressing the abstraction gap issue.

The structural information about the system, which is inherently present in these models, allows the runtime environment to automatically manage many forms of accidental complexity, from synchronizing and dispatching concurrent events to propagating data updates to all connected clients (also known as "server push" in the web developers community). The programmer, therefore, has a very simple sequential programming view, and it is the job of the runtime environment to turn that into a distributed application. Relieving the programmer of writing multithreaded code eliminates, by construction, a whole class of concurrency bugs, which are notoriously difficult to debug and fix.

We call this whole approach SUNNY, as our goal is to shine some light on the dark world of distributed systems, making it less tedious and more fun, and, at the same time, more

robust and more secure. In support of this approach, two concrete implementations were developed: (1) RED [110] (Ruby Event Driven, implemented on top of Ruby on Rails), and (2) *Sunny.js* [111] (a pure JavaScript implementation on top of Meteor). The purpose of the former is to demonstrate how the SUNNY concepts can be achieved across the full web stack (including a relational database); the latter additionally takes advantage of the "thick client", a NoSQL database, and "client-side rendering" technologies to improve scalability and responsiveness.

5.2 Example

In this section we present a simple example of a real-world application to explain the proposed programming paradigm and illustrate the expressiveness and ease of use of the SUNNY language.

This example implements a "public IRC" (Internet Relay Chat) web application, in which anyone can create a chat room and the existing rooms are public (anyone can join and send messages once joined). With most applications of this kind, the web GUI must be responsive and interactive, automatically refreshing parts of the screen whenever something important happens (e.g., a new message is received), without reloading the whole page.

Listing 8 shows a simple IRC implementation written in RED (our implementation of SUNNY for Ruby on Rails). RED programs consist of several different *models* of the system (described next), and as such are fully executable. These models are fairly high-level and mostly declarative, so we occasionally refer to them as *specifications*, even though they are not full functional specifications in the traditional sense.

The **data model** of the IRC application (Listing 8(a)) consists of a `User` record (which specializes the RED library `AuthUser` record and adds a status field), a `Msg` record (where each message has a textual body and a sender), and a `ChatRoom` record (each room has a name, a set of participating users, and a sequence of messages that have been sent). These fields are defined using the `refs` and `owns` keywords: the former denotes aggregation (simple referencing, without any constraints), and the latter denotes composition (implying that (1) when a record is deleted, all owned records should be deleted, and (2) no two distinct records can point to the same record via the same owned field).

The **network model** in this example (Listing 8(b)) consists of two machines, namely `Server` and `Client`. The `Client` machine has a corresponding `User`, whereas the `Server` machine maintains a set of active `ChatRooms`. They respectively inherit from the library `AuthClient` and `AuthServer` machines, to bring in some fairly standard (but library-defined, as opposed to built-in) user management behavior, like new user registration, sign-in and sign-out events, etc.

To implement the basic functionality of IRC, we defined an **event model** with three event types: `CreateRoom`, `JoinRoom`, and `SendMsg`, as shown in Listing 8(c).

Each event has an appropriate precondition (given in the `requires` clause) that checks that the requirements for the event are all satisfied before the event may be executed. For instance, events `CreateRoom`, `JoinRoom`, and `SendMsg` all require that the user has signed in (`client.user` is non-empty), `SendMsg` requires that the user has joined the room, etc.

A specification of the effects of an event (given in the `ensures` clause) is concerned

(a) data model

```
record User < AuthUser do
 # inherited fields
 #   name: String,
 #   email: String,
 #   pswd_hash: String,
 refs status: String
end
```

```
record Msg do
 refs
   text: Text,
   sender: User
end
```

```
record ChatRoom do
 refs
   name: String,
   members: (set User)
 owns
   messages: (seq Msg)
end
```

(b) network model

```
machine Client < AuthClient do
  refs user: User
end

machine Server < AuthServer do
  owns rooms: (set ChatRoom)
end
```

(c) event model

```
event CreateRoom do
 from client: Client
 to   serv: Server

 params roomName: String

 requires {
  client.user &&
  roomName    &&
  roomName != ""
 }

 ensures {
  room = ChatRoom.create
  room.name = roomName
  room.members = [client.user]
  serv.rooms << room
 }
end
```

```
event JoinRoom do
 from client: Client
 to   serv: Server

 params room: ChatRoom

 requires {
  u = client.user
  client.user &&
    !room.members.include?(u)
 }

 ensures {
   u = client.user
   room.members << u
 }
end
```

```
event SendMsg do
 from client: Client
 to   serv: Server

 params room: ChatRoom,
        msgText: String

 requires {
  client.user &&
  room.members.include?(client.user)
 }

 ensures {
  msg = Msg.create
  msg.text = msgText
  msg.sender = client.user
  room.messages << msg
 }
end
```

(d) security model

```
policy HideUserPrivateData do
  principal client: Client

  # restrict access to passwords
  restrict User.pswd_hash.unless do |user|
    client.user == user
  end

  # restrict access to status messages to users
  # who share at least one chat room
  # with the owner of that status message
  restrict User.status.when do |user|
    client.user != user &&
    ChatRoom.none? { |room|
      room.members.include?(client.user) &&
      room.members.include?(user)
    }
  end
end
```

```
policy FilterChatRoomMembers do
  principal client: Client

  # filter out anonymous users (those who have not
  # sent anything) from the 'members' field
  restrict ChatRoom.members.reject do |room, user|
    !room.messages.sender.include?(user) &&
    client.user != user
  end
end
```

Listing 8: A full implementation (excluding any GUI) of a simple public IRC application written in RED

only with updating relevant data records and machines to reflect the occurrence of that event. For example, the effects of the `JoinRoom` event amount to simply adding the user requesting to join the room to the set of room members; the runtime system will make sure

that this update is automatically pushed to all clients currently viewing that room. Bindings between the GUI elements and the data model are specified elsewhere (in GUI templates, to be exact), independently of the event model; this is a key to achieving separation of concerns.

By default, all fields in our models are public and visible to all machines in the system. That approach might be appropriate for the running "public IRC" example, where everything is supposed to be public anyway. For many other systems, however, it is often necessary to restrict access to sensitive data. Let us therefore define some privacy rules even for this example to show how that can be done in SUNNY, declaratively and independently of the event model.

The `HideUserPrivateData` policy from Listing 8(d) dictates that the value of a user's password should not be revealed to any other user and, similarly, that the status message of a user should not be revealed to any other user, unless the two users are currently both members of the same chat room. Note that the latter rule is dynamic, i.e., it depends on the current state of the system (two users being together in a same chat room) and thus its evaluation for two given users may change over time.

In addition to restricting access to a field entirely, when a field is of a collection type, a policy can also specify a filtering condition to be used to remove certain elements from that collection before the collection is sent to another machine. The `FilterChatRoomMembers` policy hides those members of a chat room who have not sent any messages (this simulates, to some extent, "invisible users", a feature supported by some chat clients).

SUNNY automatically checks policies at every field access; if any policy is violated the access is forbidden simply by replacing the field value with an empty value.

5.3 What is Different About SUNNY

Interactive multi-user applications, even when having relatively simple functional requirements, are difficult to write using today's programming languages and available state-of-the-art frameworks, the main reason being the abstraction gap between the problem domain and the concepts available at the implementation level.

Just as one example, current systems typically do not offer much help with structuring and organizing the system around events, despite proper event handling being at the core of most interactive applications. Instead, they offer callbacks, which can be registered from any source code location, almost inevitably leading to what is known as *callback hell* [49]. As a consequence, programs end up being cluttered, the flow structure becomes very difficult to infer from the source code, leading to programs that are hard to understand and maintain.

Other than event-handling, the programmer has to face a number of other technological barriers, including concurrency, object-relational mapping, server push, etc. Even though these technological barriers have been individually overcome, the solutions sometimes come in a form of best-practices or guidelines, so the programmer still has to spend time implementing them for the project at hand, which is, for the barriers mentioned above, time-consuming, tedious, and also error-prone.

We illustrate these points in terms of three concrete platforms for developing web applications.

5.3.1 The Java Approach

The Java language, which gained much of its success from being proposed as a platform for web development, is still one of the top choices for development of enterprise web systems. The language being mature, the runtime (JVM) being fast and solid, and an abundance of freely available third-party libraries are some of the points in favor.

The trend of web development in Java still seems to be based around manually configuring and integrating a multitude of standalone, highly specialized libraries, designed independently to solve various web-related tasks, as opposed to having a single overarching framework designed to address most of the common issues. A highly experienced Java expert, who is already familiar with the existing libraries for web development, object-relational mapping, database management, server push, and such (also already knowing how to configure them all so that they can interoperate and work well together) would have a good start developing our IRC example. For the rest of us, however, the situation is much worse. For someone already familiar with Java, but not too familiar with web development in Java, the effort just to get a handle on all the necessary libraries would by far exceed the effort needed to implement the functionality of our example.

Even the expert would have to be very careful about managing concurrent requests on the server side, setting up event processing queues (to avoid common concurrency issues), implementing corresponding producer and consumer threads, and so on. Probably equally cumbersome would be manually keeping track of which clients are viewing what, automatically propagating updates when the data underlying the views change, and implementing Ajax-style code on the client side to refresh the GUI smoothly. All these are generic enough tasks, for which the implementation does not seem to differ much from one project to another, so it seems unfortunate that they have to be repeated every time. One of the design goals of SUNNY was to explicitly address this issue, and let the framework, not the programmer, fight the technology.

5.3.2 The Rails Approach

In contrast to Java, the design of Rails [6] adopted the "convention over configuration" school of thought: instead of manually configuring every single aspect of the application, if certain conventions are followed, the Rails framework will automatically perform most of the boilerplate tasks behind the scene and "magically" make things happen.

Underneath the surface, unfortunately, it is still a configuration mess, and the magic is mostly concerned with low-level configuration of different components and how to tie them all together. This creates problems for many Rails programmers, because, as this magic has no high-level semantics, it is often difficult to understand and remember not only how it works, but also what it does. In SUNNY, we aim to offer a different kind of magic, which is easy to understand at the conceptual level (e.g., data updates are automatically propagated to clients, all the way to automatically refreshing the GUI), so the programmer need not understand the technical details behind its implementation.

By imposing some structure on how the system should be organized and implemented (e.g., using the Model View Controller (MVC) architecture), Rails can indeed provide a lot of benefits for free. One of the most appealing features of Rails (especially back when it first appeared) is "scaffolding": given just a few model files describing how the data structures are organized, Rails automatically generates a running web application, with the full stack, from the database to the web server, automatically configured and set up.

While scaffolding greatly reduces the startup cost of developing a new application (even for inexperienced programmers), it is not meant to be a permanent, system-level solution. The reason is that it is based on code generation from *transient* models: the generated files (including database configuration files, Rails controller classes, HTML views) work fine at the beginning, but as soon as something needs to be changed, everything needs to be changed manually, since there is nothing to keep them in sync otherwise. Furthermore, the models used for scaffolding support only scalar, primitive-typed fields. In SUNNY, in contrast, models (like those shown in Listing 8) are first-class citizens; not only do they exist at runtime, but they are central to the whole paradigm (i.e., the entire runtime semantics is built around them). Our models are also much richer, so there is enough information available to the SUNNY runtime environment to interpret them on the fly, instead of generating code up front. That way, the common problem of having inconsistencies between the models and the code is eliminated in SUNNY.

Concurrency in Ruby is an interesting topic. Ruby is inherently not concurrent (because of a Global Interpreter Lock). As a result, Rails programmers can safely ignore threads and synchronization, and still have no data race issues. This, of course, comes at the cost of low scalability. When a more scalable implementation is needed, typically solutions require that the system is restructured so that blocking operations (like I/O) are offloaded to a different process, which is at the same time told what to do upon completion of the requested operation (the so called *Reactor* pattern). Refactoring a system in this manner is almost never trivial nor straightforward.

We believe that concurrency and parallel processing do not have to be sacrificed to this extent to give the programmer a safe sequential programming model, as explained in more detail in Section 5.4.1.

5.3.3 The Meteor Approach

Meteor [4] is a newer web framework for fast and convenient development of modern web applications. Meteor has been rapidly gaining popularity. It is a pure JavaScript implementation (both server and client have to be written in JavaScript) of an event-driven (publish/subscribe) system which also automatically propagates updates to all connected clients whenever the shared data changes.

Unlike SUNNY, Meteor focuses on providing a platform for automatic data propagation, whereas SUNNY is designed to also handle other aspects of the system, including richer models for shared data, GUI scaffolding, automated support for concurrency, etc. Specifically, Meteor does not offer much structure to help design the system, nor does it have rich models of the underlying shared data. The data model in Meteor consists of a number of flat collections (corresponding directly to database tables), with no type information, and no explicit relationship between different model classes. Rich models enable

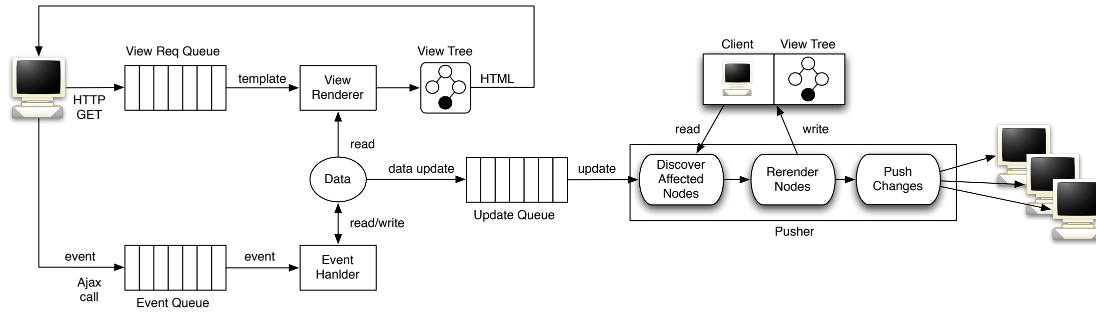


Figure 5-1: Internal architecture of SUNNY's runtime environment for concurrent processing of events and user requests.

both software engineering benefits (like automated test generation and verification of end-to-end properties), as well as productivity benefits (like automated GUI scaffolding)[1].

5.4 The SUNNY Approach

A key idea of SUNNY is to make it possible to think about different events in isolation, and only in terms of modifications to the data model they entail. Therefore, in the design phase, the programmer does not have to think about other issues, such as how to update the user interface to reflect the changes, or even about security and privacy policies; those can be specified separately and independently from the core event model. Limiting the specification this way is what forces the programmer to focus on the core logic of the system first (hence reducing the chances of software bugs in those core parts of the system) and what enables us to provide a unified and overarching runtime environment for fully automated resource management and constant data access monitoring for security violations.

The main components of SUNNY are: (1) a Domain Specific Programming Language (2) a Runtime Environment (3) an Online Code Generator (4) a Dynamic Template-Based Rendering Engine The following subsections walk through a sample execution of our system (still using the running IRC example) to better illustrate how the system works and how the benefits are achieved.

5.4.1 Sample Execution

Consider a scenario in which a user initially opens the home page of our IRC application. This request is received by the web server via the HTTP GET method and placed in a processing queue (namely *View Req Queue*, Figure 5-1, top pipeline). From there, it is picked up by the *View Renderer* component, while the web server can immediately go back to serving incoming requests.

Let us assume that the view corresponding to the home page is the *irc* template shown in Listing 9(a) and that the user is not logged in yet. These templates are written in the ERB

[1]In contrast to GUI scaffolding implemented in Rails, ours is not a one-off code generation approach; it is rather based on generic (application-agnostic) templates which get evaluated at runtime, so again, there is no problem of falling out of sync.

(a) template file: *irc.html.erb*

```
<% if client.user %>
 <%= render client.user %>
 <%= render server.rooms, :as => 'room' %>
<% else %>
 <form id="login-form">
  Email:    <input type="text" name="email"/>
  Password: <input type="password"
                   name="password"/>
 </form>
 <button data-trigger-event="SignIn"
         data-params-form="login-form">
   Sign In</button>
<% end %>
```

(b) template file: *user.html.erb*

```
<div class="User">
 Welcome <%= user.name %> (<%= user.email %>)
</div>
```

(c) template file: *chat_room.html.erb*

```
<div class="ChatRoom">
 Name:    <%= room.name %>
 Members: <%= room.members.name.join(", ") %>
 Posts:   <%= render room.messages %>


 <input id="txt-<%=room.id%>" type="text"/>
 <button
   data-trigger-event="SendMsg"
   data-param-room="${new ChatRoom(<%=room.id%>)}"
   data-param-msgText="${$('#txt-<%=room.id%>').val()}">
  Send</button>
</div>
```

(d) template file: *msg.html.erb*

```
<div class="post">
 <%= msg.sender.name %>: <%= msg.text %>
</div>
```

Listing 9: ERB template views for the IRC example from Listing 8

language, which allows arbitrary Ruby expressions to be embedded inside the `<% %>` and `<%= %>` marks (the difference being that only the latter produces a textual output, while the output of the former is ignored). The *View Renderer*, therefore, evaluates the "else" branch of the template, and returns a login page with two input fields (for email and password) and a "Sign-in" button.

While rendering a template, the *View Renderer* also maintains a *View Tree* structure which holds a single node for each Ruby expression that was evaluated during the execution of the template (templates can invoke other templates, potentially creating a hierarchy of nodes). Each node stores a list of fields that were accessed while the corresponding expression was being evaluated. In the case of this example, there is only one node in that tree, and the only field that was accessed was the `user` field of the current client instance (during the evaluation of the "if" condition).

On the client side, our JavaScript library automatically recognizes the "Sign-in" button by its `data-trigger-event` HTML5 attribute, and, according to its value, associates it with the `SignIn` event (which is a part of the previously imported `Auth` library. More concretely, it assigns an "onclick" handler to it, so that when the button is clicked, the associated form (discovered via the `data-params-form` attribute) is submitted (via an Ajax call) as the parameters of the `SignIn` event.

When the user clicks this button, the `SignIn` event is triggered and received on the server side via the bottom processing pipeline in Figure 5-1. The *EventHandler* then picks it up from the queue, checks its precondition, and if it holds (in this case it does, since the `requires` method is empty), proceeds to execute its postcondition. Assuming that the user entered valid credentials, the execution will assign value to the `user` field of the current `client` instance (the `client` instance is always implicit and denotes the machine which submitted the currently executing event).

Any modification to the data model triggers an internal "data update" signal, which is

placed in the *Update Queue* (the right-most pipeline in Figure 5-1). A component called *Pusher* is in charge of serving the *Update Queue*. Every time an update is received, it goes through a list of all connected clients and corresponding view trees, discovers which nodes could potentially be affected by the current update (by checking their list of field accesses), re-renders those nodes, updates the global *Client* → *View Tree* map, and pushes those changes to clients. On the client side, only the corresponding portion of the HTML DOM is replaced by the newly rendered text.

In the running scenario, the only node that was stored for the current client was dependent on the `user` field, so only it has to be re-rendered. The new content is produced by executing the "then" branch, which amounts to rendering the *user.html.erb* template for the current user (the user object is by default available to the callee template via the `user` variable), and rendering the *chat_room.html.erb* template once for each room on the server (in this case the default variable name would be "chat_room", but it is instead explicitly set to "room" via the `:as` option).

The execution then continues in the same manner: clients continue to perform actions by triggering events from the domain, and the server keeps processing events, detecting changes in the data model, and re-rendering parts of the client views when needed. An explanation of how asynchronous message sending is declaratively specified directly in an HTML template (no separate JavaScript file), and without any Ajax code, is given in Section 5.4.4.

To get a running version of this sample execution, if using RED, the programmer only needs to:

- write the data, machine, and event models from Listing 8 (the security model is not necessary);
- write the HTML templates from Listing 9;
- deploy the application to a server running Ruby on Rails with our extensions; and
- set the application home page to *irc.html.erb* (by configuring the root route in Rails).

Comparing to implementing the same application in standard Rails:

- in place of our data model, the programmer would write ActiveRecord model classes (one model class per record), which are more verbose and require more configuration (as discussed in Section 5.4.4);
- in place of our machine model, the programmer would likely use in-memory classes and the Rails session storage (not affecting the complexity of the implementation);
- in place of our event model, the programmer would write controllers of approximately the same complexity;
- the HTML templates would remain the same, as well as the deployment process.

Additionally, the Rails programmer would have to

- write a database schema (discussed in Section 5.4.4), carefully following the Rails naming convention;
- write a controller for each model class implementing the standard CRUD (Create, Read, Update, Delete) operations (again, certain naming conventions have to be followed);
- configure routes for each controller;
- decide on a third party library to use to implement the server push (pushing data updates to connected clients in real time);
- implement server-side code that keeps track of what data each client is currently displaying;
- implement server-side code that detects model changes (made during the execution of controllers);
- implement server-side code that pushes data changes to each client whenever a piece of data currently being displayed on that client is changed;
- implement client-side code that listens for data changes from the server;
- implement client-side code that dynamically re-renders affected parts of the GUI whenever a data update is received.

In both cases, a CSS file is necessary in order to make the GUI look pretty.

While RED provides dynamic GUI updates for free, and for that does not require the programmer to write any JavaScript, it does not prevent him or her from doing so; RED comes with a client-side JavaScript library (see Section 5.4.4) which can be used to interact with the server-side, customize how the GUI gets updated (e.g., implement special visual effects or animations), asynchronously trigger events, etc.

5.4.2 Domain-Specific Programming Language

We designed a domain-specific language for writing SUNNY models in order to better emphasize the key concepts of our paradigm. This language has strong foundations in the Alloy modeling language [78], a *relational* language based on first-order logic. Alloy is declarative, and as such, it is not executable per se; it is instead used primarily for modeling and checking logical properties of various kinds of systems. Most of Alloy's expressive power comes from its relational base (including all the supported relational operators), which, however, can be efficiently executable in the context of object-oriented programs [137]. For example, the dot operator ('`.`') is actually a relational join, so an expression that fetches all users currently present in any chat room on a server can be written simply as `Server.rooms.members`.

In RED, we implemented this domain-specific language as an adaptation of αRby (presented in Chapter 4), which makes it deeply embedded in Ruby. Concretely, each of `record`, `machine`, and `event` is just a function that takes a name, a hash of field $name \rightarrow type$ pairs,

and a block; it (1) returns a `Class` having those field names as attributes, while storing the type information in a separate meta-class, (2) creates, in the current module, a constant with the given name and assigns the newly created class to it, and (3) if a block is given, evaluates that block in the context of that class. The block parameter can be used to define additional instance methods, but also to define fields with more information than just name and type (e.g., the call to the `owns` function in Listing 8(a), which additionally specifies that whenever a chat room is deleted, the deletion operation should be propagated to all the messages referenced by that room via the `messages` field).

Having this syntactic sugar (instead of just using the built-in `class` keyword) provides a convenient way of specifying typed fields, but more importantly, being in charge of class generation also gives us an easy way to hook into all field accesses, where we perform the necessary policy checks.

Note that, however, none of our language features mandated this implementation choice; a different implementation targeting a different platform is possible.

5.4.3 Runtime Environment

One of our main goals is to relieve the programmer of having to explicitly implement a distributed system, i.e., explicitly synchronize multiple processes, handle inter-process communication, manage queues and messages, ensure data consistency, and a number of other tasks typical for distributed and concurrent programming. By introducing a specific programming model (as described previously), we tailored a generic runtime environment to automate all those tasks. The runtime implements a standard message-passing architecture, a well-known and widely used idiom for designing distributed systems, which we use to dispatch events and data updates between entities (Figure 5-1).

Another important role of the runtime environment is to automatically check and enforce privacy policies at runtime. Policies are checked at every field access attempted by the user-defined code: all relevant restriction rules are discovered and applied. Instead of throwing an exception when the access is forbidden, an empty relation is returned. This makes it possible for the client code to be written in a mostly policy-agnostic style. For example, the client code can simply say `room.members` to fetch the members of a chat room, and rely on the runtime to return only those elements that the client is allowed to see.

Policies are also considered when objects are being serialized (by the runtime) prior to being sent to another machine (e.g., as part of the automatic propagation of data updates). Consider a client (`client`) attempting to fetch all rooms by executing `server.rooms`. This is a legal operation, as all field accesses are permitted (it is a public IRC application). Any sensitive data (e.g., the password and status fields of the room members), however, must still be hidden or their values properly filtered out, which our runtime does automatically (simply by returning an empty relation every time access is forbidden).

This illustrates the declarative nature of our privacy policies, and how the runtime can automatically enforce them. It also shows that the operational code (e.g., event handlers, embedded GUI formulas, etc.) usually can be written independently of privacy policies, and does not need to be updated when policies change.

5.4.4 Online Code Generator

Many of the responsibilities of the runtime environment are enabled by the code automatically generated from the core models, on the fly, during the system initialization phase. In addition, we use code generation to automate various common tasks. These tasks are briefly described next.

Database Migrations

The richness of our data model makes it possible to handle data persistence fully automatically. This includes (1) generating and maintaining a database schema, and (2) implementing an object-relational mapper (i.e., mapping domain objects onto that schema).

A database schema provides a way to persist all relevant information from the domain model. Because the schema is always supposed to closely mirror the model, ideally it should not have to be written/programmed separately. In standard Rails, however, that is not the case; the schema exists as a standalone code artifact, and the programmer is in charge of maintaining it and keeping it in sync with the application model. Although Rails comes with automated generators that can create schema skeleton files from simple $name \rightarrow type$ pairs, they only work for primitive types and scalar fields; for more advanced features like type inheritance and non-scalar fields (many-to-many relations), the programmer has to manually extend the generated schema file in such a way that it works with the object-relational mapper on the other side.

Figure 5-2(a) gives a full listing of the database schema (in the form of a Ruby migration class, standard for the Rails framework) that RED automatically generates for the IRC example. This schema supports all the features of the model, so the programmer does not even have to look at it.

ActiveRecord (the object-relational mapper used in Rails and in our framework) implements the *single table inheritance* strategy for handling inheritance. Hence, for each base type we generate one table with columns for all fields of all of its subtypes, plus an extra string-valued column (named `:type`) where the actual record type is stored. For example, in the `:auth_users` table, the first three columns correspond to fields from `AuthUser` and the fourth column is for the single field from `User`. Furthermore, in every other table referencing such a table, an additional "column type" column must be added to denote the declared type of the corresponding field, as in the `:msg` table (columns `:sender` and `:sender_type`).

When a record field type is of arity greater than 1, a separate join table must be created to hold that relation. This is the case with the `ChatRoom.members` field (referencing a set of `Users`). The corresponding join table (`:chat_rooms_users_members`) stores all tuples of the `:members` relation by having each row point to a row in the `:chat_rooms` table and a row in the `:users` table. In the special case when a field owns a set of records (e.g., field `ChatRoom.messages`, meaning that a given message can be referenced via that field by at most one chat room), instead of a join table, a referencing column is placed in the table corresponding to the type of that field (the `:chat_room_as_message` column in table `:msgs`).

The last `create_table` statement in Figure 5-2(a) simply creates a table where the session data will be stored, and is independent of the domain data model.

Despite being mostly straightforward, writing migrations by hand is still tedious and

(a) auto-generated Rails migration file

```
class UpdateTables < ActiveRecord::Migration
 def change
  create_table :auth_clients do |t|
    t.column :auth_token, :string
    t.references :user
    t.column :user_type, :string
    t.references :user
    t.column :user_type, :string
    t.column :type, :string
    t.timestamps
  end
  create_table :auth_servers do |t|
    t.column :type, :string
    t.timestamps
  end
  create_table :auth_users do |t|
    t.column :name, :string
    t.column :email, :string
    t.column :password_hash, :string
    t.column :status, :string
    t.column :type, :string
    t.timestamps
  end
  create_table :msgs do |t|
    t.column :text, :text
    t.references :sender
    t.column :sender_type, :string
    t.references :chat_room_as_message
    t.timestamps
  end
  create_table :chat_rooms do |t|
    t.column :name, :string
    t.references :server_as_room
    t.column :server_as_room_type, :string
    t.timestamps
  end
  create_table :chat_rooms_users_members,
               :id => false do |t|
    t.column :chat_room_id, :int
    t.column :user_id, :int
  end
  create_table :sessions do |t|
    t.string :session_id, :null => false
    t.text :data
    t.timestamps
  end
  add_index :sessions, :session_id
  add_index :sessions, :updated_at
 end
end
```

(b) auto-generated ActiveRecord classes

```
class Msg < Red::Model::Record # < ActiveRecord::Base
 attr_accessible :text
 belongs_to      :sender,
                 :class_name  => "User",
                 :foreign_key => :sender_id
 belongs_to      :chat_room_as_message,
                 :class_name  => "ChatRoom",
                 :foreign_key => :chat_room_as_message_id,
                 :inverse_of  => :messages
 # interceptors for field getters and setters
 ...
end

class ChatRoom < Red::Model::Record # < ActiveRecord::Base
 attr_accessible         :name
 has_and_belongs_to_many :members,
               :class_name              => "User",
               :foreign_key             => :chat_room_id,
               :association_foreign_key => :user_id,
               :join_table  => "chat_rooms_users_members"
 has_many      :messages,
               :class_name  => "Msg",
               :foreign_key => :chat_room_as_message_id,
               :dependent   => :destroy
 belongs_to    :server_as_room,
               :class_name  => "Server",
               :foreign_key => :server_as_room_id,
               :inverse_of  => :rooms
 # interceptors for field getters and setters
 ...
end

class User < RedLib::Web::Auth::AuthUser
  attr_accessible         :status
  has_and_belongs_to_many :chat_rooms_as_member,
                :class_name              => "ChatRoom",
                :foreign_key             => :user_id,
                :association_foreign_key => :chat_room_id,
                :join_table => "chat_rooms_users_members"
  has_many      :msgs_as_sender,
                :class_name  => "Msg",
                :foreign_key => :sender_id,
                :inverse_of  => :sender
  has_many      :clients_as_user,
                :class_name  => "Client",
                :foreign_key => :user_id,
                :inverse_of  => :user
 # interceptors for field getters and setters
 ...
end
```

Figure 5-2: Snippets of automatically generated code for the IRC example

time consuming, and, for developers new to Rails, can often be a source of mysterious runtime errors. Even after those initial errors have been fixed, the gap between the schema and the application model still remains. RED eliminates all these issues by having a single unified model of the system and automatically driving various implementation-level technologies (such as the database schema maintenance) directly from it.

ActiveRecord Classes and Reflections

As explained in Section 5.4.2, the `record` keyword in RED is actually implemented as a Ruby function that creates a new class and assigns a named constant to it. This section explains generated record classes (listed in Figure 5-2(b)) in more detail.

`ActiveRecord` provides "*reflections*" for specifying associations between *model classes* (i.e., records in our terminology). Primitive fields are declared with `attr_accessible` (e.g., `ChatRoom.name`), one-to-many associations with `has_many` on one side and `belongs_to` on the other (e.g., `ChatRoom.messages`), and `has_and_belongs_to_many` is used for many-to-many associations with (e.g., `ChatRoom.members`). Various options can be provided to specify the exact mapping onto the underlying database schema.

As with migration generators, Rails provides generators for ActiveRecord model classes as well, but again, with limited features and capabilities. While most of the schema-mapping options (e.g., `:foreign_key`, `:join_table`, `:association_foreign_key`) can be omitted if the naming convention is followed when the schema is written, the programmer still has to manually write these reflections for all but primitive fields. Furthermore, ActiveRecord requires that reflections are written on both sides of an association, meaning that each non-primitive field has to have its inverse explicitly declared in the opposite class (which is another step that our system eliminates). Finally, even though the database schema and the model classes are coupled, there is nothing that keeps them in sync in standard Rails. This not only makes the development process more cumbersome and error-prone, but also makes it difficult to perform any system redesign or refactoring.

Controlling the generation of model classes also lets us intercept all field accesses, where we perform all the necessary security checks, detect changes to the data model for the purpose of updating client views, wrap the results of getter methods to enable special syntax (e.g., the Alloy-style relational join chains), etc.

JavaScript Model for the Client-Side

One of the main ideas behind SUNNY is to have a single unified model of the system, and a model-based programming paradigm that extends beyond language and system boundaries. In RED, we wanted to preserve this idea and enable the same kind of model-based programming style on both the server side and the client side, despite the language mismatch. More concretely, we wanted to provide the same high-level programming constructs for instantiating and asynchronously firing events in JavaScript on the client side, as well as constructing and manipulating records.

To that end, we translate the system domain model into JavaScript, to make all the model meta-data available on the client side. We also implemented a separate JavaScript library that provides prototypes for the generated model classes, as well as many utility operations.

Listing 10 gives an excerpt from the translation of the IRC application's domain model. Up on the top are constructor functions for all records, machines, and events. The `mk_record` and `mk_event` functions (part of our library) take a name and (optionally) a super constructor, and return a constructor function with the given name and a prototype extending the super constructor's prototype. This is followed by the meta-data for each record, machine,

and event, which contains various information about the type hierarchy, fields, field types, etc. All this information is necessary for our library to be able to provide generic and application-agnostic operations. One such operation we mentioned before, in Section 5.4.1, where we talked about how DOM elements having the `data-trigger-event` attribute are automatically turned into event triggers.

Let us finally take a look at how asynchronous message sending is implemented on the client side, that is, how such an operation can be specified declaratively, directly in an HTML template file, without writing any Ajax code.

The *chat_room.html.erb* template (Listing 9(c)) contains a text input field and a send button, with the intention to trigger the `SendMsg` event and send whatever message is in the text input field whenever the send button is pressed. To achieve that, we added three HTML5 data attributes to the send button element; we used `data-trigger-event`, as before, to denote the type of the event, and two `data-param` attributes to specify the two mandatory arguments of the `SendMsg` event, `room` and `msgText`.

The value for the `room` parameter is known statically—it is exactly the chat room object for which the *chat_room* template is being executed. However, that value is an object, so it is not possible to directly embed it in the template as a string-valued attribute. Instead, we inline a small piece of JavaScript code that, when executed, creates an equivalent room on the client side. Knowing the id of that room, and having a full replica of the model classes on the client, that code is as simple as `new ChatRoom(<%=room.id%>)`[2]; we only need to tell our JavaScript library that the value we are passing is not a string, but code, by enclosing it in `${}`[3].

The value for the `msgText` parameter is not known statically, and has to be retrieved dynamically when the user presses the send button. As in the previous case, we can specify that by inlining a piece of JavaScript that finds the input text field by its id (using the jQuery syntax `$('#<id>')`) and reads its current value (by calling the `.val()` function).

An alternative approach to declaratively specifying event parameter bindings, that would require no JavaScript from the programmer, would be to somehow annotate the input text field (e.g., again by using the HTML5 data attributes) as the designated value holder for the `msgText` event parameter. A drawback of such an approach is that, in general, it leads to code fragmentation, where a single conceptual task can be specified in various different (and not predetermined) places, potentially significantly reducing code readability. For that reason, we thought it was better to have all the code and specification in one place, even if the user has to write some JavaScript.

[2]Our JavaScript library actually does not complain if a numeric id is passed where a record object is expected—having all the meta-model information available, it can easily find the event parameter by the name, look up its type, and reflectively construct an instance of that type. Instead of using this shortcut (which works only for record objects) in the main text, we used a more verbose version to illustrate a more general approach and all of its power and flexibility.

[3]Note that this dollar sign has nothing to do with the jQuery dollar sign; it is rather our own syntax for recognizing attribute values that should be computed by evaluating the JavaScript code inside `${}`.

```
/* ------------- record signatures ------------- */
var AuthUser   = Red.mk_record("AuthUser");
var User       = Red.mk_record("User", AuthUser);
var Msg        = Red.mk_record("Msg");
var ChatRoom   = Red.mk_record("ChatRoom");
var AuthClient = Red.mk_record("AuthClient");
var AuthServer = Red.mk_record("AuthServer");
var Client     = Red.mk_record("Client", AuthClient);
var Server     = Red.mk_record("Server", AuthServer);

/* ------------- event signatures ------------- */
var Register   = Red.mk_event("Register");
var SignIn     = Red.mk_event("SignIn");
var SignOut    = Red.mk_event("SignOut");
var Unregister = Red.mk_event("Unregister");
var CreateRoom = Red.mk_event("CreateRoom");
var JoinRoom   = Red.mk_event("JoinRoom");
var SendMsg    = Red.mk_event("SendMsg");

/* ------------- record meta ------------- */
ChatRoom.meta = new Red.Model.RecordMeta({
 "name"      : "ChatRoom",
 "short_name": "ChatRoom",
 "sigCls"    : ChatRoom,
 "abstract"  : false,
 "parentSig" : Red.Model.Record,
 "subsigs"   : [],
 "fields"    : [
   new Red.Model.Field({parent: ChatRoom, name: "name",     type: "String", multiplicity: "one" }),
   new Red.Model.Field({parent: ChatRoom, name: "members",  type: User,     multiplicity: "set" }),
   new Red.Model.Field({parent: ChatRoom, name: "messages", type: Msg,      multiplicity: "seq",
                        owned: true })
 ]
});
...

/* ------------- event meta ------------- */
SendMsg.meta = new Red.Model.EventMeta({
 "name"      : "SendMsg",
 "short_name": "SendMsg",
 "sigCls"    : SendMsg,
 "abstract"  : false,
 "parentSig" : Red.Model.Event,
 "subsigs"   : [],
 "params"    : [
   new Red.Model.Field({parent: SendMsg, name: "room",    type: ChatRoom, multiplicity: "one" }),
   new Red.Model.Field({parent: SendMsg, name: "msgText", type: "String", multiplicity: "one" })
 ]
});
...
```

Listing 10: Excerpt from the JavaScript translation of the domain model, which the client-side code can program against

5.4.5 Dynamic Template-Based Rendering Engine

To go along with this declarative approach for programming the core business logic of an event-based system, RED implements a mechanism for declaratively building graphical user interfaces. The main responsibility of this mechanism is to automatically and efficiently update and re-render the GUI (or relevant parts of it) when a change is detected in the data model. This idea is similar to the concept of "data bindings" (e.g., [128, 134]), but is more general and more flexible.

Traditionally, GUIs are built by first constructing a basic visual layout, and then registering callbacks to listen for events and dynamically update bits and pieces of the GUI when those events occur. In contrast, we want the basic visual layout (like the one in Listing 9) to be sufficient for a dynamic and fully responsive GUI. In other words, we want to let the designer implement (design) a single static visualization of the data model, and from that point on rely on the underlying mechanisms to appropriately and efficiently re-render that same visualization every time the underlying data changes.

To implement this approach, we expand on the well-known technique of writing GUI widgets as textual templates with embedded formulas (used to display actual values from the data model) and using a *template engine* [9] to evaluate the formulas and paste the results in the final output. To specify input templates, we use the ERB language (the default template language in Rails) without any modifications. Unlike the existing renderer for ERB, however, our system detects and keeps track of all field accesses that happen during the evaluation of embedded formulas. Consequently, the result of the rendering procedure is not a static text, but a *view tree* where embedded formulas are hierarchically structured and associated with corresponding field accesses (as illustrated in Section 5.4.1). That view tree is what enables the permanent data bindings—whenever the underlying data changes, the system can search the tree, find the affected nodes, and automatically re-render them.

In the context of web applications, only a textual response can be sent back to the client. Therefore, when an HTTP request is received, the associated template is rendered, and a view tree is produced. The view tree is saved only on the server side. The client receives the same plain-text result that the standard ERB renderer would produce along with some meta-data to denote node delimiters; the browser renders the plain-text response, and our client-side JavaScript library saves the meta-data. When a data change is detected on the server-side, the server finds and re-renders the affected nodes and pushes plain-text *node updates* to corresponding clients; each client then, already having the meta-data, knows where to cut and paste the received update to automatically refresh the GUI.

5.5 Semantics

This section formalizes SUNNY's notion of policies, as well as the computational model behind it. The syntax introduced in Section 2.4 is used here again.

Figure 5-3(a) lists the datatypes defined by SUNNY. `Sig` denotes a user-defined record type; the three predefined sigs are `SunnyUser`, `SunnyClient`, and `SunnyServer`. A `Value` is either a `Record` or an `Object`. A `Record` instance has fields (a set of name-value pairs) and a corresponding type (`Sig`). A `Policy` is assigned a prototype `Operation` to which it applies and a checker function; the checker function, for a given client and a concrete operation, returns an `Outcome` to designate whether the operation should be `Allowed` or `Denied`. The special case is the `Read` operation, for which the policy checker can also return `Restricted` to provide a different value to serve as the result (these policies will be referred to as *filtering policies*). This feature is particularly convenient for defining client-specific views of the data model, for example, to filter out sensitive information for unauthorized clients. Finally, SUNNY defines the standard CRUD operations (`Create`, `Read`, `Update`, `Delete`), and two additional `Push` and `Pull` operations for inserting and removing a tuple

(a) SUNNY syntactic domains

```
type Sig        = SunnyUser | SunnyClient | SunnyServer | <user-defined types>
type Value      = Record(fields: String × Value, type: Sig)
                | Object
type Operation  = Create(type: Sig)
                | Read(r: Record, fld: String, val: Value)
                | Update(r: Record, fld: String, val: Value)
                | Push(r: Record, fld: String, val: Value)
                | Pull(r: Record, fld: String, val: Value)
                | Delete(r: Record)
type Policy     = {op: Operation,
                   checker: SunnyClient → Operation → Outcome}
type Outcome    = Allowed | Restricted(val: Value) | Denied(reason: String)
type FldComp    = {client: SunnyClient, r: Record, fld: String}
type Dep        = {comps: FldComp list}
type SunnyMeta  = {sigs: Sig list,
                   policies: Policy list,
                   fieldDeps: Record × String × Dep,
                   queryDeps: Sig × Dep,
                   server: SunnyServer,
                   online: SunnyClient list,
                   client: SunnyClient option,
                   stack: Operation list,
                   comp: FldComp option}

let Sunny       = SunnyMeta(...) // global Sunny variable
```

(b) built-in functions

```
fold   : (A → E → A) → A → E list → A                 functional fold
filter : (E → Bool) → E list → E list                 functional filter
keys   : A × B → A                                    extract keys from a dictionary
```

(c) framework-provided datatypes and functions

```
type SigListener = {created: Record → unit,
                    updated: Record → String list → unit,
                    deleted: String → unit}

observeChanges : Sig → SigListener → unit                     observe changes to sig records
push           : SunnyClient → Sig → String → Object → unit   push msg to client
db_create      : Record → unit                                create record in the database
db_read        : Record → String → Value                      read field value from the database
db_update      : Record → String → Value → unit               write field value to the database
db_delete      : Record → unit                                delete record from the database
db_find        : Record → String → Record list                query the database
```

Figure 5-3: Datatypes, global variables, built-in and framework-provided functions

from a relation, respectively.

Datatypes `FldComp` and `Dep` are used for keeping track of field dependencies on the server side. The former designates a "field computation" for a given client, while the latter keeps a list of computations that depend on it. The purpose of this is to automatically re-run a field computation whenever any of the fields it depends on changes, to ensure that all clients always have the current view of the system.

Finally, the `Sunny` global variable keeps both (1) all the application metadata: sigs, policies, and dependency objects, and (2) some dynamic information about the current state of the system execution: the server record, a list `SunnyClient` records representing currently connected clients, the *principal client* (on whose behalf the code is currently running), the current *field computation*, and a stack of operations currently being checked. The metadata is populated once, upon initialization, after which it remains immutable; the dynamic variables are mutable and used at runtime to communicate between different SUNNY components.

Figures 5-3(b) and (c) summarize the functions assumed to be provided either by the programming language or the underlying web framework.

5.5.1 Policy Checking

Policies are attached to operation prototypes. Multiple policies can be defined for the same operation. Before each operation is executed, applicable policies are dynamically discovered and checked first. If no policy is applicable, the operation is by default allowed (a concrete implementation can easily make this default configurable). If any policy returns `Denied` the operation is denied; when `Restricted` is returned for a `Read` operation, the result is set to the restricted value.

Policy checking is formalized in the `checkPolicies` function in Figure 5-4. If there is no current client, the operation is executing on behalf of the server, thus, it is allowed. Otherwise, applicable policies are discovered (using the `opsMatch` auxiliary function) and their *checker* functions are executed against the principal client (`Sunny.`*client*). The checker results (outcomes) are finally combined using the `AND` function, which out of two given outcomes selects the more restrictive one.

Since policies can depend on sensitive values (those guarded by policies), there may exist circular dependencies among policies. To avoid infinite loops, the `checkPolicies` function maintains a stack of concrete operations for which the policies are being checked. If a current operation is found on the stack, `Allowed` is immediately returned.

The stack of operations currently under policy checking is kept in the `Sunny.`*stack* global variable. Our formalization of how the `checkPolicies` function maintains that variable makes use of a special language construct, **`with`** `var ↤ val` **`do`** `<block>`, which is a shortcut for the following: (1) set the value of the `var` variable (which is typically global) to `val`, (2) execute the given block, and finally (3) restore the value of `var` to its original value. We introduce this shortcut construct, as it will be used several more times later in this section.

Figure 5-5 formally shows how the policy checker is invoked before each CRUD operation is performed. The most interesting is the `read` function: when access is denied, instead of raising an exception, an empty value is returned. This supports the idea of policy-agnostic programming; for example, it allows the programmer to have generic UIs

```
opsMatch:  Bool → Operation → Operation → Bool
let opsMatch = λ(prototypeOnly, o1, o2)·
  let recEq = λ(r1, r2)· if prototypeOnly then r1.type == r2.type else eq(r1, r2)
  match o1, o2
  | Create, Create  →  o1.type == o2.type
  | Delete, Delete  →  o1.r.type == o2.r.type
  | Read, Read      →  recEq(o1.r, o2.r) ∧ o1.fld == o2.fld
  | Update, Update  →  recEq(o1.r, o2.r) ∧ o1.fld == o2.fld
  | Push, Push      →  recEq(o1.r, o2.r) ∧ o1.fld == o2.fld
  | Pull, Pull      →  recEq(o1.r, o2.r) ∧ o1.fld == o2.fld
  | _, _            →  FALSE
```

```
AND: Outcome → Outcome → Outcome
let AND = λ(out1, out2)·
  match out1, out2
  | Denied, _       →  out1
  | _, Denied       →  out2
  | _, Restricted   →  out2
  | _, _            →  out1
```

```
applicablePolicies:  Operation → Policy list
let applicablePolicies = λ(op)·
  filter λ(p)·opsMatch(TRUE, p.op, op), Sunny.policies
```

```
checkPolicies:  Operation → Outcome
let checkPolicies = λ(op)·
  if filter opsMatch(FALSE, op), Sunny.stack == []
    Allowed()  // recursive call → allow to avoid infinite loops
  else
    with Sunny.stack ↤ op :: Sunny.stack do
      match Sunny.client
      | None        →  Allowed()  // no principal client → privileged mode → allow
      | Some(clnt)  →  let checkAndCombine = λ(acc, policy)·
                         let o = match op, acc
                         | Read, Restricted  →  Read(op.r, op.fld, acc.val)
                         | _, _              →  op
                         AND(acc, policy.checker(clnt, o))
                       fold checkAndCombine, Allowed(), applicablePolicies(op)
```

Figure 5-4: Formalization of policy checking in SUNNY

designed to display the entire underlying data model, and safely rely on the runtime to remove any sensitive information. The read function also invokes depend to establish a dependency between any enclosing computation and this read operation (explained in more detail shortly). Two other interesting cases are the *push* and *pull* functions: if no applicable push/pull policies are found, update policies are checked instead, but if both kind of policies are found, the more specific push/pull take precedence.

```
create:  Sig → Record
let create = λ(s)·
  match checkPolicies(Create(s))
  | Denied   → raise(AccessDenied)
  | _        → db_create(Record({}, s))
```

```
read:  Record → String → Value
let read = λ(r, fld)·
  depend(SunnyMeta.fieldDeps[r][fld])
  let val = db_read(r, fld)
  match checkPolicies(Read(r, fld, val))
  | Denied          → []
  | Allowed         → val
  | Restricted(v)   → v
```

```
update:  Record → String → Value → unit
let update = λ(r, fld, val)·
  match checkPolicies(Update(r, fld, val))
  | Denied   → raise(AccessDenied)
  | _        → db_update(r, fld, val)
```

```
push:  Record → String → Value → unit
let push = λ(r, fld, val)·
  let pushOp = Push(r, fld, val)
  let op = applicablePolicies(pushOp) == [] ? pushOp : Update(r, fld, val)
  match checkPolicies(op)
  | Denied   → raise(AccessDenied)
  | _        → db_push(r, fld, val)
```

```
pull:  Record → String → Value → unit
let pull = λ(r, fld, val)·
  let pullOp = Pull(r, fld, val)
  let op = applicablePolicies(pullOp) == [] ? pullOp : Update(r, fld, val)
  match checkPolicies(op)
  | Denied   → raise(AccessDenied)
  | _        → db_pull(r, fld, val)
```

```
delete:  Record → unit
let delete = λ(r)·
  match checkPolicies(Delete(r.type))
  | Denied   → raise(AccessDenied)
  | _        → db_delete(r)
```

```
find:  Sig → String → Record list
let find = λ(sig, query)·
  depend(SunnyMeta.queryDeps[sig])
  db_find(sig, query)
```

Figure 5-5: Formalization of CRUD operations in SUNNY

5.5.2 Reactivity

In this thesis, by "reactive system" we mean distributed system that automatically keeps the shared data up to date on all connected peers. In a web framework, this is typically done by explicitly pushing data updates from the server to the connected clients. Existing approaches like Touch Develop [158, 159] and Meteor [4] implement this for "public" data sources only, by replicating their content on all clients. In the presence of declarative privacy policies (as proposed in this thesis), this task becomes more difficult. The policies can deny access to certain data elements (i.e., fields, in SUNNY), or even modify their content. The logic behind such a policy is often defined in terms of other data elements; consequently, executing a policy to determine a field value can dynamically induce dependencies to other fields. A reactive system that promises to always keep the data consistent and updated on all clients must, therefore, (1) remember those dependencies, (2) automatically recalculate field values upon field changes, and (3) automatically send the updated field values to relevant clients.

Figure 5-6 formalizes this process. Upon system initialization, for each record type (`Sunny.`*sigs*) callbacks are registered to observe changes to its records. We assume the existence of the *`observeChanges`* function, which simply notifies all registered listeners whenever a record is created, updated, or deleted. Before forwarding a received change to the online clients, computations dependent on that change are invalidated first. This is done by invoking the `changed` function, passing an instance of the `Dep` datatype; all three listeners trigger the "query" dependency for the associated record type, while the `updated` listener additionally triggers the "field" dependencies for updated fields. Next, separately for each client, field values are filtered out according to applicable policies, after which the update is finally pushed to the client (by calling the framework-provided *`push`* function). The `each_clnt` function iterates through the online clients, and the `restrict` function performs the field filtering.

The `each_clnt` function is simple in its nature: it iterates through the list of online clients (`Sunny.`*online*) and each time passes the client to a callback function. The main reason for using a global variable to designate the principal client (`Sunny.`*client*) is that any practical implementation would likely have to do the same. That is simply because, for example, a client record is required for policy checking, which is done at every field access inside of the corresponding field getter method, and getter methods typically cannot accept an additional client argument.

The `restrict` function builds a map of field name-value pairs by folding over all fields names of a given record and for each performing the `read` operation. The `read` operation will consult all applicable policies, so the return values will be appropriate for the current principal client. More importantly, just before invoking `read`, the `restrict` function sets the current field computation (`Sunny.`*comp*). This is done so that all field accesses and other dependencies induced during the policy checking phase are remembered for this field computation. For example, as explained earlier in this section, every `read` operation calls `depend` on the `Dep` object associated with its target field. As formalized now in Figure 5-6, the `depend` function simply remembers the current computation, if one is set. On the other hand, whenever a dependency is triggered (e.g., inside *`observeChanges`*), the `changed` function is called, which, as now formally defined, goes through the list of its remembered

computations, reevaluates the fields of the associated record (by calling `restrict` on it) and pushing the change to the associated client.

The design of our server-side reactive field computations is very much inspired by that of Meteor [4]; Meteor, however, supports only client-side reactive computations.

```
depend:  Dep → unit
let depend = λ(dep)·
  match Sunny.comp
  | Some(comp) → dep.comps ↤ comp::dep.comps

changed:  Dep → unit
let changed = λ(dep)·
  let comps = dep.comps
  dep.comps ↤ []
  for c in comps do
    with Sunny.client ↤ Some(c.client) do
      push(c.client, c.r.type, "changed", restrict(c.r))

restrict:  Record → String × Value
let restrict = λ(r)·
  let accumFldNameVals = λ(accDict, fldName)·
    let fComp = Some(FldComp(Sunny.client, r, fldName))
    with Sunny.comp ↤ fComp do
      accDict + fldName × read(r, fldName)
  fold accumFldNameVals, {}, keys(r.fields)

each_clnt:  (SunnyClient → unit) → unit
let each_clnt = λ(cb)·
  for clnt in Sunny.online do
    with Sunny.client ↤ Some(clnt) do cb(clnt)
```

```
publishing changes (executed once at initialization)
for s in Sunny.sigs do
  observeChanges s, SigListener(
    created = λ(r)·
      changed(Sunny.queryDeps[s])
      each_clnt λ(c)·push(c, s, "added", restrict(r))
    updated = λ(r, fldNames)·
      changed(Sunny.queryDeps[s])
      for f in fldNames do changed(Sunny.fieldDeps[r][f])
      each_clnt λ(c)·push(c, s, "changed", restrict(r))
    deleted = λ(id)·
      changed(Sunny.queryDeps[s])
      each_clnt λ(c)·push(c, s, "removed", id)
  )
```

Figure 5-6: Formalization of auto publishing in SUNNY

5.5.3 Concurrency Model

We assume the concurrency model of fibers. Fibers are similar to threads in that the execution can switch from one fiber to another before the first fiber has finished. The big difference with fibers is that the switch is always initiated explicitly, at predefined points. In other words, a fiber cannot be interrupted unless it explicitly yields the execution (e.g., by invoking a special function) to another fiber.

Our formalization does not contain any explicit yield points. Yields can still happen within our functions, however. For example, every call to a database function (e.g., `db_find`) potentially yields to another fiber. Since multiple fibers can be queued up at a same time (e.g., each HTTP request executes in a separate fiber, sig listeners (registered with `observeChanges`) are concurrently invoked by the database in separate fibers, etc.), the fact that global shared variables are used (e.g., `Sunny.`*client*, `Sunny.`*comp*, etc.) means that data races may happen. In practice, we mitigated this by keeping the shared variables in a fiber-local storage.

5.6 Automated Reasoning and Analysis

Although SUNNY simplifies the development of interactive web applications, and by construction eliminates a whole class of concurrency bugs, it does not eliminate all possible bugs. The user implementation of events can still fail to satisfy the functional requirements of the application. Applying the standard software quality assurance techniques to SUNNY programs is, therefore, still of high importance. We designed SUNNY with this in mind, and in this section we discuss how our programming paradigm is amenable to techniques like automated testing, model checking, and software verification. This thesis, however, we only discusses how such techniques could be carried out in principle.

5.6.1 Testing

Testing is the most widely used method for checking program correctness. Testing an event-driven system is both challenging and time consuming, because one needs to generate *realizable* traces (sequences of events). The challenging part in discovering realizable traces is that the preconditions need to hold for each event in the sequence, and the time-consuming part is that the traces can be long, and therefore, there can be too many of them to explore manually. Having both preconditions and postconditions of each event formally specified in our event model allows us to use symbolic-execution techniques [89], and build on recent successes in this domain [166], to discover possible traces automatically.

A symbolic execution engine would start with an empty path condition; at each step, the engine would consider all events from the model and discover the ones that are realizable from the current state (this can be done by using an automated SMT solver [24,48] to check if there exists a model in which both the current path condition and the event's precondition are satisfied). When an event is found to be realizable, a new state is created and the event's postcondition is appended to the path condition for the new state. Since at each step of this process multiple events may be found to be realizable, the algorithm proceeds by exploring

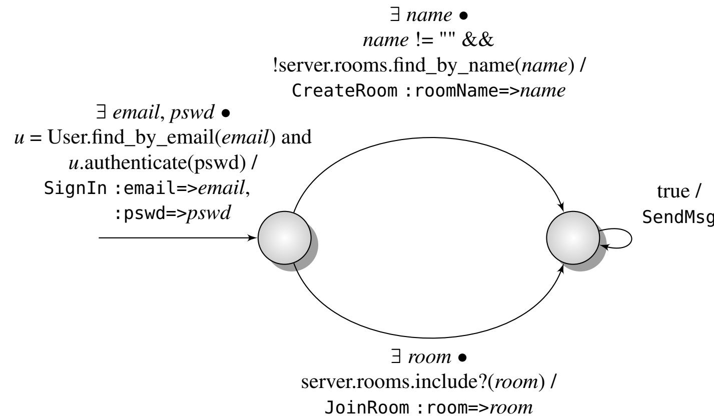


Figure 5-7: State diagram for the IRC example

the entire graph, effectively yielding a *state diagram* by the end. Figure 5-7 depicts the state diagram extracted from the running example (Listing 8). Each node in the diagram describes a symbolic state and each edge describes a transition that can happen when the condition on the edge is satisfied and the event is executed. For example, moving from the initial state to the next state requires that a user initiates a `SignIn` event and provides a correct name and password. This transition results in the execution of the postcondition of the `SignIn` event.

In addition to automated testing of traces, a state diagram can be used to automatically create a test environment—the state necessary before the execution of a test—for all unit tests. Considering Figure 1, if a developer wants to test the `SendMsg` event, there should be a registered user in a room. To create such a state, a sequence of other events have to be executed before `SendMsg`. Inferring from Figure 5-7, `SignIn` and `CreateRoom` event handlers must be executed. Executing these events requires solving the precondition of each event on the path.

Functional unit testing of events also becomes easier. A black-box unit test for the `SendMsg` event would have to check that the message sent indeed gets added to the list of messages of the given room, that it gets added to the very end of the list, that no other messages get dropped from that list, etc. In SUNNY, this can be done directly, without having to set up any mock objects, e.g., to abstract the network and the actual peer points, as no network is required.

In a traditional event-driven system, an implementation of a single functional unit is often fragmented over several classes. Consider the `SignIn` event: the user sends his or her credentials to the server, the server checks the authenticity, sends back the result, and based on that result, both the server and the client update their local state. In the traditional model, the client-side code can initiate the event (by sending a message to the server), and schedule a continuation that will run when a response is received. The continuation, which is typically a separate procedure, then implements the logic for updating the local state based on the server response. Such fragmented code is very hard to test as a unit, so it is often turned into an integration test, and integration tests are typically more laborious to write and require more elaborate setup. In SUNNY, because of the shared view of the global

data, there is no need for such fragmentation; the event handler can be a single procedure that updates only the global data model, meaning that it can easily be tested as a unit.

5.6.2 Model Checking

Loosely coupled events without explicit synchronization and communication allow model checking to scale. The source of non-determinism in SUNNY models is the order in which events are executed. Because of the non-determinism in scheduling, a model may exhibit different behavior for the same input (i.e., the same values of event parameters) with a different order of event execution. The goal of software model checking is conceptually to explore all orders to ensure the correct execution. Note that the exploration need consider only the semantics of the model and not the semantics of the underlying runtime system. Based on our prior experience with model checking actor programs [96, 156], X10 programs [65], and database applications [64], we believe that an efficient model checking approach can be developed for our new paradigm.

For example, a model checker can be used to check end-to-end properties for all scenarios that the system can possibly exhibit. One such property could be "it is impossible that at one point two different chat rooms have two different users with the same name". The tool can automatically either confirm that the property in question always holds, or find a scenario (i.e., a sequence of events leading to a state) in which the property is violated.

5.6.3 Verification and Program Synthesis

The technique of discovering realizable sequences of events can also be used to synthesize higher-level operations. For example, a novice IRC user may wonder what are the steps that need to be taken in order to post a message in a chat room. Given such an end-goal, a tool can discover that one possible scenario to achieve that goal is to first `SignIn`, then `JoinRoom`, and finally `SendMsg`. An alternative solution would be to `CreateRoom` instead of `JoinRoom` at the second step. These scenarios can be displayed to the designer and serve as a guide to better understanding possible functionalities of the system (which can be especially useful for bigger systems with many possible events).

5.7 Discussion

It can be argued that designing a system around a given property is the best way to ensure that the system correctly implements that property [79]. The SUNNY approach is certainly in that spirit since it encourages the programmer to carefully design and specify the core part of an event-driven system, i.e., the event model. Furthermore, the programmer does so mostly declaratively, by specifying key properties of events in isolation, without being bogged down by the operational details of the entire distributed system.

We believe that, in most cases, even the event effects (postconditions) might be specified fully declaratively, and yet efficiently executed. We showed previously that declarative specifications can be executable (within a traditional object-oriented language) with certain performance handicaps [119]. Moreover, Near and Jackson [129] showed that, in a setting

of a typical web application, most server-side actions (or "events" in our terminology) boil down to (possibly conditional) assignments to variables, which is still declarative, but much easier to execute efficiently. They also showed how this fact can be exploited to build a scalable verifier, which is of comparable complexity to executing a declarative postcondition in the first place.

Our system also lends itself to model-based user interface software tools, which, by definition, take a high-level declarative model of an interactive system and help the programmer build a user interface (UI) for it (either through an integrated development environment or automated code generation) [146]. For example, a UI can be automatically generated from a SUNNY model that contains generic widgets for querying existing records, creating new instances of records defined in the model, creating associations between existing records via the fields defined in the model, triggering events, and so on, all while respecting the model-defined invariants, event preconditions, and privacy policies. Some existing implementations of scaffolding can already generate a graphical UI that supports standard CRUD operations (create, read, update, delete) for all data model classes; in contrast, with SUNNY models *scaffolding of events* is supported, thus enabling a fully generic user interface that actually covers the *full* functionality of the system.

5.8 Evaluation

5.8.1 Gallery of *Sunny.js* Applications

We used *Sunny.js*, our pure-JavaScript implementation of SUNNY, to develop a number of prototypical reactive web applications. In this section, our intention is to demonstrate the applicability of the SUNNY approach and the expressive power of its policies. *Sunny.js* is written in CoffeeScript, so the syntax of our code listings is slightly different from the one used in Section 5.2, but it closely follows all the presented language concepts and ideas. When we informally describe policies, we say how they apply to a certain *user*, where by "*user*" (or "acting user") we formally mean the corresponding `SunnyUser` record associated (via the `user` field) with the implicit client (`this.client`, as written in policy listings) on whose behalf the code is running.

Chat

This application is the same as our main chat example, explained in more detail in Section 5.2. Here, thus, we only summarize the policies, formally in Listing 11, and informally as follows:

(1) all `User` fields except `name`, `avatar`, and `status` are private , i.e., not readable by other *users*;

(2) a `User` record cannot be edited or deleted by other *users*;

(3) `Users` whose status is "busy" are not visible (or searchable) by other *users*;

(4) a `Client` record cannot be edited or deleted by other *clients*;

(5) a Msg record cannot be edited or deleted by a *user* who did not send it;

(6) ChatRoom messages are not shown to *clients* without a logged-in user;

(7) ChatRoom messages containing "#private" are not shown to *users* who are not members of that room.

Figure 5-8 shows screenshots of three different clients interacting with the Chat application at the same time: one where the logged-in user is Bob, whose status is "busy" (Figure 5-8(a)), one where the logged-in user is Alice, whose status is something other than "busy" (Figure 5-8(b)), and one where no user is logged-in (Figure 5-8(c)). The same template, containing no knowledge of application's policies, was used to render all three clients. For example, the following snippets were used to render a room's members and messages:

```
{{#each room.members}}
  <div>{{this.name}}</div>
{{/each}}
```

```
{{#each room.messages}}
  <b><span>{{salute this.sender}}</span></b>: [...]
{{/each}}
```

, where `salute` is a helper function defined as `salute = (user) -> user?.name || "<unnamed>"`. These templates do require some defensive programming, e.g., to check for `null` references where they would normally not be expected, but they are completely policy-agnostic. The differences in the rendered views are a direct effect of the `User.find` policy (Listing 11), which literally removes all `User` records whose status is "busy" from individual client databases, so when those user records are referenced via other objects (e.g., `message.sender`), they cannot be found and `null` is returned. In the case of array fields (e.g., `room.members`), SUNNY automatically removes them.

Party Planner

PartyPlanner offers a simple functionality for organizing social events, but requires intricate policies for hiding sensitive data. Furthermore, policies themselves depends on sensitive data, creating circular dependencies which can lead to information leaks if not handled carefully. This particular example is often used by Yang et al. to motivate Jeeves [3, 168], an information flow framework for automatically enforcing privacy policies by using a constraint solver.

Listing 12 the data model and the security model defined for the PartyPlanner application. A `Party` can be created by any user. It contains fields for specifying name, time, and location, a boolean field (`finalized`) for indicating whether all party preparations have been finalized (used later for defining policies), a list of hosts, and a list of guests.

The following privacy policies must hold:

(1) all `User` fields except `name`, and `avatar` are private , i.e., not readable by other *users*;

(2) a `User` record cannot be edited or deleted by other *users*;

(3) only hosts are allowed to edit/delete a Party, but guest may remove themselves from the `guests` list.

(4) all `Party` fields must not be readable by *users* who are neither in `hosts` or `guests`;

```
user class User
  status: Text

record class Msg
  text: Text
  sender: User
  time: Val

record class ChatRoom
  name: Text
  members: set User
  messages: compose seq Msg

client class Client
  user: User
  selectedRooms: set ChatRoom

server class Server
  rooms: compose set ChatRoom
```

```
policy User,
  # ONLY for user records not equal to the client logged-in user
  precondition: (u) -> not u.equals(this.client?.user)
  # deny reads of private fields (all other than 'name' and 'status')
  read:        "! name, status": () -> return this.deny()
  # deny deletions and all writes
  update:      "*":              () -> return this.deny()
  destroy:                       () -> return this.deny()
  # hide "busy" Users
  find: (users) ->
    clnt = this.client
    filterNot users, (u) -> u.status=="busy" && !u.equals(clnt?.user)

policy Client,
  precondition: (c) -> c.user && !c.user.equals(this.client?.user)
  update:    "*": () -> return this.deny("can't edit other clients")
  destroy:        () -> return this.deny("can't delete other clients")

policy Msg,
  precondition: (m) -> !(m.sender && m.sender.equals(this.client?.user))
  update:    "*": () -> return this.deny("can't edit other's messages")
  destroy:        () -> return this.deny("can't delete other's messages")

policy ChatRoom,
  read: messages: (room, msgs) ->
    # don't show messages to clients without a logged-in user
    return this.deny() if not this.client?.user
    # allow full access to chat room members
    return this.allow() if room.members.contains this.client?.user
    # otherwise filter out messages containing '#private'
    return this.allow(filter msgs, (m) -> !/\#private\b/.test(m.text))
```

Listing 11: *Sunny.js* `Chat` application code listing: data and security models

(5) fields `hosts` and `guests` must additionally not be readable by guests if the party has not been finalized.

Notice the subtle interaction between the last two policies. Say, for example, Alice and Bob are hosting a surprise party for Eve. Eve has been added to the guest list, and the party has not been finalized yet. Because of Policy 5, to Eve, the guest list should appear as empty. Now applying Policy 4, all party information should, therefore, also be hidden from Eve. A naive implementation—one that executes policies in a privileged context (where all information is accessible)—on the other hand, would only check Policy 4, realize that Eve is indeed in the guest list, and, wrongly, allow access to the `name`, `location`, and `time` fields. Figure 5-9 shows that, in this scenario, *Sunny.js* correctly hides the sensitive information from Eve.

Social Network

In this example, we implement a simple social networking site, which we call SocNet, and focus how SUNNY polices can be used to highly customize and personalize privacy settings for individual users.

In addition to built-in name, email, and avatar fields, a `User` in SocNet also has a status message, a location, a network of labeled connections with other users, and a wall of posts. Connections in SocNet need not be symmetric; each user is free to add any other user to his/her network and label it with an arbitrary string (thus the `Text` → `User` type for the

(a) Bob's view

Sunny IRC
Create Room
Bob bob@mit.edu ▾

Bob
busy
Alice
working

online rooms:
Bob's room
Some other room
Alice's room

Bob's room
members
Bob
Alice
messages
Bob: hi
Alice: who is there?
Enter message
Send

Alice's room
members
Alice
messages
Enter message
Send

(b) Alice's view

Sunny IRC
Create Room
Alice alice@mit.edu ▾

Alice
working

online rooms:
Bob's room
Some other room
Alice's room

Bob's room
members
Alice
messages
<unnamed>: hi
Alice: who is there?
Enter message
Send

Alice's room
members
Alice
messages
Alice: #private only for members
Enter message
Send

(c) View of an anonymous (not logged-in) user

Sunny IRC
Create Room
<unnamed> Sign in ▾

Alice
working

online rooms:
Bob's room
Some other room
Alice's room

Bob's room
members
Alice
messages
Enter message
Send

Alice's room
members
Alice
messages
Enter message
Send

Figure 5-8: Views of three different users of the same Chat application

```
record class Party
  name: Text
  location: Text
  time: Val
  finalized: Bool
  hosts: set SunnyUser
  guests: set SunnyUser

client class Client
  user: SunnyUser
  selectedEvent: Party
```

```
policy SunnyUser,
  # WHEN the acting user is not the same as the user record
  precondition: (u) -> not u.equals(this.client?.user)
  # deny reads of fields other than 'name' and 'avatar'
  read:         "! name, avatar": () -> return this.deny()
  # deny deletions and all writes
  update:       "*":              () -> return this.deny()
  destroy:                        () -> return this.deny()

policy Party,
  # WHEN the acting user is not a host
  precondition: (party) -> not party.hosts.contains(this.client?.user)
  # allow users to remove themselves from guests, but no one else
  pull:
    "guests": (party, u) -> return this.allowIf(this.client?.user?.equals(u))
  # deny deletions and all writes
  update: "*": -> return this.deny()
  destroy:     -> return this.deny()

policy Party,
  # WHEN the acting user is neither a host nor a guest
  precondition: (party) ->
    u = this.client?.user
    !party.hosts.contains(u) && !party.guests.contains(u)
  # hide everything
  read:
    "name":      () -> return this.allow("<private party>")
    "location":  () -> return this.allow("<secret location>")
    "time":      () -> return this.allow("<unknown time>")
    "finalized": () -> return this.allow(false)
    "hosts":     () -> return this.allow([])
    "guests":    () -> return this.allow([])

policy Party,
  # WHEN the acting user is not a host and the party hasn't been finalized
  precondition: (party) ->
    !party.hosts.contains(this.client?.user) && !party.finalized
  # hide hosts and guests
  read:
    "hosts":  () -> return this.allow([])
    "guests": () -> return this.allow([])
```

Listing 12: *Sunny.js* PartyPlanner application code listing: data and security models

(a) Alice's view **(b) Eve's view**

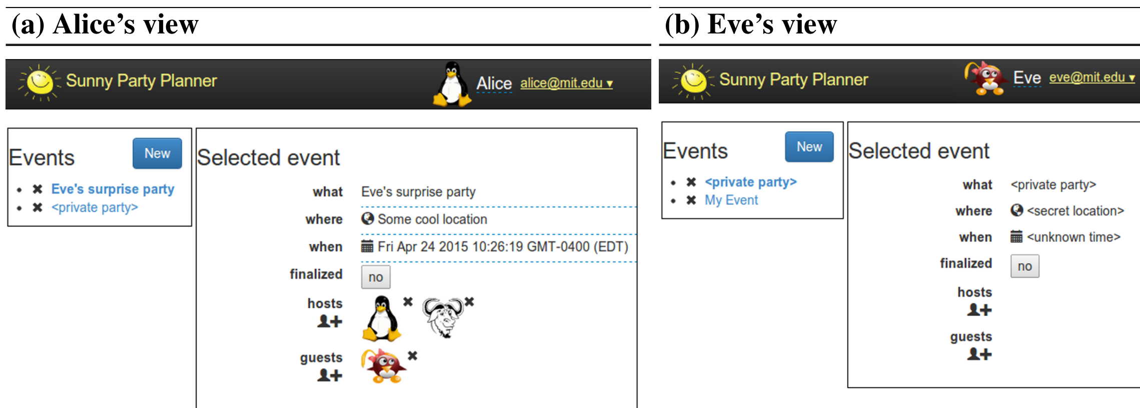


Figure 5-9: Views of two different users of the same PartyPlanner application

network field in Listing 13). A Post is authored by a single user, has a textual content, and a timestamp. A post is created from a single string, which then get converted to a list of Chunks, converting every occurrence of the “@username” pattern into a reference to a SocNet user (enclosed in a UserTagChunk record), and every occurrence of the “#tag” pattern into a tag reference (HashTagChunk).

The following generic policies are defined:

(1) all User fields except name, email, status, and avatar are private, i.e., not readable by other *users*;

(2) the email and status fields of a User record are readable only by *users* that are connections of that user record;

(3) a User record cannot be deleted or edited by other *users*, but allow posts to be added to the wall field;

(4) a Post record cannot be deleted or edited by a *user* who did not author the post.

These policies apply to all users and posts. Now assume Alice wants to customize her own privacy settings to meet her own specific needs. Specifically, she wants to hide some of her activities that might not be appropriate for her supervisors at work to see. In particular, to any user Alice labeled as “boss” in her network:

(5) Alice’s status should always appear as “working hard”;

(6) Alice’s location should always appear at “Stata Center” (which is her workplace);

(7) Alice’s network should not be shown;

(8) all Alice’s posts containing the #fun hashtag should be hidden;

(9) all posts on Alice’s wall containing the #fun hashtag should be hidden, except for those authored by the acting “boss” user (to avoid unnecessary suspicion).

The full listing of the SocNet data and security models, including Alice’s custom privacy policies, is given in Listing 13; as before, the event model is omitted, since it tends to be straightforward and boils down to simple record manipulations. Figure 5-10 shows screenshots of two different users, Bob and Carol, viewing Alice’s profile. In Alice’s network, Bob is labeled as “friends”, while Carol is labeled as “boss”, so Bob’s and Carol’s views of Alice’s profile are very different.

Today’s social networking web sites typically do not offer flexible mechanisms to their users for configuring privacy settings. A big technological reason for that is the inability of the popular web frameworks to automatically enforce arbitrary policies across the system. To implement Alice’s privacy rules in SocNet, by contrast, we only needed to write ~20 lines of SUNNY policy code. Although we do not expect end users to write SUNNY code just to specify their privacy settings, we do think that with the SUNNY engine in the back end, and an appropriate graphical user interface in the front end, the users would still have a more powerful and flexible mechanism for fine-tuning their policies, more so than what is already provided in mainstream social web sites.

```
user class User
  status: Text
  location: Text
  network: [Text, "User"]
  wall: seq "Post"

  hasConn: (u) ->
    return findIndex(
      this.network,
      (e) -> e[1].equals(u)
    ) != -1
  connKinds: (u) ->
    return map(filter(
      this.network,
      (e) -> e[1].equals(u)
    ), (e) -> e[0])

record (class Chunk)

record class TextChunk \
       extends Chunk
  text: Text
record class HashTagChunk \
       extends Chunk
  tag: Text
record class UserTagChunk \
       extends Chunk
  user: User

record class Post
  author: User
  content: Text
  time: Val
  body: compose seq Chunk
```

```
policy User, # WHEN the acting user is not the same as the user record
  precondition: (u) -> not u.equals(this.client?.user)
  read: "! name, email, status, avatar": () -> return this.deny()
        "email, status": (u) -> if !u.hasConn(this.client?.user) \
                                 then this.allow("") else this.allow()
  # deny record deletion and writes to all fields other than 'wall'
  destroy:      () -> return this.deny()
  update: "*":  () -> return this.deny()
  push: "wall": () -> return this.allow()
policy Post, # deny deletion and writes to users other than the post author
  precondition: (post) -> not post.author.equals(this.client?.user)
  update: "*": -> return this.deny()
  destroy:  () -> return this.deny()
# ========================== Alice's policies =============================
policy User, # WHEN the user record is Alice and the acting user is her boss
  precondition: (u) -> u.email == "alice@mit.edu" and
                       u.connKinds(this.client?.user).contains("boss")
  # return "professional"-looking values for status, location, and network
  read: "status":   () -> return this.allow("working hard")
        "location": () -> return this.allow("Stata Center")
        "network":  () -> return this.allow([["boss", this.client.user],
                                              ["best_friends", this.client.user]])
policy Post,
  # from each Alice's boss hide all posts that are either authored by Alice
  # or posted on Alice's wall that contain the '#fun' tag, except for post
  # authored by the acting user (to avoid suspicion)
  find: (posts) ->
    actingUser = this.client?.user
    alice = User.findOne(email: "alice@mit.edu")
    return this.allow() if !alice.connKinds(actingUser).contains("boss")
    return this.allow(filterNot posts, (post) ->
      (post.author.equals(alice) or alice.wall.contains(post)) and
      not post.author.equals(actingUser) and
      post.content.search("#fun") != -1)
```

Listing 13: *Sunny.js* SocNet application code listing: data and security models

(a) Bob's view of Alice's profile **(b) Carol's view of Alice's profile**

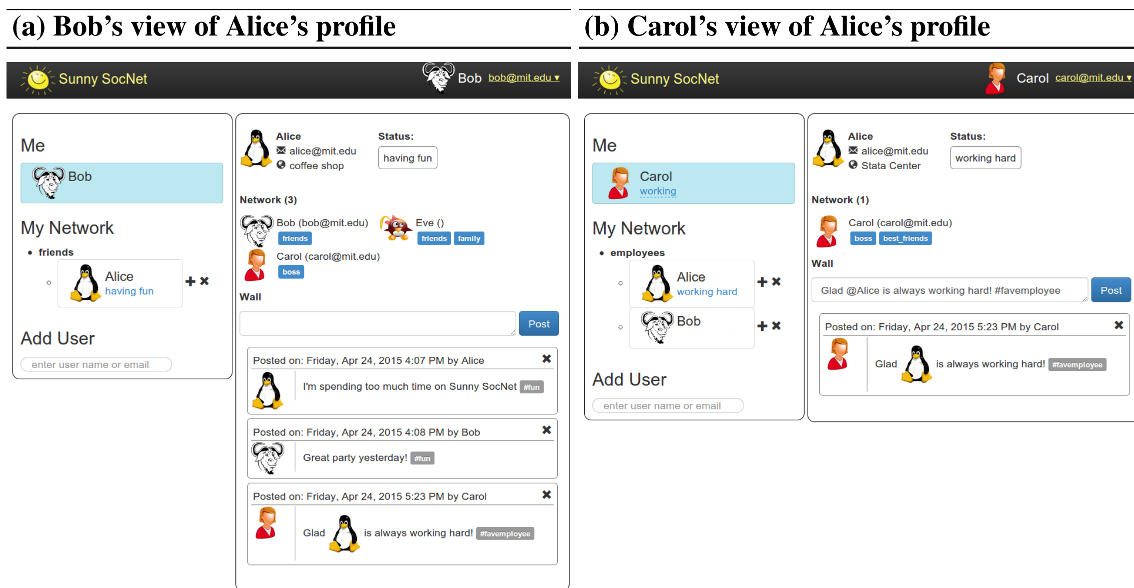


Figure 5-10: Views of two different users of the same SocNet application

5.8.2 Comparison with a Web Application in Meteor

We implemented the IRC example in Meteor, a framework designed specifically for fast development of modern, interactive web applications, and compared it to the presented implementation in SUNNY. We make no strong claims based on this simple case study; we only quantify the effort needed to develop this example (in terms of the number of lines of code) and report on our experiences using both systems.

SUNNY and Meteor share the idea that a single data model should be used across the application, even in a distributed setting, and that any updates to it should be automatically propagated to all connected nodes. The main difference is in the representation of the shared data. Meteor relies on MongoDB [5], a NoSQL database which stores data as untyped JSON documents, meaning that the database schema is fully dynamic, and can change anytime. In contrast, models in SUNNY are strongly typed, which is essential to achieving a precise code analysis, but also necessary for implementing various tools, such as the GUI builder.

For comparison, our implementation of the IRC example in Meteor is given in Listing 14. The number of lines of code is about the same, but we believe that SUNNY models tend to be more readable because they make much more explicit both conceptual and structural information about the system. Furthermore, because all the concepts in SUNNY models have a precisely defined semantics, these models can serve as a good documentation on their own.

Another consequence of lack of structure in the Meteor code is the tendency to tightly couple business logic and GUI code. For example, events are often directly tied to JavaScript UI events (e.g., lines 6, 24, 44), and their handlers can fetch values directly from the DOM elements (e.g., lines 7, 8, 25, 26).

We believe that our model-based paradigm has a clear advantage over the dynamic NoSQL model when it comes to applying tools and techniques for various code analyses. In other words, Meteor is mainly focused on providing a platform where data updates are automatically propagated to relevant clients; we are also concerned about the software engineering aspects of the system, its overall design, correctness, testability, and analyzability, as described in Section 5.6. Most of the ideas from that section would be difficult to apply to Meteor programs.

5.8.3 Limitations

Scalability

The current SUNNY architecture is not readily amenable to horizontal scaling, that is, scaling out from one physical web server to multiple web servers simply by adding more nodes or deploying the application on a cluster of servers. The main reason for that is the centralized data model, which assumes a single central database server, and the sequential semantics of event execution, which requires that all events are executed atomically and in complete isolation from each other. To partially overcome these issues, a distributed database could be used instead, and SUNNY events could be implemented using an appropriate transaction mechanism supported by the database.

```
Rooms = new Meteor.Collection("rooms");

if (Meteor.isClient) {
 // Create Room
 Template.irc.events({
  'click input.createRoom': function () {
   var roomName = $("#roomName").val();
   var userName = $("#userName").val();
   // Ignore empty names
   if (roomName) {
    var room = Rooms.findOne({name: roomName});
    if (room == undefined) {
     Rooms.insert({name: roomName, creator: userName,
                   members: [userName], messages: []});
     Session.set("userName", userName);
     Session.set("user_id", userName);
    }
   }
  }
 });

 // Join Room
 Template.irc.events({
  'click input.joinRoom' : function () {
   var roomName = $("#roomName").val();
   var userName = $("#userName").val();
   // Check if room exist
   var room = Rooms.findOne({name: roomName});
   if (room != undefined) {
    // Check if name is taken
    var userRoom = Rooms.findOne({
     members: { $in: [userName] }});
    if (userRoom == undefined) {
     Rooms.update({_id: room._id},
                  {$push: {members: userName}});
     Session.set("userName", userName);
    }
   }
  }
 });

 // Send a Message
 Template.irc.events({
  'click input.send': function () {
   var userName = Session.get("userName");
   // Create a message to be sent
   var message = Session.get("userName") + ": " +
         $("#message").val() + "\n";
   var room = Rooms.findOne({
    members: { $in: [Session.get("userName")] }});
   Rooms.update({_id: room._id},
                {$push: {messages: message}});
  }
 });
```

Listing 14: Implementation of the IRC example in Meteor

The current SUNNY server-side implementation additionally suffers from sequentially reacting to data changes to propagate data updates to all connected clients. This step can be expensive because it involves checking all applicable policies. An improved implementation, whoever, can easily mitigate this problem by checking policies and updating clients in parallel, because policy code is always read only.

Inefficient Queries

SUNNY uses an object-oriented data model, and intentionally hides the underlying database query language. This mindset often leads to writing code that first loads a large set of objects from the database, only to next apply the standard `map` and `filter` functions over it. This approach can be significantly less efficient than using the database-supported `SELECT` and `WHERE` clauses.

Mitigation techniques include implementing a richer object-oriented API which matches the expressive power of the underlying database query language. Alternatively, the existing SUNNY API function could be implemented lazily, so that they collect as many calls as possible before translating them into a single database query. Finally, Cheung et al. show how program synthesis can be used to automatically synthesize database queries from object-oriented code using traditional object-relational mapping libraries [33, 34].

Drawbacks of Declarative Policies

While the declarative nature of SUNNY policies makes it possible to separate them cleanly from the application code, it does not prevent unexpected interactions between different policies. Assume the chat application and the "find" policy for hiding busy users from Listing 11, as well as that a `User` record Bob whose status is "busy" exists in the database. If at that point a new client connects and attempts to log in using Bob's credentials (i.e., execute `client.user = User.find(user: "Bob", password: "123")`), the operation will fail, because the "find" policy will prevent any client without a logged in user from seeing any "busy" user records (including Bob's). While executing policies in the same client context (which the default behavior in SUNNY) is often desirable and typically prevents unintentional information leaks, in this particular case the only workaround is to execute the `User.find` operation in the privileged context (for which SUNNY provides special language constructs).

5.9 Related Work

5.9.1 Event-Driven Programming

There are two main styles of building distributed systems: (1) asynchronous or event-driven, and (2) using threads and locks. There has been a lot of debate over whether one is superior to the other. Dabek et al. [39] convincingly argue that event-driven systems lead to more robust software, offer better performance, and can be easily programmed given an appropriate framework or library.

There exist many frameworks or libraries designed to support the event-driven programming paradigm. They are all similar to ours in that they provide an easy, event-driven way of writing distributed applications. Meteor, previously discussed, is one such library; another popular one is *Node.js* [157]. They eliminate the need to manually manage threads and event queues, but typically do not provide an abstract model of the system, amenable to formal analysis.

Approaches like *TinyGALS* [32] and ESP* [152], which focus on programming embedded systems, also provide special support for events. The *TinyGALS* framework additionally offers a structured model of the whole system and also implements global scheduling and event handling. It uses code generation to translate models into an executable form, unlike ESP* which embeds the Statechart [71] concepts in a high-level general-purpose (Java-like) language. ESP* mainly focuses on correctly implementing the Statechart semantics.

Tasks [57] provides language support for complex tasks, which may consist of multiple asynchronous calls, to be written as a single sequential unit (procedure), without having to explicitly register callback functions. This is achieved by a translation of such sequential procedures to a continuation-passing style code. Tasks is not concerned with specifying the top-level event model of the system, and is orthogonal to our framework.

The *implicit invocation* mechanism [61] provides a formal way of specifying and designing event-driven systems. Events and the bindings of events to methods (handlers) are decoupled and specified independently (so that the handler can be invoked "implicitly" by the runtime system). This provides maximum flexibility but can make systems difficult to understand and analyze. In our framework, we decided to take the middle ground by requiring that one event handler (the most essential one, the one that implements the business logic of the system by updating the core data model) is explicitly bound to an event.

Functional Reactive Programming [51] is a programming paradigm for working with mutable values in a functional programming language. Its best known application is in fully functional, declarative programming of graphical user interfaces that automatically react to changing values, both continuous (like time, position, velocity) and discrete (also called events). Implementations of this paradigm include Ur/Web [35] (a domain-specific, statically typed functional programming language, design for single-tier web development), Elm [38] (a standalone language that compiles to HTML, CSS and JavaScript) and Flapjax [107] (an implementation embedded in JavaScript, designed specifically to work with Ajax). Our approach to self-updating GUI can also be seen as a specific application of functional reactive programming.

5.9.2 Data-Centric Programming

Another, increasingly popular, method for specifying distributed data management is using Datalog-style declarative rules and has been applied in the domain of networking (e.g., Declarative Networking [101], Overlog [102]), distributed computing (e.g., the BOOM project [19], Netlog [68]), and also web applications (e.g., Webdamlog [11], Reactors [55], Hilda [167], Active XML [10]). The declarative nature of Datalog rules makes this method particularly suitable for implementing intrinsically complicated network protocols (or other algorithms that have to maintain complex invariants); manually writing an imperative procedure that correctly implements the specification and respects the invariants is a lot more difficult in this case.

By contrast, we focus on applications that boil down to simple data manipulation in a distributed environment (which constitutes a large portion of today's web applications), and one of our goals is to provide a programming environment that is easy to use by even non-expert programmers who are already familiar with the object-oriented paradigm.

5.9.3 Code Generation and Program Synthesis

The idea of using increasingly higher-level abstractions for application programming has been a common trend since the 1950s and the first Autocoder [66] systems which offered an automatic translation from a high-level symbolic language into actual (machine-level) object code. The main argument is that software engineering would be easier if programmers could spend their time editing high-level code and specifications, rather than trying to maintain optimized programs [22]. Our approach is well aligned with this idea, with a strong emphasis on a particular and widely used domain of web application programming.

Executable UML [106] (xUML) also aims to enable programming at a high level of abstraction by providing a formal semantics to various models in the UML family. Model-driven development approaches based on xUML (e.g., [108, 109]) translate the UML diagrams by generating code for the target language, and then ensure that the diagrams and the code are kept in sync. Our system is conceptually similar, and it also follows the model-driven development idea, but instead of using code generation to translate models (diagrams) to code, we want to make models first-class citizens and to have an extensive framework that implements the desired semantics by essentially interpreting the models at runtime (an actual implementation may generate and evaluate some code on the fly to achieve that). Minimizing the amount of auto-generated code makes the development process more convenient, as there is no need to regenerate the code every time the model changes.

Similar to code generation, the main goal of program synthesis is also to translate code from a high-level (often abstract, declarative) form to a low-level executable language. Unlike code generation, however, a simple translation algorithm is often not sufficient; instead, more advanced (but typically less efficient) techniques (e.g., search algorithms, constraint solving, etc.) have to be used. The state of the art in program synthesis focuses on synthesizing programs from various descriptions, e.g,. sketches [150], functional specifications [93], input-output pairs [72], graphical input-output heaps [148], or first-order declarative pre- and post-conditions [99].

The core of our framework is a little further from the traditional program synthesis techniques; although it does aim to provide a high-level surface language for specifying/-modeling various aspects of the system (events, privacy policies, GUI templates), it does not perform any complex search-based procedure to synthesize a piece of code. Given the declarative and formal nature of our models, however, program synthesis is still relevant to this work, as it might be applied to implement some advanced extensions, e.g., to synthesize higher-level operations from basic events (as briefly discussed in Section 5.6).

5.9.4 Declarative Privacy Policies

In their most general form, policies are used to map each *user* (*subject*), *resource*(*object*) and *action* to a decision, and are consulted every time an action is performed on a resource by a user [95]. In our framework, resources correspond to fields, actions correspond to field accesses[4], and the user is the entity executing the action.

[4] Our policy language currently does not allow differentiating between reads and writes, but it could; we will consider adding that extension if we encounter examples where that distinction proves to be necessary.

Systems for checking and ensuring privacy policies are typically based either on Access Control Lists (ACL) or Information Flow (IF). ACLs attach a list of *permissions* to concrete *objects*, whereas IF specifies which flows (e.g., data flowing from variable `x` to variable `y`) are allowed in the system. In both cases, when a violation is detected, the operation is forbidden, for example by raising an exception.

Our security model is more in the style of access control lists, in the sense that we attach policies to statically defined fields (as opposed to arbitrary pieces of data), but it has a flavor of information flow as well, since we automatically check all data flowing to all different machines and ensure that no sensitive information is ever sent to a machine that does not have required permissions (which, in our system, means that there is no policy that explicitly restricts that access). Similar to the access modifiers in traditional object-oriented languages (e.g., `private`, `protected`, `public`, etc.), our model also focuses on specifying access permissions for various fields. However, the difference is that our permission policies are a lot more expressive and more flexible than static modifiers, and can also depend on the dynamic state of the program. In addition, they are completely decoupled from the data model, so the policies can be designed and developed independently.

Information flow systems either rely on sophisticated static analysis to statically verify that no violating flows can exist (e.g., Jif [126, 127]), or dynamically labeling sensitive data and tracking where it is flowing (e.g., RESIN [171] or Dytan [36]). Unlike most other information flow systems, Jeeves [168] allows policies that are specified declaratively and separately from the rest of the system, and instead of halting the execution when a violation is detected, it relies on a runtime environment to dynamically compute values of sensitive data before it is disclosed so that all policies are satisfied. This approach is similar to our serialization technique when we automatically hide the sensitive field values before the data is sent to a client.

Margrave [47, 59, 131] implements a system for analyzing policies. Similar to our system, Margrave policies are declarative and independent of the rest of the system (which they call "dynamic environment"). Their main goal, however, is to statically analyze policies against a given relational representation of the environment, and to check if a policy can be violated in any possible (feasible) scenario, whereas we are only interested in checking field accesses at runtime. To enable efficient analysis, the Margrave policy language is based on Datalog and is more restrictive than the first-order logic constraints that we allow in our policies.

Attribute-based access control (ABAC) adds attributes ($name \rightarrow value$ pairs) to any entity in the system (e.g., user, resource, subject, object, etc.) so that policies can be expressed in terms of those attributes rather than concrete entities. Our system can be viewed as an instantiation of this model: our fields can be seen as attributes, machines as subjects, and records as resources; both records and machines can have fields, and policies are free to query field values. Many other ABAC systems have been designed and implemented (e.g., [125, 164, 172]), each, however, using somewhat different model from the other. Jin et al. [84] recently proposed a formal ABAC model to serve as a standard, and used it to express the three classical access control models (discretionary [144], mandatory [142], and role-based [143]).

5.9.5 GUI Builders

Our dynamic template engine for building graphical user interfaces, combines two existing techniques: *data binding* and *templating*.

Data binding allows select GUI widget properties to be bound to concrete object fields from the domain data model, so that whenever the value of the field changes, the widget automatically updates its property. Changes can optionally be propagated in the other direction as well, that is, when the property is changed by the user, the corresponding field value gets updated simultaneously.

Templating, on the other hand, takes a free-form text input containing a number of special syntactic constructs supported by the engine which, at the time of rendering, get dynamically evaluated against the domain data model and get inlined as strings in the final output. Such constructs can include embedded expressions (formulas), control flow directives (if, for loops, etc.), or, in a general case, arbitrary code in the target programming language. This adds extra flexibility, as it allows generic programming features to be used in conjunction with static text, enabling widgets with dynamic layouts to be defined.

Even though existing data binding implementations (e.g., WPF and their textual UI layout language XAML [128] for .NET, UI binder [134] for Android, JFace [70] for Java, Backbone [133] for JavaScript) allow for textual widget templates, those templates are typically allowed to contain only simple embedded expressions (e.g., a path to an object's field), only at certain positions in the template (to provide bindings only for select widget properties). No control structures are allowed, which makes it difficult to design a widget that chooses from two different layouts depending on the state of the application. Conversely, existing template engines (e.g., ASP [103] for .NET, Haml [2] and ERB [7] for Ruby, FreeMarker [165] for Java) provide all that extra flexibility, but do not preserve data bindings, making it difficult to push changes to the client when the model changes.

In this work, we combine these two techniques, to achieve the flexibility of generic template engines and still have the luxury of pushing the changes to the clients and automatically re-rendering the UI. The main reason why that makes the problem more difficult than the sum of its parts is the fact that formulas in the template can evaluate to arbitrary elements of the target language (e.g., HTML), including language keywords, special symbols, tag names, etc. This is unlike the existing UI frameworks with data-bindings, where all bindings are assigned to (syntactically strictly defined) widget properties.

5.10 Conclusion

Advances in web frameworks have made it much easier to develop attractive, featureful web applications. Most of those efforts are, however, mainly concerned with programming servers and their clients in isolation, providing only a set of basic primitives for intercommunication between the two sides, thus imposing a clear boundary. We believe that there is an entire class of web applications, and distributed programs in general, for which that boundary can be successfully erased and removed from the conceptual programming model the programmer has to bear in mind. SUNNY is a generic programming platform for developing programs that fall into that class.

The main contribution of SUNNY lies in its architecture that enables the clear separation between four main concerns of any web applications: data, reactive GUI, events, and security. In traditional web frameworks these concerns are often cross-cutting, and thus difficult to implement independently from each other. The programmer, therefore, must ensure that the implementation of each one of them fulfills the requirements of every one of them (as opposed to implementing them in complete isolation). As an example, every GUI rendering piece of code must ensure that no sensitive information will be revealed to unauthorized clients (as opposed to globally defining rules for hiding sensitive data). Furthermore, to achieve fully reactive clients in the presence of dynamic policies for hiding sensitive data, the reactive GUI component must be aware of policies, per client policy dependencies on persisted data elements, and data updates.

To the best of our knowledge, SUNNY is the first framework that achieves fully reactive clients in the presence of dynamic policies for hiding sensitive data. Other tools either automatically enforce privacy policies (e.g., [63, 154, 168]), or implement reactive clients (e.g., [4, 35, 38, 107]). Putting these two features together creates non-trivial challenges. SUNNY achieves this by tracking per-client dynamic dependencies between policies and persisted data elements, so that when those data elements change, corresponding policies can be re-evaluated and, subsequently, the associated clients updated accordingly. This, in turn, demands certain implementation strategies be used. If pure information-flow is used to enforce policies, where labels are attached to concrete data values, it might not be possible to always associate those concrete values back to their persisted origins. If, on the other side, pure functional reactive programming is used, aside from the well-known issues with merging data updates from different sources into a single stream, it is not clear how one would implement SUNNY's find policies which can hide entire records from clients. For example, our policy for hiding users whose status is "busy" from Listing 11 must create a dependency between Alice's client and Bob's `status` field, in spite of Alice's client (Figure 5-8(b)) not having a single reactive GUI element referring to Bob's status at the time). In SUNNY, we show how this can be achieved when policies are defined for fields and checked at field accesses, and reactive clients implemented by having on each client a database replica from which all client-specific sensitive data has been first filtered out by the server. We believe that the combination of reactive programming while automatically enforcing policies is important, especially when viewed through the lens of declarative programming, because it neither compromises desired features of modern web applications, nor does it increase the programmer's burden to achieve the same.

Chapter 6

Conclusion

Increasing the level of abstraction of general-purpose languages, both programming and specification, has been a research challenge for a long time. As computers grow in power, new approaches, previously considered hopelessly intractable, find application, and become everyday tools in a variety of computer science domains. To maintain this trend, it is important that the cutting-edge technologies, usually developed for highly specialized purposes, are being generalized and made available to as wide audience as possible. While primarily focusing on advancing the state of declarative programming, this thesis was also driven by this goal.

First, this thesis presented Alloy*, the first general-purpose constraint solver capable of automatically and efficiently solving higher-order constraints over bounded relational domains. We have demonstrated how a variety of complex algorithms, typically implemented in an ad hoc fashion, can elegantly be recast as higher-order constraint solving problems, and efficiently solved in our general-purpose framework. Just the same first-order solvers have evolved as standalone components, while at the same time growing in popularity to become ubiquitously used a black boxes in a wide variety of systems, we believe the main value of our generalization rests on the potential for future development of tools that exploit it.

Second, to promote future development of tools that exploit higher-order constraint solving capabilities of Alloy*, this thesis presented αRby, a deep embedding of Alloy (the specification language of Alloy*) into Ruby. The Alloy language has found its uses within both the modeling community (for specifying and analyzing abstract domains), and the developer community (for translating various problems into Alloy and building tools on top of it). Unlike other approaches, αRby, by language design, aims to bring benefits to both of these communities, allowing the modelers to use the scripting features of Ruby and the developers to use the constraint solving features of Alloy*. More broadly, the hope is that the main idea behind αRby, namely unifying an implementation and a specification language into one, will lead to rethinking the traditional approach to specification languages, discovering new ways of integrating them with mainstream programming languages, and thus making them available and more accessible to a wider audience of programmers.

Third, SUNNY demonstrates how development of reactive web applications, a task often reserved for experienced developers, can be abstracted into a much simpler model-based paradigm, and thus made accessible to beginner programmers. Beyond this point, SUNNY

shows how declarative ideas can successfully be applied to web programming, allowing for a clean separation of concerns (such as data, events, security policies, and GUI) known to be badly intertwined in a typical web development setting. This thesis offers a formal semantics of the SUNNY paradigm, two proof-of-concept implementations, and a broad discussion of benefits. Much research still lies ahead. An ongoing project is focusing on improving scalability of the SUNNY platform by reimplementing it on top of the Actor model; another project is aiming to translate SUNNY models into Alloy so that SUNNY applications can be automatically formally analyzed for bugs and security violations (by examining all possible event interleavings); finally, powerful GUI builders are needed to replace writing GUI templates by hand. We believe that the increasing demand for web applications targeting relatively small user groups justifies our choice of somewhat sacrificing scalability to achieve a clean separation of concerns and a simple sequential semantics, significantly reducing the burden of developing such applications from scratch.

Overall, the hope is that this thesis offered convincing arguments in favor of declarative programming, be it (1) in the traditional setting of directly executing logic-based specifications in the context complex data structures, replacing manually implementing high-complexity algorithms, or (2) in a completely novel, domain-specific environment, where a model-based paradigm can be used to disentangle the world of web programming.

Bibliography

[1] Alloy* Home Page. http://alloy.mit.edu/alloy/hola.

[2] Haml template engine for Ruby. http://haml.info.

[3] Jeeves Home Page. http://projects.csail.mit.edu/jeeves.

[4] Meteor - Pure JavaScript web framework. http://meteor.com.

[5] MongoDB home page. http://www.mongodb.org/.

[6] Ruby on Rails web framework. http://rubyonrails.org/.

[7] Ruby's native templating system. http://ruby-doc.org/stdlib-1.9.3/libdoc/erb/rdoc/ERB.html.

[8] SyGuS github repository. https://github.com/rishabhs/sygus-comp14.git.

[9] Template engine for web applications. http://en.wikipedia.org/wiki/Template_engine_%28web%29.

[10] S. Abiteboul, O. Benjelloun, and T. Milo. Positive active XML. In *Proceedings of the Symposium on Principles of Database Systems*, pages 35–45, 2004.

[11] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *Proceedings of the Symposium on Principles of Database Systems*, pages 293–304, 2011.

[12] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6), 2010.

[13] J.R. Abrial and A. Hoare. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[14] Martin Aigner and Günter M Ziegler. Turán's graph theorem. In *Proofs from THE BOOK*, pages 183–187. Springer, 2001.

[15] How to think about an Alloy model: 3 levels. http://alloy.mit.edu/alloy/tutorials/online/sidenote-levels-of-understanding.html.

[16] Alloy: A language and tool for relational models. `http://alloy.mit.edu/alloy`.

[17] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE, 2013.

[18] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis competition report, 2014. `http://sygus.seas.upenn.edu/files/sygus_extended.pdf`.

[19] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J.M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the European Conference on Computer Systems*, pages 223–236, 2010.

[20] R.D. Arthan and Lovelace Road. Undefinedness in Z: Issues for Specification and Proof. In *CADE-13 Workshop on Mechanization of Partial Functions*. Springer, 1996.

[21] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Report A–1978–4.

[22] Robert Balzer, Thomas E. Cheatham, Jr., and Cordell Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, 16(11):39–45, 1983.

[23] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *lncs*, pages 364–387. Springer, 2006.

[24] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the International Conference on Computer Aided Verification*, pages 515–518, 2004.

[25] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[26] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.

[27] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[28] Kent Beck. *Test-Driven Development*. Addison-Wesley, 2003.

[29] Armin Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

[30] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *SMT Workshop at IJCAR*, volume 20, 2012.

[31] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley, 1995.

[32] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. TinyGALS: a programming model for event-driven embedded systems. In *Proceedings of the Symposium on Applied Computing*, pages 698–704, 2003.

[33] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 931–942, New York, NY, USA, 2014. ACM.

[34] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 3–14, New York, NY, USA, 2013. ACM.

[35] Adam Chlipala. Ur/web: A simple model for programming the web. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 153–165, New York, NY, USA, 2015. ACM.

[36] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[37] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[38] Evan Czaplicki. Elm: Concurrent FRP for functional GUIs. 2012.

[39] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *Proceedings of the SIGOPS European Workshop*, pages 186–189, 2002.

[40] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Automated Deduction–CADE-21*, pages 183–198. Springer, 2007.

[41] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *lncs*, pages 337–340. Springer, 2008.

[42] Greg Dennis. *A Relational Framework for Bounded Program Verification*. PhD thesis, MIT, 2009.

[43] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *Verified Software: Theories, Tools, Experiments*, pages 130–145. Springer, 2008.

[44] Gregory David Dennis. *A relational framework for bounded program verification*. PhD thesis, Massachusetts Institute of Technology, 2009.

[45] Edsgar W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, September 1968.

[46] Edsgar W. Dijkstra. Notes on structured programming. In O.-J. Dahl, C.A.R. Hoare, and E.W. Dijkstra, editors, *Structured Programming*. Academic Press, New York, 1972.

[47] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Proceedings of the International Joint Conference on Automated Reasoning*, pages 632–646, 2006.

[48] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the International Conference on Computer Aided Verification*, pages 81–94, 2006.

[49] Jonathan Edwards. Coherent reaction. In *Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 925–932, 2009.

[50] Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5, 2005.

[51] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the International Conference on Functional Programming*, pages 263–273, 1997.

[52] Paul Erdos and Alfred Renyi. On the evolution of random graphs. *Mathematical Institute of the Hungarian Academy of Sciences, 5: 17-61*, 1960.

[53] William M. Farmer. Reasoning about partial functions with the aid of a computer. *Erkenntnis*, 43, 1995.

[54] Joao F Ferreira, Alexandra Mendes, Alcino Cunha, Carlos Baquero, Paulo Silva, Luís Soares Barbosa, and José Nuno Oliveira. Logic training through algorithmic problem solving. In *Tools for Teaching Logic*, pages 62–69. Springer, 2011.

[55] J. Field, M.C. Marinescu, and C. Stefansen. Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications. *Theoretical Computer Science*, 410(2):168–201, 2009.

[56] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *Journal of functional programming*, 12(02):159–182, 2002.

[57] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*, pages 134–143, 2007.

[58] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th ICSE*, pages 196–205. ACM, 2005.

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

[60] Juan Pablo Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo Fabian Frias. Analysis of invariants for efficient bounded verification. In *ISSTA*, pages 25–36. ACM, 2010.

[61] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the Formal Software Development Methods*, pages 31–44, 1991.

[62] Yeting Ge and Leonardo De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification*, pages 306–320. Springer, 2009.

[63] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 47–60, Hollywood, CA, 2012. USENIX.

[64] Milos Gligoric and Rupak Majumdar. Model checking database applications. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 549–564, 2013.

[65] Milos Gligoric, Peter C. Mehlitz, and Darko Marinov. X10X: Model checking a new programming language with an "old" model checker. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 11–20, 2012.

[66] Roy Goldfinger. The IBM type 705 autocoder. In *Papers presented at the Joint ACM-AIEE-IRE Western Computer Conference*, pages 49–51, 1956.

[67] D. Gries and F.B. Schneider. *A logical approach to discrete math*. Texts and monographs in computer science. Springer-Verlag, 1993.

[68] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. *Proceedings of the International Conference on Practical Aspects of Declarative Languages*, pages 88–103, 2010.

[69] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[70] J.L. Guojie. *Professional Java Native Interfaces with SWT/JFace*. Wiley, 2006.

[71] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[72] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 317–328, 2011.

[73] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *ACM SIGPLAN Notices*, volume 46. ACM, 2011.

[74] Ian Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989.

[75] C. A. R. Hoare. An Overview of Some Formal Methods for Program Design. *IEEE Computer*, 20(9):85–91, 1987.

[76] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a sat solver. *STTT*, 10(6):503–520, 2008.

[77] Daniel Jackson. Micromodels of software: Lightweight modelling and analysis with alloy, 2002.

[78] Daniel Jackson. *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[79] Daniel Jackson, Martyn Thomas, Lynette I Millett, et al. *Software for Dependable Systems: Sufficient Evidence?* National Academies Press, 2007.

[80] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.

[81] Jean-Yves and Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[82] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[83] Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. Hmc: Verifying functional programs using abstract interpreters. In *Computer Aided Verification*, pages 470–485. Springer, 2011.

[84] Xin Jin, Ram Krishnan, and Ravi Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *Proceedings of the Data and Applications Security and Privacy*, pages 41–55. 2012.

[85] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[86] Cliff B. Jones. Reasoning about partial functions in the formal development of programs. *Electron. Notes Theor. Comput. Sci.*, 145, January 2006.

[87] Cliff B. Jones and Matthew J. Lovert. Semantic Models for a Logic of Partial Functions. *Int. J. Software and Informatics*, 5(1-2), 2011.

[88] Eunsuk Kang and Daniel Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In *Proceedings of the 1st international conference on Abstract State Machines, B and Z*, ABZ '08, Berlin, Heidelberg, 2008. Springer-Verlag.

[89] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[90] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. *ACM SIGPLAN Notices*, 2012.

[91] Viktor Kuncak and Daniel Jackson. Relational analysis of algebraic datatypes. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 207–216. ACM, 2005.

[92] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In *CAV*, pages 430–433, 2010.

[93] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 316–329, 2010.

[94] Darya Kurilova and Derek Rayside. On the simplicity of synthesizing linked data structure operations. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences*, pages 155–158. ACM, 2013.

[95] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[96] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul A. Agha. A framework for state-space exploration of Java-based actor programs. In *Proceedings of the International Conference on Automated Software Engineering*, pages 468–479, 2009.

[97] Daniel Le Berre, Anne Parrain, M Baron, J Bourgeois, Y Irrilo, F Fontaine, F Laihem, O Roussel, and L Sais. Sat4j-a satisfiability library for java, 2006.

[98] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *lncs*, pages 348–370. Springer, 2010.

[99] K. Rustan M. Leino and Aleksandar Milicevic. Program extrapolation with Jennisys. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 411–430, 2012.

[100] K Rustan M Leino and Michał Moskal. Co-induction simply: Automatic co-inductive proofs in a program verifier. Technical report, Technical Report MSR-TR-2013-49, Microsoft Research, 2013.

[101] B.T. Loo, T. Condie, M. Garofalakis, D.E. Gay, J.M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[102] B.T. Loo, T. Condie, J.M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Operating Systems Review*, volume 39, pages 75–90, 2005.

[103] M. MacDonald. *Beginning ASP.NET 4.5 in C#*. Apressus Series. Apress, 2012.

[104] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *Automated Software Engineering, 2001.*, pages 22–31. IEEE, 2001.

[105] Yukio Matsumoto and K Ishituka. Ruby programming language, 2002.

[106] S.J. Mellor and M.J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Object Technology Series. Addison-Wesley, 2002.

[107] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–20, 2009.

[108] D. Milicev. *Model-Driven Development with Executable UML*. Wrox Programmer to Programmer. Wiley, 2009.

[109] D. Milićev. Towards understanding of classes versus data types in conceptual modeling and UML. *Computer Science and Information Systems*, 9(2):505–539, 2012.

[110] Aleksandar Milicevic. Red github repository. `https://github.com/aleksandar.milicevic/red`.

[111] Aleksandar Milicevic. Sunny.js github repository. `https://github.com/aleksandar.milicevic/sunny.js`.

[112] Aleksandar Milicevic, Ido Efrati, and Daniel Jackson. αRby—An Embedding of Alloy in Ruby. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 56–71. Springer, 2014.

[113] Aleksandar Milicevic and Daniel Jackson. Preventing arithmetic overflows in alloy. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *Lecture Notes in Computer Science*, pages 108–121. Springer Berlin Heidelberg, 2012.

[114] Aleksandar Milicevic and Daniel Jackson. Preventing arithmetic overflows in alloy. *Science of Computer Programming*, 94:203–216, 2014.

[115] Aleksandar Milicevic, Daniel Jackson, Milos Gligoric, and Darko Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 17–36. ACM, 2013.

[116] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A Higher-Order Relational Constraint Solver. Technical report, MIT-CSAIL-TR-2014-018, Massachusetts Institute of Technology, 2014.

[117] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: A Higher-Order Relational Constraint Solver. In *Proceedings of the International Conference on Software Engineering*, ICSE, 2015.

[118] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520, 2011.

[119] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In *Proceedings of the International Conference on Software Engineering*, pages 511–520, 2011.

[120] Vajih Montaghami and Derek Rayside. Extending Alloy with Partial Instances. In *Abstract State Machines, Alloy, B, VDM, and Z*, pages 122–135. Springer, 2012.

[121] Vajih Montaghami and Derek Rayside. Staged evaluation of partial instances in a relational model finder. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 318–323. Springer, 2014.

[122] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), 1988.

[123] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, Inc., 2nd edition, 1998. First edition 1990.

[124] J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3), December 1987.

[125] Tim Moses et al. Extensible access control markup language (XACML) version 2.0. *Oasis Standard*, 2005.

[126] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[127] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[128] A. Nathan. *WPF 4: Unleashed*. Sams, 2010.

[129] Joseph P Near and Daniel Jackson. Rubicon: bounded verification of web applications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 60. ACM, 2012.

[130] Tim Nelson, Salman Saghafi, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Aluminum: principled scenario exploration through minimality. In *ICSE*, pages 232–241. IEEE Press, 2013.

[131] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave tool for firewall analysis. In *Proceedings of the International Conference on Large Installation System Administration*, pages 1–8, 2010.

[132] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[133] A. Osmani. *Developing Backbone.js Applications*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2013.

[134] J. Ostrander. *Android UI Fundamentals: Develop & Design*. Pearson Education, 2012.

[135] D. L. Parnas. Predicate logic for software engineering. *IEEE Trans. Softw. Eng.*, 19, September 1993.

[136] Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006. ACM, 2009.

[137] Derek Rayside, Vajihollah Montaghami, Francesca Leung, Albert Yuen, Kevin Xu, and Daniel Jackson. Synthesizing iterators from abstraction functions. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 31–40, 2012.

[138] Ruby Java Bridge. `http://rjb.rubyforge.org/`.

[139] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

[140] Nicolás Rosner, Juan Galeotti, Santiago Bermúdez, Guido Marucci Blas, Santiago Perez De Rosso, Lucas Pizzagalli, Luciano Zemín, and Marcelo F Frias. Parallel bounded analysis in code with rich invariants by refinement of field bounds. In *ISSTA*, pages 23–33. ACM, 2013.

[141] Hesam Samimi, Ei Darli Aung, and Todd D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[142] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[143] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[144] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[145] Andreas Schaad and Jonathan D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *SACMAT*, pages 13–22, 2002.

[146] Egbert Schlungbaum. Model-based user interface software tools current state of declarative models. Technical report, Graphics, visualization and usability center, Georgia institute of technology, GVU tech report, 1996.

[147] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th PLDI*, pages 15–26. ACM, 2013.

[148] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *Proceedings of the Symposium on the Foundations of Software Engineering*, pages 289–299, 2011.

[149] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic common lisp. *IRCS Technical Reports Series*, 1993.

[150] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, 2006.

[151] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1988.

[152] Vugranam C. Sreedhar and Maria-Cristina Marinescu. From statecharts to ESP: Programming with events, states and predicates for embedded systems. In *Proceedings of the International Conference on Embedded Software*, pages 48–51, 2005.

[153] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *PLDI 2011*, pages 492–503. ACM, 2011.

[154] Deian Stefan, Alejandro Russo, John C Mitchell, and David Mazières. *Flexible dynamic information flow control in Haskell*, volume 46. ACM, 2011.

[155] Bill Stoddart, Steve Dunne, and Andy Galloway. Undefined Expressions and Logic in Z and B. *Formal Methods in System Design*, 15, 1999.

[156] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the International Conference on Formal Techniques for Distributed Systems*, pages 219–234, 2012.

[157] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to build high-performance network programs. *Internet Computing, IEEE*, 14(6):80 –83, 2010.

[158] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60. ACM, 2011.

[159] Nikolai Tillmann, Michal Moskal, Jonathan De Halleux, Manuel Fahndrich, and Sebastian Burckhardt. Touchdevelop: app development on mobile devices. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 39. ACM, 2012.

[160] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT, 2008.

[161] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152. ACM, 2013.

[162] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[163] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[164] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *Proceedings of the Workshop on Formal Methods in Security Engineering*, pages 45–55, 2004.

[165] N. Willy. *Freemarker*. Culp Press, 2012.

[166] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the International Conference on Software Engineering*, pages 611–620, 2011.

[167] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *Proceedings of the International Conference on Data Engineering*, pages 32–32, 2006.

[168] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 85–96, 2012.

[169] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. *ACM SIGPLAN Notices*, 2012.

[170] Kuat Yessenov. A Light-weight Specification Language for Bounded Program Verification. Master's thesis, MIT, 2009.

[171] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the Symposium on Operating Systems Principles*, pages 291–304, 2009.

[172] Eric Yuan and Jin Tong. Attributed based access control (ABAC) for web services. In *IEEE International Conference on Web Services*, 2005.