Debugging Relational Declarative Models with Discriminating Examples

by

Vajihollah Montaghami

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Marsha Chechik Professor
Supervisor(s)	Derek Rayside Assistant Professor
Internal Member	Krzysztof Czarnecki Professor
Internal Member	Vijay Ganesh Assistant Professor
Internal-external Member	Richard Trefler Associate Professor

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Models, especially those with mathematical or logical foundations, have proven valuable to engineering practice in a wide range of disciplines, including software engineering. Models, sometimes also referred to as logical specifications in this context, enable software engineers to focus on essential abstractions, while eliding less important details of their software design. Like any human-created artifact, a model might have imperfections at certain stages of the design process: it might have internal inconsistencies, or it might not properly express the engineer's design intentions.

Validating that the model is a true expression of the engineer's intent is an important and difficult problem. One of the key challenges is that there is typically no other written artifact to compare the model to: the engineer's intention is a mental object. One successful approach to this challenge has been automated example-generation tools, such as the Alloy Analyzer. These tools produce examples (satisfying valuations of the model) for the engineer to accept or reject. These examples, along with the engineer's judgment of them, serve as crucial written artifacts of the engineer's true intentions.

Examples, like test-cases for programs, are more valuable if they reveal a discrepancy between the expressed model and the engineer's design intentions. We propose the idea of *discriminating examples* for this purpose. A discriminating example is synthesized from a combination of the engineer's expressed model and a machine-generated *hypothesis* of the engineer's true intentions. A discriminating example either satisfies the model but not the hypothesis, or satisfies the hypothesis but not the model. It shows the difference between the model and the hypothesized alternative.

The key to producing high quality discriminating examples is to generate high quality hypotheses. This dissertation explores three general forms of such hypotheses: mistakes that happen near borders; the expressed model is stronger than the engineer intends; or the expressed model is weaker than the engineer intends. We additionally propose a number of heuristics to guide the hypothesis-generation process.

We demonstrate the usefulness of discriminating examples and our hypothesis-generation techniques through a case study of an Alloy model of Dijkstra's Dining Philosophers problem. This model was written by Alloy experts and shipped with the Alloy Analyzer for several years. Previous researchers discovered the existence of a bug, but there has been no prior published account explaining how to fix it, nor has any prior tool been shown effective for assisting an engineer with this task.

Generating high-quality discriminating examples and their underlying hypotheses is computationally demanding. This dissertation shows how to make it feasible.

Acknowledgements

I would like to express my tremendous gratitude to my advisor, Professor Derek Rayside, for all his support, patience, and understanding. It has been an honor to be his first PhD student. I appreciate his immense knowledge in theoretical and practical aspects of this research and his assistance in writing reports including this dissertation. Without his guidance and direction, I cannot imagine a path to the completion of this dissertation.

I am grateful to the other members of my committee, Prof. Marsha Chechik, Prof. Krzysztof Czarnecki, Prof. Vijay Ganesh, Prof. Reid Holmes, and Prof. Richard Trefler. Their insightful questions and suggestions significantly helped me shape this research. I would also thank Prof. Paul Ward and Prof. Patrick Lam for their constructive comments on the initial draft of this dissertation.

I have been very fortunate to meet and work with brilliant people who helped me develop my skills further. In particular, I am grateful to Prof. Vijay Ganesh for all advice on precise and structural thinking, to Douglas Harder for the enjoyable opportunities in teaching, and to Mary Janet McPherson for the comprehensive training on concise writing. Special thanks to the incredible staff of the Electrical and Computer Engineering for their tremendous support and assistantship.

I would like to thank all Waterloo Formal Methods (WatForm) Lab members. Special thank to my friends Amirhossein, Pourya, Salman, Shahin, and Steven who were always so helpful in numerous ways. Amirhossein always found the time to discuss my ideas and provide invaluable feedback. I am also grateful to undergrad students, Fikayo, Bhargava, and Akshay to assist me in developing the framework and running case-studies. I thank Edward Zulkoski for his suggestions on the conference papers and presenting our paper in Toulouse.

My deep appreciation goes to my parents, Maryam and Zabihollah, and sister, Valeh, for their unparalleled encouragement to follow my dream. My sincere gratitude to Anahita for her love and understanding throughout this hard time. I am very thrilled to have such a supportive family that I can rely on at every step of my life.

Dedication

To my family with love.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The need to debug arises because the *expressed* meaning model from the *intended* meaning, but the user does not know where or why [54, 57]. Debugging can be a cumbersome and time-consuming task that persists throughout the software lifecycle [53, 109, 116]. While these truths have been mostly studied in the context of imperative programs, they are also believed to be true of declarative models [48, 55, 94].

Zeller [117], in his seminal book on debugging imperative programs, evokes an inspiring image: *Some people are true debugging gurus. They look at the code and point their finger at the screen and tell you: "Did you try X?" You try X and voila!, the failure is gone.* What has the debugging guru done? They have *identified*, *localized*, and *corrected* the bug [54, 57], and they have done this by first forming a hypothesis [117].

We have developed tools and techniques to provide some automated support for this vision in the context of relational logic models. The tools first help the user identify (and understand) the bug by forming a hypothesis about what might be wrong with the model and computing a *discriminating example* for the user to accept or reject. If the user judges that a bug has been identified, then further automated analysis helps localize which part of the model needs to change, and might provide a high-level conceptual description of the correction (but the user still needs to make the correction by hand).

This work builds on three premises that have been well-established in related areas of software engineering and computer science: examples help people understand abstractions [4, 42, 64]; near-miss examples help understand the borders of abstractions [34, 110]; and debugging is a hypothesis-driven activity [54, 57, 117].

1.1 First-Order Relational Logic Models in Alloy

Modeling software systems using languages with mathematically defined syntax and semantics assists the engineer to understand the design and find formidable issues in advance [18]. There are research efforts to develop tools for analyzing and debugging formal models such as specifications in temporal logic [8, 37, 55]. Alloy as a declarative language has been broadly used for modeling software systems in many applications [104]. In AT&T Laboratories, Zave selected Alloy, over other tools [114], to analyze distributed feature interactions [112, 113]. Newcombe et al. [80] demonstrated using Alloy and TLA [58] to analyze Amazon Web Services.

The word *model* is used with a variety of meanings in engineering, computer science, and logic. The two most common meanings in software engineering are contradictory: for much of the software engineering community, *model* means an *abstraction* (*e.g.*, a formula). For logicians, and the software engineers that use their terminology (*e.g.*, Nelson et al. [77]), it means an *example*. Bak et al. [4] recently proposed that software engineers should use *model* to mean an abstraction with accompanying examples. We use *model* to refer to the abstraction that the engineer has written: the specification; the formula. This is consistent with the usage in Jackson [42], and in much of the software engineering literature.

The Alloy language [42] is a first-order logic with relations and transitive closure. It has an associated tool [43], Alloy Analyzer for bounded analysis. Sacrificing the complete analysis, Alloy Analyzer is a tool for realizing light-weight formal methods [44]. The tool is used both to simulate and to check the consistency of logical formulas (models) written in the Alloy language. Alloy has been successfully used for a variety of software engineering tasks, such as specifying rich linked data structures and finding bugs in published protocols [42, 104].

The main idea of the Alloy Analyzer is to finitize the model to be solved: that is, to specify a finite universe over which the formula is to be evaluated. The first-order formula can then be translated to a propositional formula [102] and solved with an off-the-shelf SAT-solver, such as MiniSAT [28]. For example, the first-order formula $\forall x|p(x)$ and the finite universe $x = x_1+x_2+x_3$ would result in the propositional formula $p(x_1)\wedge p(x_2)\wedge p(x_3)$.

In Alloy language, a relation is a set of tuples of atoms. A relation has a finite arity. A set of relations is declared to be a *signature*, which is conceptually similar to a class in an object-oriented programming language. Likewise, a signature can extend other signatures or be abstract (no instances). An Alloy expression is a set of relations combined with some operators. Each Alloy expression shows a set of instances within the finite scope. The supported set operators are '+' for union, '-' for difference, and '&' for intersection.

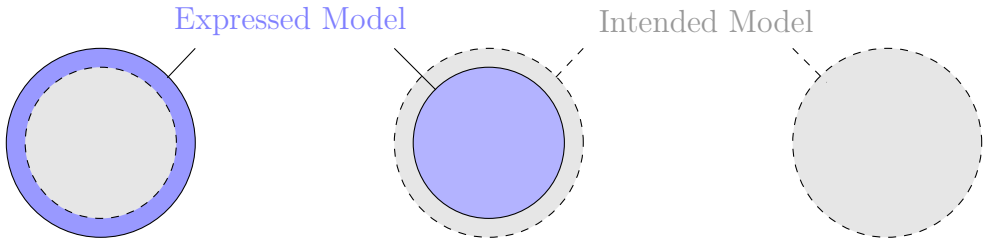
The relational operators are either unary, for instance '$\wedge$' for transitive closure and '$\sim$' for transpose, or binary, such as '.' for join and '$\rightarrow$' for product. An Alloy formula is a quantified formula over the relational expressions. A quantifier can be either 'all', 'some', 'no', and 'one'. The body of a quantified formula can be made from logical operators like '&&' for conjunction, '||' for disjunction, and '!' for negation. The relational comparison operators are also 'in' for subset and '=' for equality. An Alloy model is comprised of Alloy formulas and relations. An instance of an Alloy model is a valuation of the relations satisfying the formulas. Values that any signature can take are called atoms. A universe is a finite set of atoms and is restricted by a specific scope. The scope for each signature is given after the executing commands. An instance is a valuation of a formula within the given scope.

Alloy users can analyze a model by simulating it in a finite scope. They can validate their desired assertions over the simulated model by running a search for an instance that refutes the assertion. If the analyzer finds such an instance, the assertion is definitely rejected, and the instance witnesses as a counter-example. No found instance may seem to be a good clue for showing the assertion correctness. However, another reason might be the insufficient scope or lack of the consistency between what the user actually wants to do and what the model really means.

Definition 1 ***(Model).*** We define an Alloy model as triple $\langle R, C, B\rangle$ comprising an ordered set of relations R, a set of constraints (formulas) on those relations C, and finite bounds B for those relations.

Similar to logical modeling, logic programming is a set of logical statements to express desired results. To execute a logic program, Prolog, as the best known logical programming language, restricts the expressiveness of first-order logic. The execution of a logical program is based on theorem proving by first order resolution, whereas analysis of Alloy models is a search of a valid interpretation of a SAT formula. A Prolog program is a set of Horn clauses, but any first-order statements can be expressed in Alloy language. That is, the expressiveness of Alloy language subsumes Prolog. Near [73] proposed a technique for translating Alloy and Imperative Alloy [74] formulas, without universal quantification and negation, to Prolog statements. This allows users to produce a Prolog program, from some Alloy statements, that processes ten times larger inputs in a shorter time than Alloy Analyzer [72].

Figure 1.1 Overconstraint and underconstraint bugs. The darker area with solid border represent the expressed model, and the brighter area with dotted border show what the user actually intend.



(a) Underconstraint **(b)** Partial Overconstraint **(c)** Total Overconstraint

1.2 A Taxonomy of Bugs in Relational First-order Models

Relational first-order models, like any other programming artifacts, might not be perfect expressions of design intent.

Definition 2 ***(Expressed model).*** The model that is expressed by the user, M_E.

Definition 3 ***(Intended model).*** The model that is intended by the user, M_I. This is purely conceptual: it is in the user's mind; it is not written.

In general, bugs are inconsistencies between an intended model and an expressed model.

Definition 4 ***(Bug).*** Divergence between the user's expressed and intended models: $M_E \not\Leftrightarrow M_I$.

We consider that there are four types of bugs that might occur in this context: *underconstraint*, *partial overconstraint*, *total overconstraint*, and *insufficient scope*. This dissertation describes techniques to assist with debugging the first three categories. It also describes some techniques for computing sufficient scopes in some circumstances, which can help the user avoid insufficient scope problems, but does not provide explicit debugging support for debugging insufficient scope problems.

Definition 5 ***(Underconstraint).*** What the user intends is more restricted than what she has expressed: $M_I \Rightarrow M_E \wedge M_E \not\Rightarrow M_I$.

An *underconstrained* (model) formula allows unintended examples. For instance, a model of a linked list might permit cyclic lists when the engineer intended all lists to be acyclic. Figure 1.1a depicts a Venn diagram where the outer circle represents all of the

examples permitted by the model, and the inner circle represents the examples intended by the engineer.

Definition 6 ***(Partial overconstraint).*** What the user expressed is more restricted than her intention: $M_E \Rightarrow M_I \wedge M_I \not\Rightarrow M_E$.

A *partially overconstrained* model permits some of the engineer's intended examples, but not others. For instance, a model of a linked list might forbid lists of length one, but permit lists of other lengths.

Definition 7 ***(Total overconstraint).*** What the user expressed is unsatisfiable despite her intention: $M_E \models \text{UNSAT} \wedge M_I \models \text{SAT}$.

A *totally overconstrained* model permits no examples — against the engineer's intention. For instance, the engineer might check the safety property that a model of a linked list does not permit cyclic lists; the safety check passes because the model does not permit any lists of any kind, let alone cyclic ones.

Insufficient Scope. This problem is an artifact of analysis technology. Some first-order logic analyzers, such as the Alloy Analyzer, work by putting finite scopes on the quantifiers and translating to propositional logic (SAT). Assume a model of a linked list with distinguished head and tail nodes that do not hold data elements. An analysis that requires a list holding two data elements might fail if the number of nodes were capped at three (since two of these nodes would be the head and the tail, there would only be one node left to hold a data element). Insufficient scope can appear to the user like an overconstraint problem, since some intended examples are excluded, but the cause is different: in a true overconstraint situation, the first-order formula itself excludes the intended examples; in an insufficient scope situation, it is the finitization to propositional logic that causes the intended examples to be excluded. Scope size correlates with time taken to perform the analysis, so the user generally wants to use the smallest scope possible. Larger scopes might render the analysis infeasible.

A note on terminology. The term '*bug*' is widely used in software engineering practice, and also in the research literature, with a variety of meanings: a bug could be an incorrect program code, program state, or even program behaviour [117]. Most frequently these discussions are in the context of executable programs, but the term *debugging* is also used in the context of Linear Temporal Logic (LTL) specifications [55].

The research literature, especially in the area of dependable computing, has standardized meanings for *fault*, *failure*, and *error* [3]. What we mean by 'bug' here is a kind of

fault. In the eight-dimensional classification of faults described by Avizienis et al. [3], our bugs are human-made, internal, non-malicious, software, development faults; they might be due to either accident or incompetence, and they might or might not be deliberate (*i.e.*, the result of an explicit engineering trade-off). The persistence classification is not particularly relevant in the context of specifications. The term 'bug' has been widely used, with a variety of meanings, and our use of the term is consistent with much prior precedent.

1.3 Existing Debugging Tools and Techniques

Debugging can consume much of the effort invested in software development and maintenance [53, 109, 116]. Ensuring that high-level models are accurate reflections of the designer's intent is especially important, since models often form the intellectual foundation for the rest of the software development process [48].

Debugging, considered as a hypothesis-driven activity [54, 57, 117], is divided into three sub-activities: *identification*, *localization*, and *fixing*. The first activity includes detecting and understanding bugs [57]; however, Ko et al. [54] focused on improving bug understanding. Also, studies suggest that understanding is a key factor, and that some debugging tools (for imperative programs) provide inadequate support for this task [82].

One of the great features of the original Alloy Analyzer (and all subsequent versions) is its example (and counter-example) finder [42]. This tool finds arbitrary examples (or counter-examples). These arbitrary examples can be useful for understanding underconstraint bugs in the model, by illustrating that what the designer does not intend is actually permissible. Whether the particular example generated affirms or clashes with the designer's intent is a matter of luck and chance.

Seater [94] proposed non-example generation technique to disclose the role, *restraining* or *relaxing*, of a particular formula in a model. The technique aims at understanding a model and assists users to hypothetically argue why a constraint was not written differently, which does not become apparent by only running the model and arbitrary browsing the instances. As the technique uses non-examples to assist users to understand the effect of models' parts, discriminating examples focus on particular hypotheses for debugging a model. Using this technique, users can test their hypotheses about different syntactical mutations of the model, whereas the discriminating examples encode hypotheses based on prior experiences on slight semantics mutations of the model. Despite lack of implementation, the core idea of using non-examples for testing different hypotheses inspired our idea about discriminating examples.

Modeling by example technique [64] is that the system induces logical constraints through a dialogue of examples with the user. She begins by providing some prototypical instances to the system, and then the system responds with other instances that the user classifies as either valid or invalid. As the dialogue continues, the system refines a general formula that includes the positive examples and excludes the negative ones. Using examples for dialogue-based interactions has influenced our debugger, despite the fact that the discussed modeling be example has not been implemented.

Nelson et al. [78] proposed a technique to produce *minimal examples*. The motivation is that a minimal example reveals the essential nature of the model. Minimal examples might reveal if the lower bound of the model is underconstrained, but are less likely to reveal if the upper bound of the model is underconstrained. Nelson et al. [78] also proposed a facility for the user to build up more complicated examples from a minimal example called *scenario exploration*. This process might help the user discover underconstraint bugs on the upper bound of the model if the user specifically tries to construct an example that they believe should be illegal. This process only produces legal examples, and so is unlikely to help the user find partial overconstraint bugs.

This dissertation provides improved tool support for understanding underconstraint bugs by generating examples that might be amongst those permitted by the model but not intended by the user. This dissertation also provides improved tool support for understanding partial overconstraint bugs by generating examples that might be amongst those intended by the user but forbidden by the model.

With Alloy Analyzer, users can explore all examples of a given model whilst their order is arbitrary. A vast number of examples of a model makes arbitrary exploration infeasible. Aluminum provides focused examples by prioritizing absolute minimal examples as the first ones to be explored. This dissertation provides another approach for prioritizing examples that assist users to test different hypotheses for debugging.

Shlyakhter et al. [96] enhanced the Alloy Analyzer to use the *unsatisfiable core* feature of some SAT solvers for localizing (total) overconstraint bugs. The unsatisfiable core of a formula is a subset of the formula that by is internally inconsistent: the overconstraint is within this subset. Torlak et al. [103] subsequently enhanced this tool with improved minimization techniques that provide better localization. Total overconstraint bugs have a good localization tool in this, but limited understanding tools. This dissertation provides improved understanding and localization tools to deal with total overconstraint bugs by generating non-examples that might be amongst those prohibited by the model but intended by the user.

Partial overconstraint bugs currently have neither localization nor understanding tool

support. A partially overconstrained model permits some intended examples but forbids others. This dissertation provides techniques to generate non-examples that might be amongst those that are prohibited by the model but intended by the user.

Alloy language provides explicit syntax for specifying finite scopes for analyzing the model; the Alloy Analyzer however provides no automatic support for determining sufficient scope to exclude spurious examples. Nelson et al. [77] proposed a technique to determine scopes for Bernays-Schonfinkel-Ramsey fragment of first-order logic. Although, solving the issues with insufficient scopes is not the main goal of our research, we have developed techniques that help users to select proper scopes for particular queries in the form of $\exists\forall$.

Existing tool support for fix suggestion in relational logic models is quite limited. This dissertation provides some tool support: in some circumstances one of the proposed techniques can offer the user some high-level advice of what needs to change. For example, it might say something like 'relax the total order on relation R to be a partial order.'

Debugging involves three sub-tasks: localization, understanding, and fixing [54]. This dissertation provides tools and techniques that advance the state of the art for all three, with a particular emphasis on understanding. The techniques described herein generate examples that are useful for understanding the three main classes of bugs in logic models: underconstraint, total overconstraint, and partial overconstraint.

1.4 Discriminating Examples and Other Kinds

Declarative descriptions are easy to develop and read while being used for specifying a general concept [64]. A model is imprecise unless all the corner cases are concretely covered. Specifying and understanding the exceptional cases become easier once specific examples are used. Examples are precise and straightforward to understand. They also can be provided graphically as opposed to in textual representation of a declarative formula. Examples, as the core of our approach, play an important role in assisting the user to understanding a bug and identify the root of the problem. Concrete examples, unlike declarative models are much less abstract and more understandable [4]. In the other words, using examples, the user can concretely convey what she expects or does not. Hence, we consider the given examples as an always correct or incorrect piece of information. Note that various types of examples of a given model are different valuations of the same relations in the model.

Example In relational first-order models, an example is one interpretation that satisfies the model's constraints [Definition 10]. The examples can be specified by users or produced

mechanically.

Counter-examples Examples that refute a claim are counter-examples [42]. In the context of relational first-order models, counter-examples reveal underconstraint bugs.

Non-example An interpretation of the model that does not satisfy the model's constraints is a non-example [Definition 11].

Arbitrary example An example that is mechanically determined without considering any constraints other than the original model is an arbitrary example [42].

Minimal example An example that becomes non-example if any of its tuples is removed [78].

Near-hit and near-miss examples We define distance between two (non-)examples, as the number of changes that makes two (non-)examples identical [Definition 14 and Definition 15]. Near-hit and near-miss examples are a pair of example and non-examples with a minimal distance, respectively. Borders conceptually separate examples and non-examples of a given model. Intuitively, a border is between an example and a non-example with a minimum distance. Therefore, a pair of near-hit and near-miss examples can identify the concept of border. From this perspective, near-hit and near-miss examples can also be called *near-border* examples.

Discriminating example We define them as examples or non-examples that explore particular hypotheses about how the model should be modified to express the user's intent more clearly [Definition 17]. Different types of examples can encode such hypotheses. Using near-hit and near-miss examples as discriminating examples can evaluate the hypothesis that bugs happen around the borders. Discriminating examples as non-examples or examples produced from a weaker or stronger mutation of a model can be used to assess the hypothesis that the model has an overconstraint or underconstraint bug, respectively.

1.5 Thesis and Contributions

Examples being concrete and thus more understandable than higher level abstractions [4], are widely used for exploring different aspects of models [104]. Showing the possibility of using non-examples in finding overconstraints is the key to this project. Non-examples have been used in similar domains to assist learning procedures [110]. Our debugger asks the user a series of focused questions in the form of examples and non-examples that explore

particular hypotheses about how the model should be modified to more clearly express the user's intent. This dissertation is about exploring new approaches to demonstrate:

Discriminating examples are feasible and useful for first-order relational logic languages.

A variety of techniques are employed to make the computation of discriminating examples feasible, including: partial instances, staged evaluation, counter-example guided inductive synthesis, pre-computed analyses of modeling patterns, search heuristics, and parallelization. The utility of discriminating examples for debugging models is demonstrated with illustrative case studies. In comparison to the existing tools, we show the usefulness of discriminating examples in debugging the Dijkstra's dining philosophers problem. All in all, this dissertation describes our proposed techniques and tools that make the following contributions:

Contribution 1 (Discriminating Examples): exploring the idea of using (non-)examples that are focused on confirming or rejecting a particular hypothesis about the model;

Contribution 2 (Formal Definition of Near-miss and Near-hit examples): higher-order quantifiers are used to define Near-miss and Near-hit examples in a way that can be feasibly computed with modern solving techniques, including techniques described in this dissertation;

Contribution 3 (Relational Pattern-based Mutation): an approach for producing discriminating examples by weakening or strengthening the model;

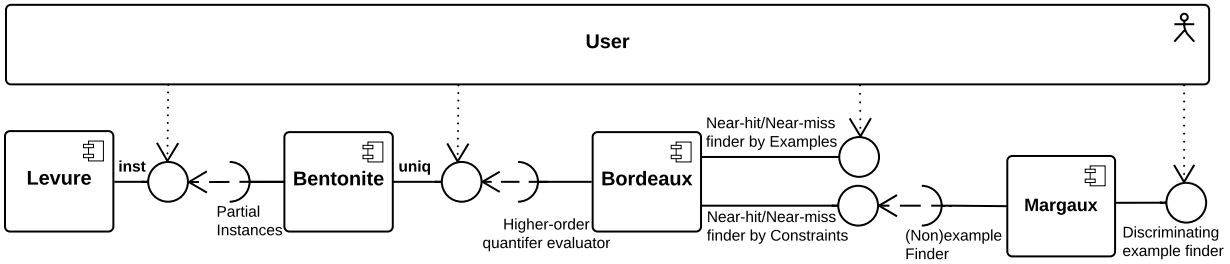
Contribution 4 (Temporal Patterns for Ternary Relations): a library of debug patterns encoding temporal structural changes of binary relations;

Contribution 5 (Expressing Examples): introducing **inst**, a new syntax and semantics for integrating partial instances at the level of Alloy language and analyzer;

Contribution 6 (Expressing Generator Axioms): introducing **uniq**, a new syntax for synthesizing generator axioms and mechanizing scope selection to run $\exists\forall$ queries;

Contribution 7 (Optimization): demonstrating the computational feasibility of the proposed approaches through case studies, using appropriate techniques.

Figure 1.2 Interacting components for producing discriminating examples



1.5.1 Tools Developed

We have developed four major components[1] (Figure 1.2), in order to realize and evaluate the concept of discriminating examples. BORDEAUX is a component for producing examples and non-examples close to borders. By integrating this component with Alloy Analyzer, users can browse non-examples as well as examples. We have developed MARGAUX to assist users in debugging relational models through discriminating examples. Such discriminating examples check whether bugs happen near the borders or particular semantics are misused. MARGAUX interacts with BORDEAUX to check the first hypothesis, *i.e.*, mistakes happen near the borders. For the second hypothesis, the debugger analyzes the model and finds relevant debug patterns for generating mutations that produce particular discriminating examples. Pattern-based inference and mutation are our contributions for producing discriminating examples. The inference uses the concept of debug patterns to approximate the model and its constituent constraints. Such patterns are conceptually similar to software engineering patterns encoding best practices. We have demonstrated the possibility of this approach in revealing and fixing very subtle bugs [68].

Developing BORDEAUX needed a tool to solve $\exists\forall$ queries. BENTONITE [67] is our contribution for running such $\exists\forall$ queries. In addition, BENTONITE has been used for analyzing models that need the exact scope for composite structures to prevent spurious examples or counter-examples [24]. To achieve better performance, BENTONITE exhaustively uses LEVURE [66] which is our contribution that encodes and simulates the concept of the partial instances in Alloy language and analyzer. LEVURE has also been used in other research projects to gain better performance [49, 81].

The foremost future work of this dissertation is to empirically measure the usability of the proposed techniques in collaboration with a real programmer. In addition to improving

[1]To assess the idea, we have developed 2,615, 3,114, 5,497, and 25,580 lines of Java code (measured by CLOC [21]) for LEVURE, BENTONITE, BORDEAUX, and MARGAUX, respectively.

the performance of the techniques and tools, employing more libraries of debug patterns will enrich the inference technique. A new library of debug patterns can be categorized by experts, in the form of best practices or by a machinery technique based on frequencies.

1.5.2 Dissertation Organization

Levure, Bentonite, Bordeaux, and Margaux are the different tools collaborating with each other to facilitate the debugging of formal models. Through the chapters of this dissertation, we explore the ideas behind each tool, propose techniques to realize each, and assess their efficiency.

Chapter 2 explains Levure for encoding partial instances in Alloy and translating them into the underlying solver's language. The chapter also shows how partial instance blocks can be used for testing Alloy models.

Chapter 3 describes the computational cost of Bentonite in running $\exists\forall$ queries. In addition to our technique to find scopes and stage the simulation, the chapter includes comparisons between Bentonite and its successors.

Chapter 4 demonstrates the concept of non-examples for debugging models. Moreover, this chapter shows our definition for relative distances and its applications in producing near-hit and near-miss examples with Bordeaux.

Chapter 5 describes how to compute discriminating examples by performing inference with a library of debug patterns, followed by mutation. Additionally, the heuristics and optimizations that make this computation *feasible* are described and measured. These ideas are embodied in the Margaux tool.

Chapter 6 demonstrates that discriminating examples are more *useful* for debugging than existing tools and techniques are in a few case studies, including the Dijkstra's dining philosophers problem.

Chapter 7 concludes that *discriminating examples are feasible and useful for first-order relational logic languages.* Directions for future work are also discussed.

1.5.3 Research Questions

This dissertation provides empirical answers to the following research questions.

Chapter 2

LEVURE: A Syntactic Extension of Alloy with Examples

One of the great features of the Alloy Analyzer tool is computing and visualizing examples (or counter-examples) to help illustrate the meaning of the model to the user. However, the Alloy language has no explicit syntactic support for the user to write these examples down in a way that can be used in the future debugging, testing, and development of the model. Perhaps surprisingly, whereas the Alloy language has no way for the user to write examples, the backend Kodkod API [102] not only has support for encoding examples (*partial instances*), it also has features to perform computation with them more efficiently. The Alloy Analyzer makes some of this improved computational efficiency available to the user implicitly via static analyses of the model. Our syntactic extension, listed as **Contribution 6**, makes this backend functionality explicitly available to the user, as a bonus on top of the debugging and development benefits of explicitly encoding examples with the model [4].

Figure 2.1 introduces our syntax extension by describing three instances of a linked list: simple, single, and cyclic. In the simple instance, the line Node = head + middle + tail says that there are exactly three node atoms and their names are head, middle, and tail. The next two lines give the exact bounds for the next and val relations in terms of these atoms and the integers. The single and cyclic instances are defined in a similar manner. Each **inst** block is a corner-case of the linked-list model. While developing the model, one can reuse them to ensure that the model aligns with one's intentions.

The **inst** block gives the Alloy user direct access to Kodkod's partial instance feature. Previously, any modeller wishing to specify an instance had to do it implicitly either by a

constraint or by a constant function. Consider the phrase val = n→0 which gives an exact bound for the val relation in the single instance of Figure 2.1. In Alloy 4, the modeller could have achieved a similar semantics result with the constraint **fact** {val = n→0} or by commenting out the val relation declaration and introducing a constant function of the same name: **fun** val[] : Node→**Int** {n→0}. As described below, our new syntax extension affords the modeller greater clarity and modularity, and corresponds to more consistently efficient translation.

Figure 2.1 Alloy model of a linked list with instances expressed in proposed syntax

```
1 sig Node { next : lone Node, val : one Int }
2 inst simple { Node = head + middle + tail, -- introduce three atoms
3               next = head→middle + middle→tail, -- exact bound for next relation
4               val = head→0 + middle→1 + tail→2 } -- exact bound for val relation
5 inst single { Node = n, no next, val = n→0 }
6 inst cyclic { Node = a + b, next = a→b + b→a, val = a→0 + b→1}
```

Section 2.1 describes four ways in which partial instances benefit the Alloy user: test-driven development, regression testing, example-driven modeling, and combined modeling and meta-modeling. Section 2.2 describes Levure, our proposed extension to Alloy. Section 2.3 presents experiments that demonstrate the increased computational efficiency achieved by directly exposing Kodkod's partial instance feature, compared to that achieved by adopting traditional Alloy syntax. Section 2.4 considers two other possible ways to make Kodkod's partial instance feature available to Alloy users, and argues that our main proposal is preferable. Section 2.6 summarizes the chapter.

2.1 Using Partial Instances for Alloy models

By explicitly expressing examples using partial instance without sacrificing performance, one can apply common program testing practices in Alloy models. In this section, we explore two major testing practices that utilize adding partial instance to the Alloy language and its Analyzer. We also demonstrate the utility of adding partial instances to the Alloy surface syntax in example-driven modeling and combined modeling and meta-modeling. As we show in the next sections, these practices are still applicable, but the performance significantly degrades without partial instances.

Test-Driven Development Partial instances enable modellers to apply test-driven development [7] methodology to their Alloy models. Consider the following example scenario.

Figure 2.2 A skeletal Alloy model of a binary tree

```
1 sig Node { left, right : lone Node, val : one Int}
2 pred wellFormedTree[] { } -- to be filled in by students
3 run wellFormedTree for 3
```

When we teach Alloy to senior undergraduates, the first in-class exercise is to write invariants for a binary tree. The lecturer, who has a computer running Alloy, displays the skeletal Alloy model listed in Figure 2.2.

The lecturer runs the simulation, the class looks at the result and tells the lecturer in plain language what is wrong with the displayed instance, and then the lecturer translates that plain language into formal constraints within the wellFormedTree predicate.

During this initial exercise, it is common for students to identify an instance of the model where some node y is both the left and right child of some node x. When this occurs, the students usually give a constraint such as 'the left and right children cannot be equal,' which the lecturer translates as **all** n : Node | n.left != n.right. The students tend to be satisfied with this translation, but the astute reader will notice that this formalization prevents leaf nodes, forcing the tree to be cyclic (*i.e.*, a leaf node has no left child and no right child, and clearly the empty set is equal to the empty set). The students typically do not realize this overconstraint for fifteen or twenty minutes.

Had the students been following test-driven development with partial instances, they may have realized the folly of the proposed formalization sooner. Suppose that the students had first written the two simple partial instances in Figure 2.3. Figure 2.3a lists a tree of a single node that the students expect to be legal. Figure 2.3b lists a tree with self-loops that the students expect to be illegal. When the wellFormedTree predicate is empty at the beginning of the lecture, the illegal self-loops test fails. When the bogus constraint n.left != n.right is added, then the singleton tree test fails. Having concrete tests (partial instances) to detect errors in the program (model) is the essence of test-driven development.

A difference between test-driven development for imperative code versus that for declarative logic models is the role of positive and negative examples. With imperative code, the programmer writes positive test cases for empty procedures (or code stubs) that initially fail. With declarative logic models, a positive example (such as a singleton tree) will succeed with an empty wellFormedTree predicate. Only once the predicate becomes overconstrained will the positive example fail. In contrast, negative examples will fail with the empty predicate, and will only pass with a properly constrained predicate. Consider, for example, the negative example of a node that is its own child in Figure 2.3b. If wellFormedTree is underconstrained (*e.g.*, empty) then this test will fail. Thus, the pro-

Figure 2.3 Two partial instances of a binary tree: (a) a legal singleton tree, and (b) a tree with illegal self-loops

(a)

```
1 inst SingletonTree { Node = n, no left, no right, val = n→0 }
2 run wellFormedTree for SingletonTree expect 1
```

(b)

```
1 inst IllegalSelfLoops { Node = n, left = n→n, right = n→n }
2 run wellFormedTree for IllegalSelfLoops expect 0
```

grammer builds up a procedure for constructing positive examples, and the modeller builds up a predicate to rule out negative examples.

Regression Testing of Alloy Models Like programs, models evolve: requirements change, extra properties need to be checked, refactoring is needed for readability, and so on. As with programs, some form of regression testing can provide assurance that the model (or program) still corresponds to programmer intent.

For an Alloy model with associated safety properties, partial instances can be used in regression testing to detect over-constrained models. When a model becomes over-constrained, the safety properties will still hold; however, the modeller might be unaware of over-constraints. Regression testing of Alloy instances can be effective in detecting these occurrences. The users following a TDD approach can have their initial tests do double duty as regression tests.

Example-Driven Modeling The idea of Example-Driven Modeling (EDM) [4] is to utilize explicit examples as first-class citizens for comprehending, verifying, and validating software modeling. With an extensive use of examples, EDM makes modeling accessible for stakeholders who have limited training in software modeling, yet know the problem domain very well. EDM aims to improve the quality of software modeling be synthesizing abstraction from examples and deriving examples from abstract models. As discussed by Bak et al. [4], explicit syntactic support for examples is an important part of an EDM system. With our proposed syntax, EDM can benefit from partial examples modeling that only aims for important properties in a given model.

Combined Modeling and Meta-Modeling Alloy is sometimes used to define new modeling languages. We will refer to such activity as 'meta-modeling.' Let L name the

Alloy model that describes the new language, and let M name an Alloy model that describes a model written in the new language. At present, there is often no mechanical connection between L and M. Our facility for adding partial instances to Alloy makes it easier to have L and M tightly integrated. We examine the work of Cai and Sullivan [12, 13, 14, 99] as a case study to illustrate these points and demonstrate the performance improvement achieved by **inst** block.

2.2 Language Extension

We propose to add an **inst** block to the Alloy language, *i.e.*, **Contribution 5**, allowing the user to specify a partial instance, as illustrated above in Figures 2.1, 2.2, and 2.3. The partial instances in those examples only use exact bounds; Kodkod and our syntax support lower and upper bounds as well, using the **in** and includes keywords, respectively. The lower bound is a set of tuples that a relation must have, and the upper bound is the one that relation might have [101].

These **inst** blocks are given names and used in Alloy commands. Whereas once a user might write **run** p **for** 3, they will now write **run** p **for** i, indicating that predicate p is to be simulated in the context of partial instance i.

An **inst** block, like a **sig** block, may have an appended fact. For **inst** blocks, the appended fact is only expected to be true when that **inst** block is part of the command being executed. The purpose of this appended fact is to give the specifier an opportunity to write constraints that mention the atom names introduced in the **inst** block — these names are not available elsewhere in the model.

Figure 2.4a lists the grammar for our proposed extension to the Alloy language to support partial instances. An iBk has a name, a list of iSts and optionally an appended fact. Each iSt alternative that contains a var bounds either a signature or a field (whichever is named by the var). The one iSt alternative that does not name a var provides the default number of atoms for each signature. A relation (signature or field) name can only appear on the left-hand side of at most one iSt in each iBk.

An iSt that names a signature on its left-hand side introduces atom names on its right-hand side. These atom names can then be used to describe the bounds on fields. An iXpr is an expression that describes a set of tuples using the normal Alloy union (+) and cross-product ($\rightarrow$) operators along with the names of the atoms. If the user wishes to specify both an upper and lower bound for relation r, she can write an iSt such as

Figure 2.4 Grammar and preliminary type definitions

(a) Grammar

⟨*iBlk*⟩ := 'inst' id ('extends' id)? '{' ⟨*iSt*⟩[,⟨*iSt*⟩]* '}' ('{' ⟨*frml*⟩ '}')?

⟨*iSt*⟩ := ⟨*n*⟩
| 'exactly' ⟨*n*⟩ ⟨*var*⟩
| ⟨*var*⟩ '=' ⟨*iXpr*⟩
| ⟨*var*⟩ 'in' ⟨*iXpr*⟩
| ⟨*var*⟩ 'include' ⟨*iXpr*⟩
| ⟨*var*⟩ 'include' ⟨*iXpr*⟩ 'moreover' ⟨*iXpr*⟩
| 'no' ⟨*var*⟩

⟨*iXpr*⟩ := ⟨*iXpr*⟩ '->' ⟨*iXpr*⟩
| ⟨*iXpr*⟩ '+' ⟨*iXpr*⟩
| '(' ⟨*iXpr*⟩ ')'
| ⟨*atm*⟩

(b) Preliminary type definitions

⟨*prb*⟩ := ⟨*univ*⟩ ⟨*iSt*⟩* ⟨*frml*⟩*

⟨*univ*⟩ := {⟨*atm*⟩[,⟨*atm*⟩]*}

⟨*tpl*⟩ := ⟨*atm*[,*atm*]*⟩

⟨*cnst*⟩ := {tpl[,tpl]*} | {} [×{}]*

⟨*var*⟩ := id

⟨*atm*⟩ := id

⟨*sig*⟩ := ⟨*var*⟩

⟨*sigs*⟩ := ⟨*sig*⟩*

⟨*n*⟩ := int

r **include** x + y **moreover** p + q, specifying a lower bound of x + y and an upper bound of x + y + p + q.

One partial instance block may extend another. The semantics of partial instance extension are simply concatenation and conjunction. Let p name the base partial instance block; let q name the extending partial instance block; and let r name the result of applying the extension to q. The text of r is the concatenation of the text of p with the text of q. The appended fact of r is the conjunction of p's appended fact with q's appended fact. The result, r, must follow the same well-formedness guidelines as p and q: no relation can be named on the left-hand side of more than one statement. This restriction keeps both regular semantics and extension semantics simple, as it prevents statements from interfering with each other (notwithstanding quantitative statements that interact with named statements in a well-defined manner as formalized below).

2.2.1 Semantics

We define the semantics of the partial instance block as an extension of the Kodkod semantics [101]. The Kodkod semantics take a universe and relation bounds as inputs. The purpose of the partial instance block is for the user to specify the universe and relation bounds.

Figure 2.5 describes how the universe is constructed from a partial instance block by the U function, which in turn makes use of the N, X, and G functions. Preliminary type definitions are given above in Figure 2.4b. First the N function constructs a universe in

Figure 2.5 Universe construction

$evr : sig \to univ$
$U : iBlk \to sigs \to evr$
$G : iSt^* \to sigs \to evr \to univ$ $\quad G' : iSt \to sigs \to evr$
$X : iSt^* \to sigs \to evr \to evr$ $\quad X' : iSt \to sigs \to evr \to evr$
$N : iSt^* \to sigs \to evr$ $\quad N' : iSt \to sigs \to evr$
$K : sig \to int \to univ$
$Q : iXpr \to univ$

$$
\begin{array}{lcl}
U[\![iBlk, sigs]\!] & := & G[\![iSt_1 \dots iSt_n, sigs, X[\![iSt^*, sigs, N[\![iSt^*, sigs, \emptyset]\!]]\!]]\!] \\
G[\![iSt^*, sig, evr]\!] & := & G[\![iSt_1 \dots iSt_n, sigs, evr]\!] \\
G[\![iSt_1 \dots iSt_n, sigs, evr]\!] & := & G[\![iSt_2 \dots iSt_n, sigs, evr \ +\!+ \ G'[\![iSt_1, sigs]\!]]\!] \\
G[\![[], sigs, evr]\!] & := & evr \\
G'[\![v \ [=|\textbf{in}|\textbf{include}] \ p, sigs]\!] & := & \{(a,b) | a \in sigs \wedge a = v \wedge b \in Q[\![p]\!]\} \\
G'[\![v \ \textbf{include} \ p \ \textbf{moreover} \ q, sigs]\!] & := & \{(a,b) | a \in sigs \wedge a = v \wedge b \in Q[\![p]\!] \cup Q[\![q]\!]\} \\
X[\![iSt^*, sigs, evr]\!] & := & X[\![iSt_1 \dots iSt_n, sigs, evr]\!] \\
X[\![iSt_1 \dots iSt_n, sigs, evr]\!] & := & X[\![iSt_2 \dots iSt_n, sigs, evr \ +\!+ \ X'[\![iSt_1, sigs]\!]]\!] \\
X[\![[], sigs, evr]\!] & := & evr \\
X'[\![\textbf{exactly} \ n \ v, sigs]\!] & := & \{(a,b) | a \in sigs \wedge \ a = v \wedge b \in K[\![v,n]\!]\} \\
N[\![iSt^*, sigs, evr]\!] & := & N[\![iSt_1 \dots iSt_n, sigs, evr]\!] \\
N[\![iSt_1 \dots iSt_n, sigs, evr]\!] & := & N[\![iSt_2 \dots iSt_n, sigs, evr \ +\!+ \ N'[\![iSt_1, sigs]\!]]\!] \\
N[\![[], sigs, evr]\!] & := & evr \\
N'[\![n, sigs]\!] & := & \{(a,b) | a \in sigs \wedge b \in K[\![a,n]\!]\} \\
K[\![v,n]\!] & := & \{\langle ToString(v) +' \$' + ToString(n-1) \rangle\} \cup K[\![v, n-1]\!] \\
K[\![v,0]\!] & := & \langle\rangle \\
Q[\![p]\!] & := & \{\langle ToString(p) \rangle\} \\
Q[\![p+q]\!] & := & Q[\![p]\!] \cup Q[\![q]\!] \\
Q[\![p \to q]\!] & := & \langle\rangle
\end{array}
$$

which each sig has the default number of atoms. The X function takes this default universe and returns a universe that complies with the **exactly** statements in the partial instance block. Finally, the G function adds atoms named in upper and lower bound statements. All of these functions take as input a set of the sigs declared in the model. This set of **sig** names is used to distinguish statements that might introduce atoms (which name a **sig** on the left-hand side) from statements that bound relations (which name a field on the left-hand side).

Once the universe is constructed (Figure 2.5), then the bounds can be constructed (Figure 2.6). Figure 2.6 starts by redefining the top-level function P from the Kodkod semantics [101] to indicate that the universe and the relation bounds are generated from the partial instance block.

2.3 Experiment

Having a prototype of LEVURE, we perform an experiment to answer:

RQ 2.1. *What performance improvement is achieved using **inst**?*

Figure 2.6 Bounds construction (building on formalization of [101])

$P : problem \to binding \to boolean$	—	top-level function, re-defined from [101]
$F : formula \to binding \to boolean$	—	formulas, definition given in [101]
$S : iSt^* \to sigs \to evr \to binding \to boolean$	—	list of inst statements
$S' : iSt \to sigs \to evr \to binding \to boolean$	—	individual inst statement
$C : iXpr \to univ \to cnst$	—	expressions
$W : var \to sigs \to evr \to univ$	—	
$P[\![sigs.U[\![iBk, sigs]\!]\ iSt_1 \ldots iSt_n\ frml^*]\!]_b$	:=	$S[\![iSt_1 \ldots iSt_n, sigs, U[\![iBk, sigs]\!]]\!]_b \wedge F[\![frml^*]\!]_b$
$S[\![iSt_1 \ldots iSt_n, sigs, evr]\!]_b$	:=	$S[\![iSt_2 \ldots iSt_n, evr, sigs]\!]_b \wedge S'[\![iSt_1, evr, sigs]\!]_b$
$S[\![[], evr, sigs]\!]_b$	:=	$true$
$S'[\![\textbf{exactly}\ n\ v, evr, sigs]\!]_b$	:=	$W[\![v, sigs, evr]\!] \subseteq b(v) \subseteq W[\![v, sigs, evr]\!]$
$S'[\![v{=}p, evr, sigs]\!]_b$	:=	$C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p, sigs.evr]\!]$
$S'[\![v\ \textbf{in}\ p, evr, sigs]\!]_b$	:=	$C[\![\emptyset, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p, sigs.evr]\!]$
$S'[\![v\ \textbf{include}\ p, evr, sigs]\!]_b$	:=	$C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq W[\![v, sigs, evr]\!]$
$S'[\![v\ \textbf{include}\ p\ \textbf{moreover}\ q, evr, sigs]\!]_b$	:=	$C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p + q, sigs.evr]\!]$
$S'[\![\textbf{no}\ v]\!]_b$	:=	$b(v) = \emptyset$
$C[\![p + q, univ]\!]$:= $C[\![p, univ]\!] \cup C[\![q, univ]\!]$		
$C[\![p \to q, univ]\!]$:= $\{\langle p_1, \ldots, p_n, q_1, \ldots, q_m\rangle \mid \langle p_1, \ldots, p_n\rangle \in C[\![p, univ]\!] \wedge \langle q_1, \ldots, q_m\rangle \in C[\![q, univ]\!]\}$		
$C[\![p, univ]\!]$:= $\{p' \mid p' \in univ\ \wedge\ ToString(p') = p\}$		
$W[\![v, sigs, evr]\!]$:= $\{\langle p_1, \ldots, p_n\rangle \mid (v \in sigs\ \implies\ p_1 \in v.evr) \wedge (v \notin sigs\ \implies\ p_i \in v_i.evr)\}$		

To answer this question, we compared using the **inst** block to two alternative modeling styles in two different versions of Alloy 4.2. The two different styles were constraining relations with facts and using constant functions instead of relations. Constant functions are just expressions that are inlined at their point of use. They add clauses but not variables to the generated SAT formula. Alloy 4.x includes some inference capability to translate constraints on relations as bounds. In response to a draft of this extension, the Alloy development team improved this inference capability. We refer to this enhanced version as A4.2′, and to the version of Alloy 4.2 from January 2012 as A4.2. We refer to our version of Alloy with the **inst** block as LEVURE. We devised a micro-benchmark, explained below, to evaluate the computational efficiency of the proposed **inst** block compared to A4.2′ and A4.2. All tests have been done on Intel i7-2600K CPU at 3.40GHz with 16GB memory. The performance results are essentially the same with both MiniSAT and Sat4J, although we report only the Sat4J results here.

2.3.1 Micro Benchmark

We devised a micro-benchmark to illustrate the upper bound on the potential performance improvements of exposing Kodkod's partial instance features through our new syntax. Our micro-benchmark has a single signature S and a single binary relation r that maps S to S. For our **inst** block, we want to introduce some named atoms of sig S, and then define relation r to be a fully connected graph (*i.e.*, map every S atom to every other S atom).

Figure 2.7 Example models for micro-benchmark experiment

(a) By Fact	*(b)* By Constant-Function	*(c)* By Inst-Block
one sig S0,S1 **extends** S{} **fact** {r=S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f	**one sig** S0,S1 **extends** S{} **fun** r[]:S→S{S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f	**inst** b { S=S0 + S1, r=S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f **for** b

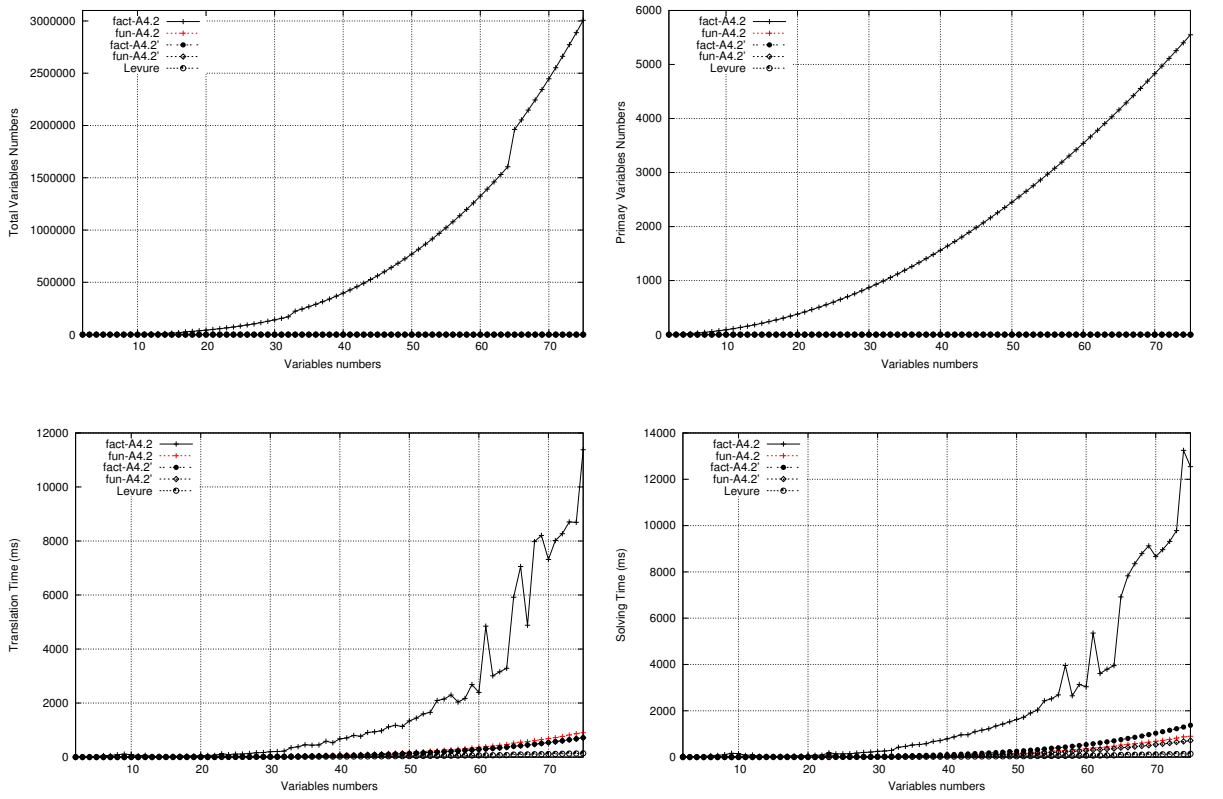
Figure 2.7 lists forms of modeling examples in the three different syntax: *(a)* constraining relation r with a **fact**; *(b)* replacing relation r with a constant function named r; and *(c)* using our new **inst** block syntax. The example listings in Figure 2.7 show these models where signature S has two atoms (S0 and S1). For the plots in Figure 2.8, we generated these models with signature S having up to seventy-five atoms. The cardinality of relation r is proportional to the square of the cardinality of signature S (as one would expect from a fully connected graph).

Figure 2.8 shows graphs characterizing how the translations of the three syntactic approaches shown in Figure 2.7 scale on different measures: *(a)* total number of variables in the resulting Boolean (SAT) formula; *(b)* number of primary variables in the resulting Boolean (SAT) formula; *(c)* time taken by Kodkod to translate the Alloy model to SAT; and *(d)* time taken by the SAT-solver to find a solution. We make a number of observations from the data in Figure 2.8:

1. The inference capability of A4.2 is incomplete: it is unable to deduce that the constraints on r can be translated as bounds rather than as variables and clauses. Therefore, the number of variables and the translation and solving times grow exponentially.
2. All other strategies show very little growth as the number of atoms increases.
3. The improved inference in A4.2′ is effective (A4.2′ Fact column).
4. The number of SAT variables produced by the constant function encoding, the improved inference, and the partial instance strategies is the same (low).
5. The partial instance encoding has the fastest translation and solving times (by a narrow margin).

The main conclusion of Figure 2.8 is that if the user chooses to use constant functions instead of relations or writes facts in a manner that Alloy can infer bounds from, then

Figure 2.8 Statements using **inst** block (*e.g.*, Figure 2.7c) take less analysis time compared to two other forms (*e.g.*, Figure 2.7a and Figure 2.7b) that are executable with Alloy Analyzer 4.2 and its enhanced version, Alloy Analyzer 4.2′.



there is little performance gain from the **inst** block. Regarding RQ 2.1, the **inst** block does provide the best performance, and does so without the users having to worry about whether their writing style is comprehensible to Alloy's bounds inference facility.

The micro-benchmark evaluates the models containing one **sig** and relation. Without **inst** block, the time grows exponentially as the number of the signature's atoms increases. As this growth happens regardless of the number of relations, the performance of the analysis is highly influenced by the number of tuples.

Figure 2.9 Partial instance encoding of Irwin *et alia*'s description [41] of the design space for a matrix manipulation program

```
1  inst IrwinMatrixDesignSpace {
2    AugmentedConstraintNetwork = ACN,
3    Variable = Density + Struct + Alg,
4    Value = dense+sparse + links+array + traverse+lookup + other,
5    domain = Density→(dense+sparse) + Struct→(links+array+other) +
6      Alg→(lookup+traverse+other),
7    dominates = ACN →((Struct→Density)+(Alg→Density)),
8    solutions = ACN →Solution,
9  }{−−Appended facts have access to atom names introduced in inst block
10   all s : Solution | {
11     let x = {p : Variable, q : Value | some b : s.bindings | p=b.var and q=b.val} |{
12       (Struct→links) in x ⇒ (Density→sparse) in x
13       (Struct→array) in x ⇒ (Density→dense) in x
14       (Alg→lookup) in x ⇒ (Struct→array) in x
15       (Alg→traverse) in x ⇒ (Struct→links) in x
16   }}
17 }
18 run createMatrixACN for IrwinMatrixDesignSpace
```

2.3.2 Staged Evaluation

In a series of papers over the last ten years Cai & Sullivan *et alia* have been exploring formal techniques for assessing modularity in software design [12, 13, 14, 99]. This is a serious, high-quality research effort that (we claim) illustrates some of the shortcomings of the current Alloy surface syntax that our proposal for integrating partial instances addresses.

Cai & Sullivan have written their meta-model (L) in Z [12]. This meta-model is then implicitly encoded in the Java source of their tool Simon. Given a model of a software design in their language, Simon produces a specialized Alloy model (M) that is used to check modularity properties of the proposed software design. There is no mechanically analyzed connection between L and M.

We have translated the Cai & Sullivan meta-model from Z to Alloy and used our partial instance feature to write some of Cai & Sullivan's specific models as partial instances of this meta-model. Figure 2.9 lists our encoding of Cai & Sullivan's study of Irwin *et alia*'s example of designing a program to store and manipulate a matrix [41]. There are three variables (decisions) in this design space (line 3): the density of the matrix, the underlying data structure used to encode the matrix, and the algorithm used to manipulate that structure. More specifically, the matrix may be dense or sparse, the structure may be a linked list or an array (or other), and the algorithm may be either 'lookup' or 'traversal' (or other) (lines 4–6). In the vocabulary of Cai & Sullivan, the density decision dominates the data structure and algorithm decisions (line 7). The intuition here is that one selects

Figure 2.10 Iwrin matrix design space partial instance (Figure 2.9) extended with binding atoms generated by a previous simulation

```
1 inst IrwinMatrixDesignSpace_WithBindings extends IrwinMatrixDesignSpace {
2   Binding = B0+B1+B2+B3+B4+B5+B6+B7,
3   var = (B0+B1)→Struct + (B2+B3+B4)→Density + (B5+B6+B7)→Alg,
4   val = B0→dense + B1→sparse + B2→links + B3→array + B4→other +
5         B5→traverse + B6→lookup + B7→other }
6 run createMatrixACN for IrwinMatrixDesignSpace_WithBindings
```

the data structure and algorithm depending on whether the density is expected to be dense or sparse. Additionally, the partial instance block is followed by a list of facts (lines 9–17) that constrain valid solutions of the design space to those where the algorithm and data structure are natural matches for the matrix density and each other. A fact appended to a partial instance block can make use of the atom names introduced in that block.

The proposed partial instance feature offers Alloy users the opportunity to stage evaluation of their models, which might potentially save time when certain parts of the model are not changing and other parts are. Consider, for example, the model in Figure 2.9 that describes the design space of a program to manipulate matrices. The partial instance of Figure 2.9 is written in terms of Cai & Sullivan's meta-model, which has (design) variables, values, bindings of variables to values, and 'states'. (The 'states' are of a design automaton, which is a concept they use to analyze design spaces that we do not explain here.)

Suppose that the user wishes to experiment with the constraints written in the appended fact of Figure 2.9. These constraints do not affect the space of valid binding atoms. Therefore, the user could stage the evaluation of the model by saving the legal bindings in a partial instance, such as in Figure 2.10. Subsequent simulations would not have to resolve this part of the model.

Table 2.1 characterizes the potential performance improvements from staged evaluation using the Irwin matrix design space example of Cai & Sullivan. The translation time for the model from Figure 2.10 is over ten times faster than the translation time for the model from Figure 2.9, and the solving time is three times faster, for an overall improvement of seven times. Obviously the speedup to be gained from staged evaluation depends on the particulars of the model in question; other models will likely produce different results than this one.

Table 2.1 also shows performance results for A4.2 and A4.2′ simulating a model equivalent to Figure 2.9 (*i.e.*, not staged). In this particular case there is no significant difference between A4.2 and A4.2′. We suspect that this is the case because the domain relation is constrained piecewise across a number of appended facts. All of these piecewise constraints

Table 2.1 Performance improvements from staged evaluation of Cai & Sullivan case study

	Total Vars	Pri. Vars	Clauses	Translation time (ms)	Solving time (ms)
LEVURE (Fig. 2.9)	59,694	773	162,642	12,742	6,744
LEVURE (Fig. 2.10 — staged)	20,060	503	37,148	986	2,174
A4.2	59,953	768	162,417	11,976	27,415
A4.2′	59,953	768	162,417	11,188	27,730

add up to an exact bound on domain, but a fairly sophisticated whole-model analysis would be needed to deduce that. Answering RQ 2.1, LEVURE results in four times faster solving time than A4.2′ for the model in Figure 2.9, at the expense of a 10% slowdown in translation time.

2.4 Alternatives Considered

In this section we consider some alternative approaches for examples in Alloy and argue for the approach proposed in this chapter.

2.4.1 Static Analysis

Alloy 4.x already includes the capability to infer when constraints might be encoded as Kodkod bounds rather than as SAT clauses. Although it is not yet perfect, this capability will continue to improve. Given this capability, no extra syntax is needed to realize the main performance benefits of Kodkod's partial instance feature.

We argue that there are software engineering benefits to our new syntax beyond the performance gains that it affords. The proposed syntax makes it easy for the specifier to run different commands with different instances, or to run commands with no **inst** block (the norm in Alloy now). Writing an **inst** block implicitly via constraints in the traditional Alloy syntax makes it difficult to switch from running a command with an **inst** block to running a command without a **inst** block. For example, to run the fragment in Figure 2.7a without a **inst** block, we would want to remove the keyword **abstract** from the signature S and remove the sub-signatures S1, S2, S3. With the **inst** block syntax, one does not have to edit the text of the model to run it in these different ways. A number of our use cases described above depend on this affordance of the new syntax.

Figure 2.11 Syntactic alternatives for the body of the **inst** block

(a) object-oriented style

```
sig S{r: S}
inst i{S=S1+S2+S3,
   S1.r=S2, S2.r=S3}
```

(b) set-oriented style

```
sig S{r: S}
inst i{S={S1,S2,S3},
     r={S1→S2,S2→S3}}
```

(c) relational style

```
sig S{r: S}
inst i{S=S1+S2+S3,
     r=S1→S2+S2→S3}
```

2.4.2 Syntactic Alternatives for the inst Block

There are a variety of different ways in which one could specify the body of a **inst** block. We consider the proposal described above to be a 'relational' style because each statement specifies a different relation.

Alternatively, one could imagine an 'object-oriented' syntax in which relations are defined piecewise with respect to individual atoms. Figure 2.11a lists a small example of this syntax. The same example is listed in the relational style in Figure 2.11c. The object-oriented style syntax is intuitively appealing for some examples; however, its piecewise nature makes the bound being defined unclear: does Figure 2.11a define a lower bound or an exact bound for relation r?

Another alternative syntax is 'set-oriented' style, shown in Figure 2.11b. This style is concise and consistent with common mathematical notation, but it does not conform to the existing Alloy expression grammar.

Our proposed relational style syntax (Figure 2.11c) conforms to the existing Alloy expression grammar and has a clear and uniform way to specify lower, exact, and upper bounds.

2.5 Related Work

Torlak and Jackson [102] proposed expressing partial instances using scope definitions for finite bound analysis. When the idea was published in 2012, there was no direct support for expressing and using partial instances at the level of Alloy language and its analyzer [66]. Sullivan et al. [98] proposed a framework for unit testing Alloy models. This idea is similar to what we explained in Section 2.1 for realizing Test-Driven Development with partial instance. As an application of explicit encoding partial instances The tool has also been used for developing other ideas: for test-case generation [49] and multi-objective optimization [81]. In the former case, the generated test-cases can be saved as partial instances for the internal solver. For the latter case, the optimizer employs a multi-step

solving process and, the output of each step could be transferred to the next step in a partial instance block. Cunha et al. [20], Macedo and Cunha [61] employed partial instances to accelerate their technique for bidirectional model transformation.

2.6 Summary

Exposing Kodkod's partial instance feature in the form of explicit examples increases the computational efficiency of Alloy Analyzer for producing discriminating examples and general Alloy users. In this chapter, we have showed evidence of **Contribution 5**. Our proposed syntax can efficiently realize the idea of bug finding with explicit testing, as explained in Section 2.1. Our extension also makes staged model solving feasible by decreasing the size of universe of discourse for particular formulas, such $\exists\forall$ discussed in Chapter 3.

In addition, the extension facilitates test-driven development of Alloy models; regression testing of Alloy models; to support ideas such as example-driven modeling and combined modeling and meta-modeling. While Alloy currently has an inference mechanism that makes use of Kodkod's partial instance functionality behind the scenes, these engineering benefits are substantially facilitated by explicit syntactic support for examples.

There is more than one possible way to expose Kodkod's partial instance feature to the Alloy user. We have explored a number of alternatives, and recommend a new named block with statements written in a relational style. This recommendation is backwards compatible with existing Alloy models and the existing Alloy expression grammar; it affords the user a uniform way to express exact, upper, and lower bounds; it combines with Alloy commands in a modular fashion; and it can be easily and efficiently translated to Kodkod.

Chapter 3

BENTONITE: An Extension of Alloy for ∃∀ Queries

The kinds of properties that the Alloy Analyzer is usually asked to run or check are generally of the form 'find an example' or 'find a counter-example'. Less commonly, in historical practice, are analyses of the form 'there exists x for all y ($\exists\forall$)', where both x and y are two composite structures., *i.e.*, signatures embodying a set of fields in Alloy terminology. These kinds of analyses are cumbersome for the user to specify in the Alloy language, and the analyzer does not always scale to the required computation.

Finding discriminating examples that are near borders needs the queries in the form of $\exists\forall$. As explained in Chapter 4, a query for finding the relative minimal distance between an example and a non-example can be specified such that there exists a distance that is shorter than all other distances. Having a solver to run such queries is a must to produce discriminating examples near borders. In general, we have identified two major applications for $\exists\forall$ statements: performing model checking analysis and finding optimum instances.

Researchers attempt to use Alloy for analyzing model checking of queries (*e.g.*, [19, 24, 30, 106]). In this case, a user typically wants to check whether every state in a transition system must have a successor: '**all** x : State | **some** y : State | next[x,y]'. To refute this proposition, Alloy Analyzer tries to find an instance consistent with the negation of the proposition, which is in the form of $\exists\forall$, *i.e.*,'**some** x : State | **all** y : State | **not** next[x,y]'. Without taking care of generator axioms, such as using **uniq** block, Alloy Analyzer will then construct a truncated universe in which some State does not have a successor, and the proposition will be reported as false, *i.e.*, *spurious counter-example*. To assess BENTONITE in analyzing these kinds of applications, we evaluate it on FORML [24, 25], which need the analyzer to exhaustively

explore the declared world states so as to check the assertion in the form of $\exists\forall$.

Researchers have used Alloy Analyzer and its underlying solver Kodkod [102], to find optimum instances with respect to some objectives [20, 81, 85, 87]. In essence, a statement in the form of $\exists\forall$ can specify a query for some optimum instances compared to all other instances. After we published the ideas behind BENTONITE to facilitate model checking with Alloy, researchers at MIT developed Alloy* to analyze statement needing quantification over relations [65]. They showed how the tool can be also used for analyzing $\exists\forall$ statements without worrying about insufficient scopes.

In this chapter, we describe how staging the analysis improves the scalability and speed of the analyzer for these $\exists\forall$ analyses. We have also developed a small syntactic extension to the Alloy language, listed as **Contribution 6**, that makes it easier for the user to specify these kinds of analyses. In brief, the user adds the new **uniq** ('unique') keyword to the signatures (sets) that she would like to use to generate all possible objects of (the domain of the $\forall$ in the $\exists\forall$ query). From this keyword, a *generator axiom* [42], is automatically synthesized, which forces all of the desired atoms to exist in the universe of discourse. Previously the user would have had to write this generator axiom by hand [42] and specify the size of the universe of discourse, both of which are inconvenient almost to the point of being unusable.

The Alloy Analyzer without staging will attempt to solve the generator axiom and the $\exists\forall$ query simultaneously. This approach often does not scale to problems of interest, or is unacceptably slow. Our staged Alloy Analyzer, called BENTONITE, first generates all of the atoms that satisfy the generator axiom, then records these as a *partial instance*, as explained in Chapter 2, which is combined with the $\exists\forall$ query in the second stage.

As we show in this chapter, Alloy* works efficiently when the existential quantifier of a $\exists\forall$ statement is expected to be skolemized, whereas BENTONITE works more quick once such a statement is unsatisfiable, *e.g.*, the analyzer fails to refute the proposition. Through the rest of this chapter, we will explain our proposal for BENTONITE. However, we have used Alloy* to prototype BORDEAUX, expressed in Chapter 4. Our preliminary experimental studies have shown that it works more efficiently for BORDEAUX.

The remainder of this chapter is as follows: Section 3.1 provides an example to clarify the idea for extending Alloy. Section 3.2 illustrates how we syntactically and semantically augmented Alloy to solve the problem. Section 3.3 demonstrates our techniques for supporting the extension, and Section 3.4 shows experiment results for performance evaluation.

3.1 Illustrative Example

For illustration, Figure 3.1 lists a simple example of an address book (adapted from [42]). This address book, encoded as Book signature, comprises a set of names and a set of mappings (addr) of those names to aliases or locations.

Figure 3.1 Address book example (adapted from [42])

```
1  abstract sig Target {}
2  sig Loc extends Target {}
3  abstract sig Name extends Target {}
4  sig Alias, Group extends Name {}
5
6  uniq sig Book {
7    names: set Name,
8    addr: names→some Target}{
9    no n: Name | n in n.^addr
10   all a: Alias | lone a.addr}
11
12 pred add [b, b': Book, n: Name, t: Target] {
13   t in Loc or some (n.^(b.addr) & Name & t)
14   b'.addr = b.addr + n→t
15   b != b'}
16 pred del [b, b': Book, n: Name, t: Target] {
17   b != b'
18   no b.addr.n or some n.(b.addr) − t
19   b'.addr = b.addr − n→t}
20 assert InsertORRemove{
21   all b:Book|some n:Name,t:Target,b':Book| add [b,b',n,t] or del[b,b',n,t])}
22
23 check InsertORRemove for 0 but exactly 2 Loc, exactly 2 Alias, exactly 0 Group
```

The predicates add and del specify what it means to add and remove pairs from this mapping, *i.e.*, a relation to other entities. The assertion InsertOrRemove says that, for any address book, if we add or remove some pair from its mapping ,we will get another legal address book. Alloy Analyzer checks the assertion using refutation, which means it tries to find an instance satisfying the negation of property. If such an example is found, the property does not hold; otherwise, the model is consistent with the property within the given scope; that is, the result of add or remove operations leads to another atom of the book.

Checking this property requires a generator axiom; otherwise, a generator axiom the analyzer could produce a spurious counter-example by simply failing to generate some particular address book. The listing in Figure 3.1 implicitly includes such a generator axiom by use of our **uniq** keyword (line 5).

A generator axiom must embody some concept of object identity. The generator axioms implied by our **uniq** keyword treat two objects as different if they have different field values. This is different from the default concept of atom identity in Alloy (which does not consider field values), but is a meaning that users sometimes write in their models.

Before going through the rest of the chapter, some terms are defined: *a*) *Unique objects* are the objects of a signature that differ in at least one of their fields; *b*) *Possible objects* are all unique objects that a signature can have regardless of any constraints defined in the model; *c*) *Legal objects* are the possible objects of a signature that are also consistent with the related constraints in the model; and *d*) *illegal objects* are the possible objects of a signature that are inconsistent with the related constraints in the model.

3.2 Language Extension

Since Alloy runs and checks a predicate in a finite universe of discourse, the quantified variables have to be bound. Quantifying over composite structures, *i.e.*, a 'signature' embodying a set of fields in Alloy terminology, has to be bound as well. The bound of a composite structure depends on the bound of other related structures. The bound is infinite if the uniqueness of a composite signature's objects is not considered, or the signature directly or indirectly has a relation to itself [42]. However, depending on the fields of a signature, unique objects can be intractably large. Despite this fact, a restricted subset of possible objects of a signature is interesting. This subset can either be determined by a set of constraints or given as an **inst** block [§2]. Still, determining the scope of a composite structure with respect to the constraints is a hard problem [51]. This issue becomes critical for simulating $\exists\forall$ statements when the universal quantifier enumerates over objects of composite structures. Below, we express the syntax and semantics used in developing Bentonite.

3.2.1 Syntax and Semantics

We propose adding a new keyword to Alloy: *uniq*. This keyword indicates that all 'unique' atoms should be generated for a signature, *i.e.*, **Contribution 6**. What does 'uniq' mean? By definition each atom is unique. Our intended meaning extends the metaphor between Alloy and object-oriented programming languages. By 'uniq' we mean not only that each atom is unique (which is obviously already the case), but that no other atom of the same type has the same 'field' values. The metaphor we are extending is that the declarations

of relations in Alloy are syntactically arranged to look like field declarations in an object-oriented programming language. This is the same approach to generator axioms taken in Jackson's book [42].

Regardless of the arity or multiplicity of the relations, the form of the constraint that requires all atoms to be unique remains the same: simply that no two distinct atoms have the same field values. Figure 3.2 lists the general form of an Alloy signature. There are n relation declarations, r_1 through r_n. Each relation declaration names some number of other types (usually signatures) T^i, each with an associated multiplicity declaration m^i. The number of types named may be different for each relation r. Finally, there is an appended fact, constraints on which atoms are legal, that we here name Φ[this] because it is implicitly quantified over each atom of signature S.

Figure 3.2 General form of an Alloy signature

```
1 uniq sig S {
2   r1 : T1^1 m1^1 → ... → m1^i T1^i,
3   ⋮
4   rn : Tn^1 mn^1 → ... → mn^j Tn^j}
5   {Φ[this]} −− constraints that define legal atoms
```

The general form of our synthesized uniqueness predicates is listed in Figure 3.3. The idea is simply that any two distinct atoms of a **uniq** signature have some difference in at least one of their field values. This constraint generate neither the atoms nor the tuples. With respect to Φ[this], Alloy Analyzer generates a proper number of legal objects. A

Figure 3.3 General form of a synthesized uniqueness predicate

```
1 fact Unique {
2   all disjoint s,s':S | s.r1!=s'.r1 or ... or s.rn!=s'.rn
3 }
```

proposal of the implementation is explained in Section 3.3. Apart from the implementation technique, we have semantically expressed the generator axiom as a fact, called generator fact (or legal predicate), that could be integrated into the model to ensure that all the possible objects of the signature are properly generated.

The legal predicate, Φ[this], can be semantically integrated into the model by splitting the **uniq** labeled signature into the legal and illegal signatures. To do so, we change the signature to an abstract signature and extend it into *legal* and *illegal* concrete signatures (*e.g.*, Legal_S and Illegal_S extend the abstract signature S in Figure 3.4). Since an

abstract signature does not have any instances, all its instances are partitioned into its sub-signatures. In this case, the integrated legal facts state that any instances in the super-signature also belongs to the legal sub-signature, if it is consistent with the legal predicate. This form of integration only introduces a super-signature of the generated signature, which is hidden from the user.

Figure 3.4 Semantically integrating the generator axiom into the model

```
1  uniq abstract sig S{
2    p:X}{
3    Φ[this]}
4  sig Legal_S, Illegal_S extends S{}
5  fact Gen {
6    all s:S | (s in Legal_S) ⇔ Φ[s]
7    all x:X |some s:S |s.p = x }
8  ————————————————————————————————————————
9  sig Legal_S, Illegal_S extends S{}
10 fact Gen {
11   all s:S | (s in Legal_S) ⇔ Φ[s]
12   all t_1^1:T_1^1,...,t_1^i:T_1^i,...,t_n^1:T_n^1,...,t_n^j:T_n^n | some s:S |
13         s.r_1 = t_1^1 → ... t_1^i and ... and s.r_n = t_n^1 → ... t_n^i}
```

If the scope computation is a key concern in supporting generator axioms, the signature symmetry in the relation declaration seems to be a critical issue. A signature probably directly or indirectly has a relation to itself, such as in the case that T_1^1 is replaced by either S or an extension of S in Figure 3.2. These kind of relations make a circular dependency between the given bounds, including the signature, and the relation's unique tuples. Therefore, the number of distinct tuples cannot be determined, and this kind of relations has to be excluded from the uniqueness predicates as well as from the Cartesian product of exact bound calculation. In response to this issue, we raise an argument to see whether or not a relation containing the signature or an extension of it on the range plays a role in the number of unique and legal signature instances. As a case, if the relation specifies an order between the signature instances, then the recursion definition issue matters; otherwise, it can be ignored. For example, in a simple declaration of Natural numbers, *e.g.*, '**sig** Natural{next:Natural}', the next relation is a total order between two Naturals. In contrast, in a simple model of a state machine, *e.g.*, '**sig** State{trans:State}', logically, there is no total order between two States by trans. In essence, the number of signature objects does not depend on the signatures that are not in an ordered relation with.

3.2.2 Soundness and Completeness

BENTONITE is sound in finding an instance that satisfies a predicate in the form of $\exists\forall$. Consider $\forall a : A|\ \exists b : B|\ \phi(a, b)$ as the predicate such that A and B are two composite signatures labeled with **uniq**. The universe of discourse includes all legal objects of A and B with respect to corresponding constraints.

The instance returned by BENTONITE is sound. An instance means that there is an object of B that satisfies ϕ with all objects of A. Using **uniq**, BENTONITE restricts the scopes of the objects A and B to the legal objects. As Alloy Analyzer soundly quantifies over the finite universe encompassing all legal objects, the returned instance is sound. The search by BENTONITE is complete. If it does not find any instance, there is no such instance in the finite universe. Because Alloy exhaustively searches in the finite universe and that universe contains all legal objects, the search by BENTONITE is complete.

3.3 Techniques

In Alloy, to run or check any predicate quantifying over composite structures, a generator-axiom and an adequate upper-bound have to be specified. Choosing adequate upper-bound for composite structures is a critical step. The scope has to be large enough that it encompasses all the legal objects distinguished by generator axioms. Randomly picking a large bound makes the checking problem infeasible. Moreover, since the number of possible objects of a composite structure depends on its fields, any changes in the arity, multiplicity and scope of other signatures of the fields will lead to another upper bound.

With respect to the semantics of **uniq** keyword, we have proposed two techniques to ensure that an upper-bound is properly set and generator axioms are satisfied. The user confidently checks the predicates quantifying over **uniq** labeled signatures. Therefore, the techniques are based on supporting partial instances at the level of the underlying solver [102] and Alloy [§2]. Using partial instances, the techniques separate the one-step solving into two major stages. In the first stage, the universe is properly created, and in the next, the appropriate properties are checked over the created universe of discourse.

3.3.1 Non-staged solving

To ensure that all legal unique objects of a signature exist in the universe, Alloy users currently first have to make sure that all the generated objects are unique. The unique

predicate is written by hand, and totally depends on the structure of the signature. Ensuring the dependency between the signature structure and the body of a unique predicate is error prone, and unwieldy to maintain.

In order to ensure that Alloy exhaustively searches for an instance, such as while checking for a safety property, the scope of the signatures has to be exactly determined in the universe. Finding the exact scope is hard in theory (see [51]) and practice. Alloy users generally use a trial-and-error approach for finding the maximum unique objects of a signature. A user first guesses the number of the exact scope of a signature, and runs the Alloy code to see whether any object is found. If an object is found, the number is increased; otherwise it is decreased. The process is continued until the proper exact scope is found, then the model is checked for a given property. However, the universe construction and property checking are performed in one call to the SAT-solver.

3.3.2 Staged-solving using Kodkod

This technique finds a proper upper-bound by systematically checking different scopes. It looks for a concise scope through a step-by-step searching. The result of each step is reused in the next step so as to reduce the time and space complexity. The basic idea is to begin from a small exact scope, increase it with step by step, until it passes a scope making UNSAT result; then, it terminates or roll-backs until it finds the proper answer if the increase step, the number of atoms should be added or removed from the current scope, was more than one. The technique takes advantage of the the proposed feature in Alloy [§2] for specifying partial instances (which has been supported in Kodkod for some time). As shown in Section 3.2, this iteration will be eventually terminate and the precise upper-bound will be reached.

The model that is going to be checked for a property is partitioned into two slices. One slice includes a signature labeled with **uniq** and all constraints on it, and the other includes the rest of model. The first slice, along with a universe of all the signature atoms that are referred in the **uniq** labeled signature and fields is passed to the solver to see whether or not there is one object satisfying the constraints in the slice. If so, it means there exists an object of the **uniq** labeled signature with in the given initial universe.

In order to determine whether one more unique and legal object of the signature can simultaneously exist, a uniqueness predicate, as explained in Section 3.2.1, is constructed and conjoined with the previous slice. The result will be passed to the solver along with an incremented universe that includes the atoms from the previous universe and the found object(s) in the last step. The already found object(s) is wrapped as lower-bound tuples

of the signature's fields, so the solver does not need to search for the part of the solution that has already been found. The iteration is continued until the constraints become unsatisfiable within the incremented universe. If the increament was one, the last satisfiable universe is considered as a universe for checking the intended property over the whole model or fed into the universe construction of the next unique labeled signature.

3.3.3 Staged-solving using a SAT-solver

Currently, modern SAT-solvers, like MiniSat [28], provide the capabilities to get the next instance for a given SAT formula, or for incremental solving. To do so, the SAT-solver complements the current found instance and adds it to the current solving formula. Since the SAT-solver is not asked to solve a new formula, it reuses whatever it learned while finding the previous instances. It then finds the next instances, often more quick than the first one. The SAT-solvers shipped with the Alloy package support incremental solving. Users can run models, see a satisfying instance, and go for the next one if exists.

We utilized this SAT-Solvers' feature to find all legal objects of a signature. Apart from implementation details, after slicing the model in the way described previously, this technique also considers the slice containing the signature labeled **uniq**. Then, the technique passes this slice, plus a universe of all atoms of the referred signatures in the constraints and the signature's fields, into the underlying solver.

The technique repeatedly calls for the next object until the solver returns nothing. Any instance returned by the solver is a unique legal object that has to be added to the final universe. Since Kodkod applies symmetry breaking predicates to a given constraint, the solver may not return all legal instances of the slice. We therefore turned off the symmetry-breaking feature to get all objects. Finally, using all the returned objects, this technique makes the universe and wraps it as a partial-instance block. In the second major stage, the property is checked with a complete scope that is specified as a partial-instance block.

3.4 Experiment

In this section, we present the empirical evidence to find answers to the following questions:

RQ 3.1. *How efficient is the staging technique versus non-staging for running* $\exists\forall$ *formulas?*

RQ 3.2. *How does the performance of staging technique compare with CEGIS (Alloy*) for optimization (expect* SAT*) and property checking (expect* UNSAT*) application of ∃∀ formulas?*

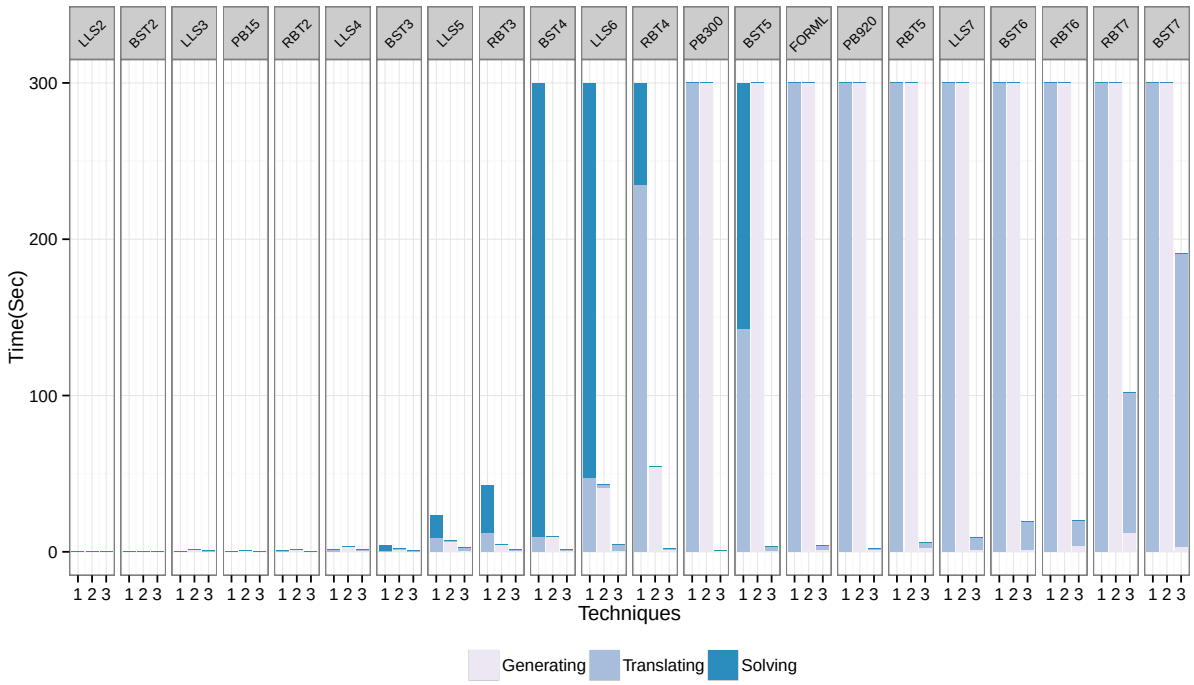
To find the answer of RQ 3.1, we examine the performance of staging and non-staging techniques on a collection of common data structure models, the phone book example of Figure 3.1, and a model automatically generated from the FORML [24, 25] analysis tool. In the staged approaches the generator axioms are solved first, and then a statement in the form of ∀∃ is checked. BENTONITE refutes the property by trying to find a counter-example for its negation of the property in the form of ∃∀. In the non-staged approach the generator axioms and the property are solved simultaneously. But first, a sufficient upper-bound has to be properly chosen. Since finding a proper upper-bound is done by trial-and-error, it is hard to measure the effort and time needed.

The data structure models we examine are sorted linked-list, binary search tree, and red-black tree. Each model has a predicate to insert, remove, and lookup an integer value in the specified data-structure. Then we check that every possible object (∀) of the data structure can be transformed into other objects of the data structure (∃) by inserting or removing an item. This is similar to the property specified for the phone book example in Figure 3.1. In order to avoid the recursive spirit of data structure models, we transform the recursive fields to a new signature and label this signature as being uniquely generated before checking the property. For example, in modeling sorted linked-list, instead of one signature for specifying a node like '**sig** Node{value: **Int**, next: Node}', we made two signatures, one for the node and another for the links between two nodes. Using this transformation, the number of generated lists will be based only on a fixed number of nodes that were generated before.

FORML is a language designed to describe product line features with an associated analysis to detect their interaction [24, 25]. The FORML analysis works by translating FORML into Alloy and then running the Alloy Analyzer. The Alloy models generated by FORML include generator axioms and ∀∃ queries. The FORML research team had found that manually staging their analysis into universe construction and property checking phases (enabled by **inst** block [§2]), produced dramatic performance improvements, a finding that in part motivated this current work. Here we show that our automatic staging generalizes their manual approach and produces similar improvements for a variety of models.

Figure 3.5 shows that staging using SAT-solver reduces the overall amount of time taken to check the property, and also improves scalability (the number of problems that can be checked within a given time bound). The first bar in each cluster of Figure 3.5 shows the times taken with/without staging, the second and third bars show the times taken

Figure 3.5 Staging improves scalability and reduces runtime of ∃∀ queries. Analysis times (bars) are broken down into three components: object generation; ∃∀ query translation, which is performed by Kodkod and includes the generated objects as a partial instance; and SAT solving the ∃∀ query. The x-axis is organized by benchmark and technique. Technique 1 is not-staged; 2 incremental growth; 3 is enumeration. Lower bars are better. Bars that hit the top are time-outs. Enumeration is the only technique that solves every problem. These are the same benchmarks as in Table 3.1.



with staging using Kodkod and SAT-solver, respectively. The times taken with staging are broken into three components: generation stage, ∃∀ query translation from relational to propositional logic (Kodkod), and the time to solve the propositional formula (SAT-solver). The times taken without staging are broken into two components: relational to propositional logic translation and propositional solving.

Figure 3.5 is sorted from the smallest problem at the left to the biggest problem at the right. For sufficiently small problems all three approaches work quickly (everything from BST3 to the left). The non-staged approach almost always completes quickly or times out: there are only two problems (LL5 and RBT3) that take a medium amount of time. The staged approach using Kodkod always out-performs the non-staged approach, but times

Table 3.1 Staging results with significantly reduced number of variables and clauses in CNF file generated to check ∃∀ query. Numbers in the column *size* represent the number of nodes in the studied data-structure models. This concept of size is not applicable to *Phone Book* and FORML models.

Specification	Size	Gen. inst.	Staged				Not-Staged	
			Enumeration		Incremental			
			Vars	Clauses	Vars	Clauses	Vars	Clauses
Sorted Linked List	2	4	19	31	18	28	1,892	3,964
	3	9	59	103	58	99	7,089	16,401
	4	25	166	323	166	323	33,327	78,833
	5	92	489	1,159	489	1,159	309,974	715,938
	6	458	1,849	5,868	TO	TO	TO	TO
	7	2,987	9,957	41,351	TO	TO	TO	TO
Binary Search Tree	2	5	25	42	24	36	4,858	9,559
	3	15	138	313	138	313	28,514	61,258
	4	51	386	938	386	938	214,564	486,202
	5	188	1,192	3,246	TO	TO	2,179,446	4,955,942
	6	731	3,345	9,766	TO	TO	TO	TO
	7	2,950	10,883	34,932	TO	TO	TO	TO
Red-black Tree	2	5	24	42	24	42	19,094	46,068
	3	12	80	154	80	154	143,220	389,576
	4	29	295	685	295	685	900,413	2,584,570
	5	74	1,669	4,477	TO	TO	TO	TO
	6	201	5,653	16,794	TO	TO	TO	TO
	7	573	10,353	30,965	TO	TO	TO	TO
Phone Book	NA	15	614	912	614	912	6,597	10,693
	NA	300	15,341	110,191	TO	TO	TO	TO
	NA	920	55,341	919,316	TO	TO	TO	TO
FORML	NA	170	903	1,582	TO	TO	TO	TO

out for the bigger models (PHONEBOOK300 to the right). By contrast, the staged approach using SAT-solver is able to complete quickly or in a reasonable amount of time for every problem we studied. The symmetry breaking was turned off for generating instances, and turned on for checking the assertion.

Table 3.1 shows the number of variables and clauses in the CNF file generated for checking the ∃∀ query. In the non-staged approach, this CNF file also contains variables and clauses for generating the universe. The highlighted rows in Table 3.1 indicate the largest problem size that the non-staged approach was able to complete in a reasonable time bound (5 minutes). The highlighted rows for the data structure problems show an improvement for staging using SAT-solver of three orders of magnitude. The highlighted rows for the phone book problem show only an order of magnitude improvement for staging, in part because the non-staged approach is only able to solve or translate this problem at a small size within the given time limit. The non-staged approach is not able to solve the

Figure 3.6 Alloy statements for expressing a property that checks whether the insert and remove operators over appropriate data structures are valid.

Style 1

```
1 assert{ all s: SLL| some s': SLL| some i: Int|
2         insert[s,i,s'] or remove[s,i,s'] }
```

Style 2

```
1 pred { some s:SLL | no s': S | some i: Int|
2        insert[s,i,s'] or remove[s,i,s'] }
```

FORML problem at all in the given time limit.

Since both staging approaches lead to the same upper-bound, the solver generates the same number of SAT variables in clauses. Compared to the non-staged approach, if the staging using SAT-solver approach completes, it generates magnitude smaller SAT formulae. BST5 problem looks like an exception here. Although, the non-staged approach generated the SAT formula, the solver could not finish it in the remaining time. In general, in either staging approaches, the translating and solving steps are negligible compared to the generating phase.

Answering RQ 3.1 in brief, creating separate stages for the generator axioms and the $\exists\forall$ query reduces the time added to check the query and increases the number of problems that can be solved within a reasonable time bound.

Unlike our staging solution for generating the entire universe before checking the assertion, Milicevic et al. [65] developed an extended Alloy Analyzer, called Alloy*, using CEGIS (counterexample guided inductive synthesis) technique to verify whether a found counter-example is spurious before reporting it to the user. The authors mentioned that generating the whole universe can be prohibitively expensive. Although this claim looks to be a reasonable intuition for some cases, such as the examples mentioned in [65], for some cases, their approach can be less scalable than generating the whole universe. Alloy* works well when one looks for a particular instance in a small universe, but it could behave randomly when the universe is large. iOn the other hand, BENTONITE outperforms Alloy* once the model is UNSAT.

To compare these two tools, we evaluate them on *Singly-linked List* (SLL) and *Binary Search Tree* (BST) models. Each model has two predicates, one for adding and one for removing integers to and from the data structure. The predicate may be valid or invalid, *i.e.*, bogus. We check the models to see whether a state of a data structure is reachable such that neither operation is applicable. To do so, we model such a statement in two styles as represented in Figure 3.6.

Depending on the validity of insert or remove, the plots in Figures 3.7d, 3.8d, and 3.9d show how the solvers check assertions and predicates over SLL and BST with different numbers of nodes.

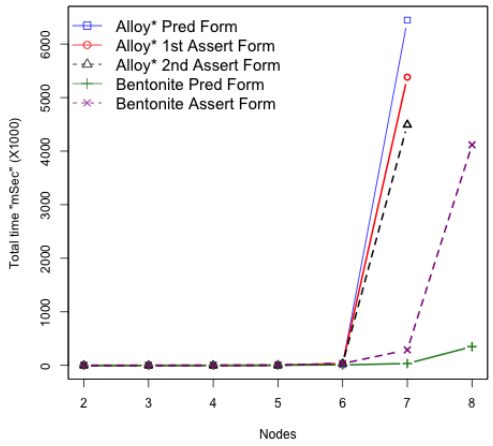
The plots in Figure 3.7 show the time needed to complete the *predicate*-style and *assertion*-style expressions with Alloy* and BENTONITE. Alloy* is also evaluated in two logically equivalences of the assertion-style. The first assertion-form follows '**all** a:A | valid[a] **implies** (...)' and the second is in the form of '**all** a:A | **not** valid[a] **or** (...)'. The data structures are checked for two to eight nodes. The X-axes of the plots in all figures are the size of each data structure, *i.e.*, the number of nodes. The Y-axes in Figure 3.7 are the total time taken to complete the process. The total time includes the generation time, translation time, and solving time in BENTONITE and translation time, process time, and solving time in Alloy*.

In Figures 3.7a and 3.7b, Alloy* always takes a longer to confirm the correctness of operation. It cannot finish the process within ten hours for SLL and BST of size eight. The predicate-style is always completed sooner using BENTONITE. Surprisingly, as the plots in Figure 3.9 show, the numbers of SAT clauses and variables are equal in both styles. But using Alloy*, the predicate-style can be done in a shorter time in BST or longer time in SLL. It is not clear to us why the second assertion-form takes less time than the first assertion-form in Figure 3.7a. The only different between the two forms is a simple replacement of the *implication* operator with an *or* operator. As Figures 3.7c and 3.7d show, Alloy* finds a counter-example quicker than either assertion style with BENTONITE.

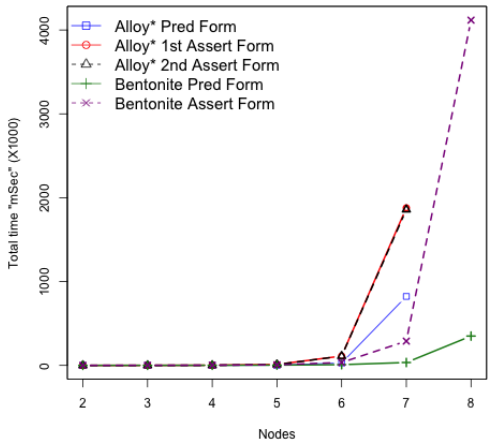
Figures 3.8a and 3.8b show how many instances, or candidates, are generated to be checked in both new solvers. As expected, Alloy* generated fewer candidates to be checked for BST. Notably, Figure 3.8a shows that Alloy* generated more candidates than BENTONITE did to confirm the correctness of SLL. Also from Figure 3.8c, Alloy* acted randomly to generate candidates, whereas the *remove* operator is an invalid operation in SLL. On the other hand, compared to BENTONITE, Alloy* produces more SAT clauses to check both data-structures (Figures 3.9a and 3.9a), because BENTONITE mainly uses a SAT-solver for generating the universe, so the final assertion check is done by Kodkod. However, it is not clear why Alloy* generates far fewer SAT variables for BST as opposed to SLL from Figure 3.9d.

Both solvers enable Alloy users to specify logical expressions needing a higher-order quantifier. Answering RQ 3.2, the preliminary experiments show that BENTONITE is more scalable when the entire universe has to be exhaustively checked, whereas Alloy* can find an actual counter-example faster, if one exists.

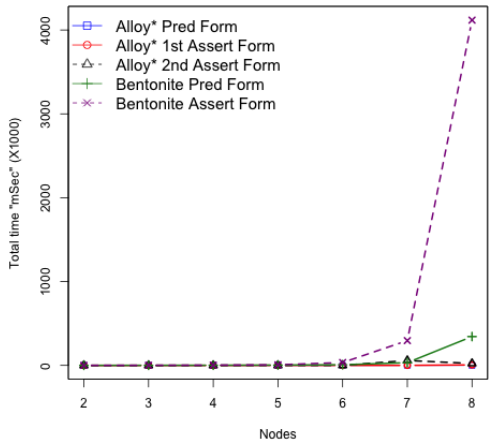
Figure 3.7 Time *VS.* Number of nodes for Singly-linked List (SLL) and Binary Search Tree models (BST). The charts show the total time (translation+execution) that Alloy* and BENTONITE spend for checking the correctness of the statements expressed in Figure 3.6 over SLL (left charts) and BST (right charts). The x-axis range from 2 nodes to 8 nodes that the data-structures already have before any node insertion or removal. In the top charts, insert and remove operators are valid so that the models are expected to be UNSAT. In the bottom charts, the remove operators are invalid (bogus), but the insert operators are valid so that the analyzers have to return a counter-example, *i.e.*, the models are SAT.



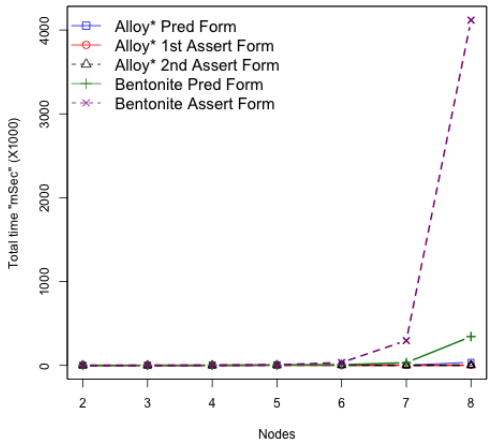
(a) Assertion: Valid Remove Valid Insert



(b) Assertion: Valid Remove Valid Insert

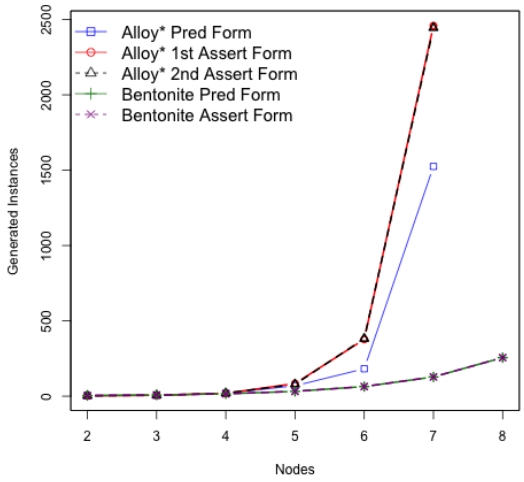


(c) Assertion: Invalid Remove Valid Insert

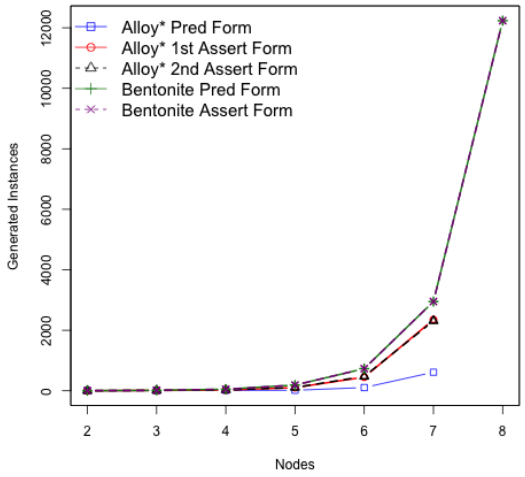


(d) Assertion: Invalid Remove Invalid Insert

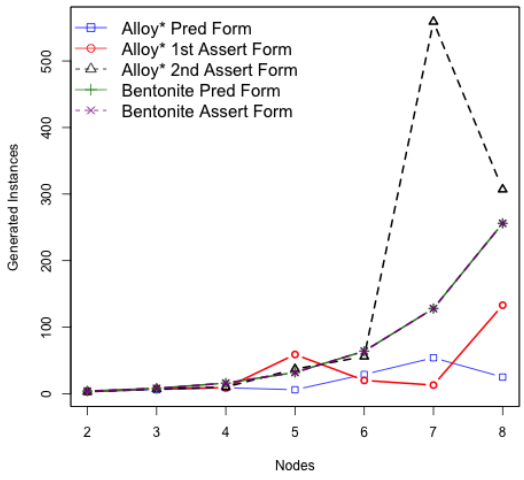
Figure 3.8 Number of instances *VS.* Number of nodes for SLL (Left charts) and BST (Right charts). The number of instances in BENTONITE is the exact number of SLL of BST atoms that should exist in the universe to avoid any spurious counter-example. For Alloy*, the number of instances is the number intermediate instances generated by CEGIS.



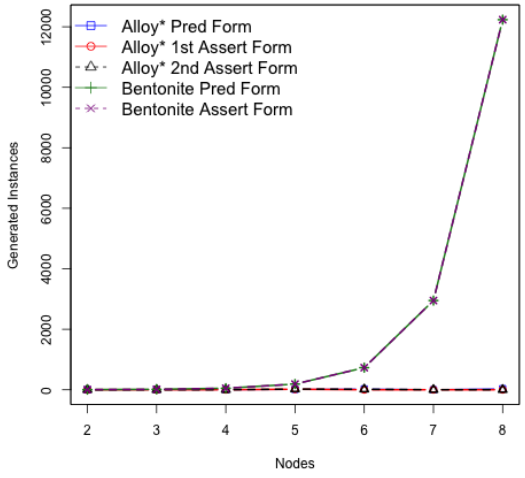
(a) Assertion: Valid Remove Valid Insert



(b) Assertion: Valid Remove Valid Insert

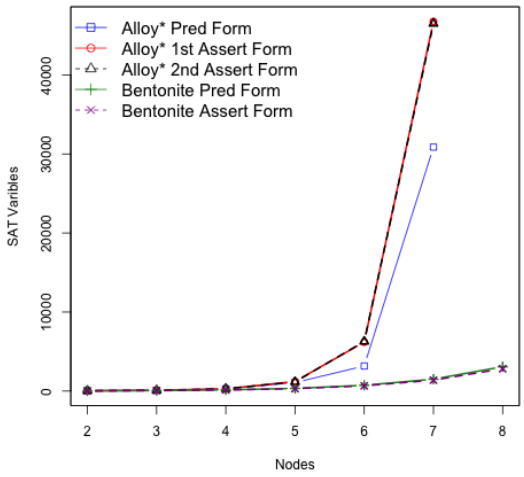


(c) Assertion: Invalid Remove Valid Insert

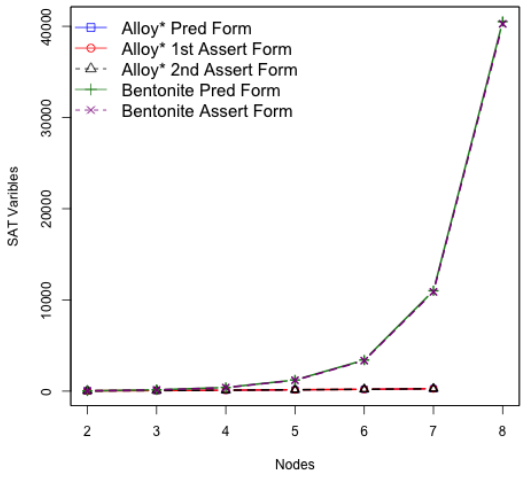


(d) Assertion: Invalid Remove Valid Insert

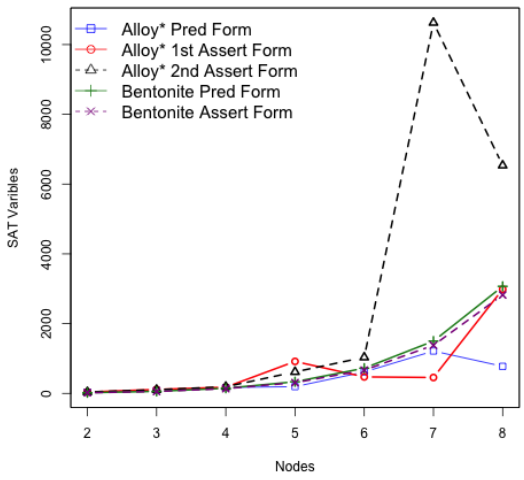
Figure 3.9 Number of SAT variable *VS.* Number of nodes for SLL (Left charts) and BST (Right charts).



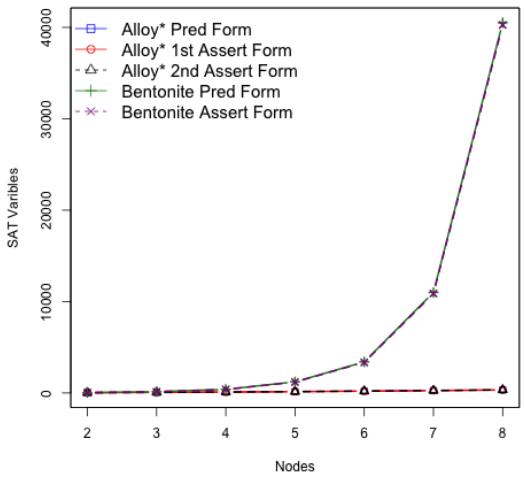
(a) Assertion: Valid Remove Valid Insert



(b) Assertion: Valid Remove Valid Insert



(c) Assertion: Invalid Remove Valid Insert



(d) Assertion: Invalid Remove Valid Insert

3.5 Related Work

Finding all distinct consistent instances of a propositional logic formula is called #SAT, or the model counting problem. Formally, the model counting problem is #P-complete; it is complete for problems as hard as polynomial-time hierarchy (PH) problems [100], and dramatically harder than an NP-complete SAT problem. Researchers tackle #SAT with both exact and approximate techniques. Biere [10, §20] reports that exact techniques scale to hundreds of variables and approximate techniques (with guarantees on the quality of the approximation) scale to a few thousand variables. When Alloy users need to write generator axioms, they usually also need to specify the correct scope, which can be difficult. Both of our staging approaches compute this scope for the user 'dynamically', by solving a sliced version of the specification. In Margrave [76], a security policy checker, Nelson et al. [77], claimed that a signature's scope can be approximated statically if predicates in the specification belong to a particular subclass of first-order logic.

The idea of generating all legal instances is also of interest in the context of test-case generation (*e.g.*, [1, 31, 50]). Khurshid and Marinov [50] used SAT-solver based enumeration with Alloy for this purpose. Symmetry breaking can significantly reduce the number of instances generated when using SAT-solver based solution enumeration [51, 102]. We require symmetry breaking to be turned off for now, as our experiments have shown that symmetry breaking can still lead to spurious counter-examples for $\forall\exists$ queries. Ganov et al. [33] used model partitioning and incremental solving techniques for developing domain specific solvers based on Alloy. Their approach uses annotations for partitioning the model and distributing the parts among different solvers.

3.6 Summary

Staging using SAT-solver based solution enumeration and partial instances enables the Alloy Analyzer to scale to larger $\exists\forall$ queries than were previously feasible. This type of queries can be used for finding an optimized solution, expecting SAT, or model checking, expecting UNSAT. In Chapter 4, we use queries in this form for finding discriminating examples near borders.

Alloy models with $\exists\forall$ queries usually require *generator axioms* [42]. A common pattern for generator axioms is to treat atoms as 'objects' that are distinguished by their 'field' values. We propose a new keyword for Alloy, **uniq**, from which such generator axioms (and their associated equality predicates) can be automatically synthesized, *i.e.*, **Contribution**

6. This keyword makes it clearer and easier for the user to specify their intent and facilitates the staging techniques.

Our staging approach is to first solve the generator axioms, and then feed the resulting atoms into the property checking stage as a partial instance. Our experiments of checking such queries for model checking show that this staging strategy reduces the number of variables and clauses in the CNF file generated for the property checking stage by orders of magnitude, and thereby scales the analysis to a larger number of potential problems. Also, our empirical study shows that the staging strategy has better performance than CEGIS technique for the models that are expected to be UNSAT but is outperformed by that technique for satisfiable models. Alloy*, a tool using CEGIS, was developed after BENTONITE, so we have used it for our further studies.

Chapter 4

BORDEAUX: An Extension of Alloy for Producing Near-border Examples

Examples can help people understand abstractions [4, 34, 110, 115] such as models. One of the great features of the Alloy Analyzer is that it can mechanically generate examples of the user's model (formula). These examples are consistent with the model. If the user deems the generated example undesirable, then it is a concrete representation of an *underconstraint* problem in the model: the model needs to be tightened to exclude the undesirable example. The Alloy Analyzer generates examples arbitrarily, without specifically targeting towards either desirable or undesirable examples.

A facility for generating *near-miss examples* (*i.e.*, non-examples) might help the user diagnose partial overconstraint bugs. What the user might like to see is an example that is formally excluded by the model but which she actually intends the model to include (*i.e.*, is desirable). Cognitive psychologists have found that near-miss examples revealing contrast are effective for human learning [34].

A simple, if inconvenient, technique for generating non-examples is to manually negate the model and use Alloy's existing example generation facility. But the chances of this technique generating examples that are desirable is slim, since there are typically so many more non-examples than examples. The chances of a near-miss example being desirable are higher, because a near-miss example is similar to examples that are desirable.

We have developed a technique and prototype tool, named BORDEAUX, for doing *relative minimization* of examples. Given mutually inconsistent constraints, A and C, it will search for examples a and c, respectively, that are at a minimum distance to each other (measured by the number of tuples added or removed).

To find a near-miss example for A, simply set C to be the negation of A and commence the relative minimization procedure. We say that a is a *near-hit* example and that c is a *near-miss* example (both of A with respect to C). The space between the near-hit and the near-miss is the *border*: there are, by definition, no examples of either A or C on the border. Examples consistent with either A or C must be within A or C and hence are not on the border. Therefore, the distance of an example to the border cannot be assessed directly: only the distance between the near-hit and the near-miss examples can be measured.

To further guide the search towards desirable near-miss examples, the BORDEAUX tool has an affordance for the user to specify which relations are permitted to differ. BORDEAUX uses Alloy* [65], which is an extension of Alloy Analyzer, to solve formulas with higher-order quantifiers.

The experiments in Section 4.5 compare BORDEAUX with Aluminum [78] and the Alloy Analyzer version 4.2. BORDEAUX does a better job of producing pairs of near-hit and near-miss examples that are close to each other, with some computational cost. In some cases the absolute minimization technique of Aluminum produces results similar to the relative minimization technique of BORDEAUX, but in other cases the results differ significantly.

Based on observations of the experiments, we design and implement two optimizations for BORDEAUX in Section 4.6: scope tightening and parallelization. The key observation is that, in practice, the near-hit and near-miss are usually very close to each other. The optimizations reduce the computational cost of BORDEAUX by over an order of magnitude.

In the next section, we review related works and discuss how BORDEAUX differs from similar tools. Section 4.1 sketches an illustrative example. In Section 4.2, we define the concepts and formulas for finding near-hit and near-miss examples, and discuss some other special cases of these formulas that might be interesting for users, listed as **Contribution 2**. Section 4.5 demonstrates the experimental evaluation of BORDEAUX and its comparison with the state-of-the-art Alloy analysis tools. Two approaches to optimize the prototype are described in Section 4.6. Section 4.8 concludes.

4.1 Illustrative Example

Consider a model that describes an undergraduate degree in computer engineering, as in Figure 4.1. In this illustrative model, a student must take two courses to graduate, and she must have taken all necessary prerequisites for each course.

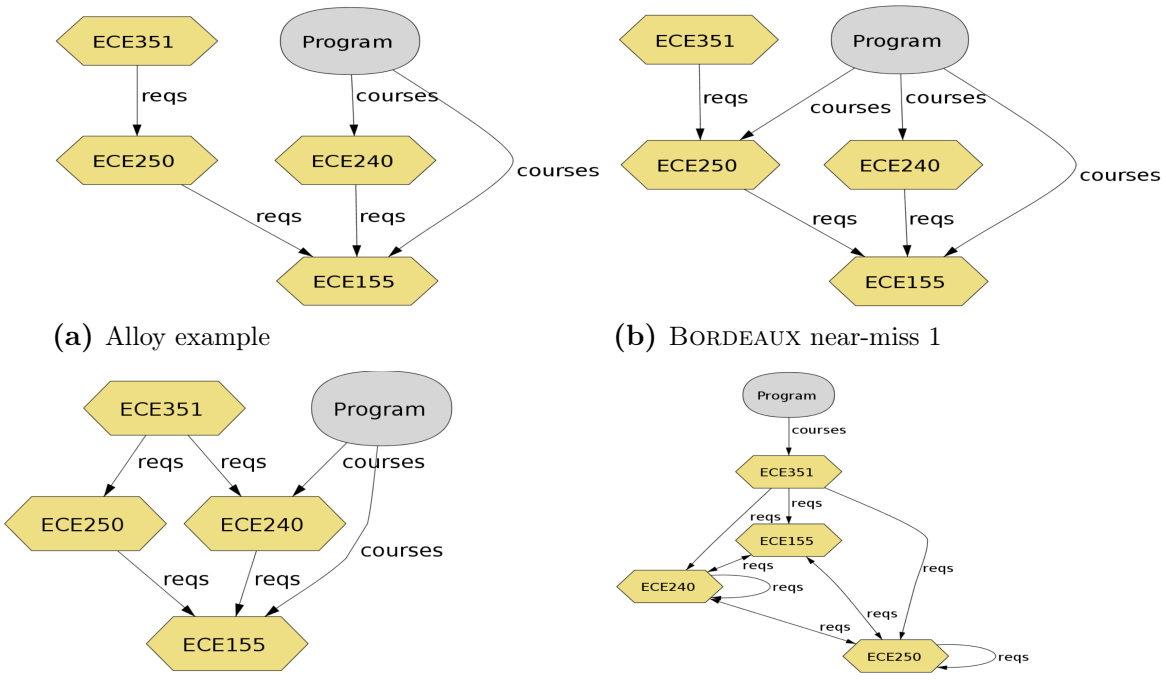
Figure 4.1 Model of requirements for undergraduate Computer Engineering degree

```
1 abstract sig Course{reqs: set Course}
2 one sig ECE155, ECE240, ECE250, ECE351 extends Course{}
3 one sig Program{courses: set Course}
4 pred prerequisites{ reqs = ECE240→ECE155 + ECE250→ECE155 + ECE351→ECE250 }
5 fun graduationPlan[]: Program{ {p: Program| eq[#p.courses, 2] and
6       all c: p.courses| some c.reqs implies c.reqs in p.courses} }
7 pred showSuccesfulPlan[]{  prerequisites and some graduationPlan }
8 run showSuccesfulPlan
```

One can ask Alloy Analyzer to generate an example consistent with the model, the analyzer generates an example similar to Figure 4.2a. Everything looks OK: this example corresponds with the user's intentions. But this model harbours a partial overconstraint bug: there are examples that the user intends, but which are not consistent with the model.

Bordeaux generates two near-miss examples (Figs. 4.2b & 4.2c). These are non-examples at a minimum distance from the example in Figure 4.2a, adding one tuple to

Figure 4.2 Examples revealing an overconstraint issue in the model of Figure 4.1



(a) Alloy example

(b) Bordeaux near-miss 1

(c) Bordeaux near-miss 2

(d) Alloy non-example

relations courses and reqs, respectively. The first near-miss example reveals the partial over-constraint: a student is prevented from graduating if they take an extra course. The user rectifies this by changing the equality predicate (eq[]) on Line 6 of Figure 4.1 to a less-than-or-equal-to (leq[]). The second near-miss example is not interesting to the user because it just involves a perturbation of the pre-requisites. Subsequent searches can be set to exclude the reqs relation.

Alloy can be used to generate an arbitrary non-example (*e.g.*, Fig. 4.2d) if the user manually negates the model. This unfocused non-example is unlikely to be meaningful for the user, as it might be too divergent from her intention.

4.2 Proximate Pair-Finder Formula

The formal definitions in this section constitute **Contribution 2** of this dissertation. As described above, one of the main challenges in formalizing near-miss and near-hit examples is formalizing the concept of *border*. This concept has not been formalized in any prior work that we are aware of in this area (*e.g.*, not in any of Seater [94], Mendel [64], or Nelson et al. [78]). As far as we see it, there is no way to explicitly formalize the border, since there are no examples on the border: the border is the space in between the near-misses and near-hits. Consequently, we do not explicitly define border, and we define near-miss and near-hit relatively to each other.

Our definitions begin with some basics, progress to the concept of distance, and then to near-miss and near-hit. Finally, we introduce the *Proximate Pair-Finder Formula*, which is a generalization of the idea of finding a near-miss/near-hit pair, and is the definition that our implementation embodies.

Definition 8 ***(Valuation).*** A valuation V of model M [Definition 1] is a sequence of sets of tuples, where each entry in the sequence corresponds to a relation in M, and is within M's bounds B. Let $\mathcal{V}$ name the set of all possible valuations of M.

The size (#) of a valuation is the number of tuples: $\#V \triangleq \sum_{i=1}^{|R|} |V_i|$

Definition 9 ***(Instance).*** An instance I of model M is a type-correct valuation of M, according to Alloy's type system [27]. Briefly, every atom contained in the instance will be in exactly one unary relation, and the columns of each non-unary relation will be restricted in terms of the unary relations.

Suppose that I and J are two instances of model M.

The difference of I and J ($I - J$) is a valuation of model M that, for each relation, contains the tuples from I that are not in J: $(I - J)_i \triangleq I_i - J_i$.

We say that J is a subset ($\subset$) of I if there is at least one relation for which J's tuples are a strict subset of I's tuples, and no relation for which I's tuples are not included in J's tuples; formally: $J \subset I \triangleq \wedge_{i=1}^{|R|}(J_i \subseteq I_i) \ \wedge \ \exists i | J_i \subset I_i$

Let $\mathcal{I}$ name the set of all instances of model M.

Definition 10 ***(Example).*** Instance I is an example of model M if I satisfies M's constraints C: *i.e.*, $C[I]$ is true.

Definition 11 ***(Non-example).*** Instance J is a non-example of model M if J does not satisfy M's constraints C: *i.e.*, $\neg C[J]$ is true.

Definition 12 ***(Distance).*** The distance m between two instances I and J is the number of tuples that must be added to one of the instances to make it equal to the other instance. This definition only applies when one of the instances is a subset of the other, or in the special case where they are already equals the distance is zero. This definition is symmetrical in that $D[m, I, J] = D[m, J, I]$. This definition is not transitive: due to the complex nature of instances, there is no linear ordering on them.

$$D[m,\ I,\ J] \triangleq (I \subset J \wedge m = \#(J - I)) \vee (J \subset I \wedge m = \#(I - J)) \vee (I = J \wedge m = 0)$$

In other words, this definition of distance is a restricted version of the well-known Levenshtein Edit Distance [71]. The Levenshtein distance allows three operations: addition, removal, and substitution. Our definition excludes substitution because it does not make sense in this context. Levenshtein distance is defined on strings, which are ordered. Our distance is defined over sets, which are not ordered. Levenshtein substitution is defined with respect to the ordering of the string: the character at position x has been changed. Since sets are not ordered, this concept of substitution does not make sense in this context. Our definition has an additional restriction. The usual definition of Levenshtein edit distance allows combinations of additions and removals. Our definition of distance requires that all of the operations be either additions or removals: *i.e.*, that one of the sets must be a subset of the other one.

For example, consider the strings *aa* and *aab*: the distance between them is 1; the addition of the *b*. Consider the strings *aa* and *abb*: the distance between them is not measurable with our definition, because neither is a subset of the other; the Levenshtein distance would be 2 (switching second *a* to *b* and adding a *b* at the end).

Definition 13 ***(Minimum distance).*** Whereas we define distance between two individual examples, we define *minimum distance* between two (disjoint) sets of examples. The minimum

distance between the two sets is the least distance between some example from the first set, and some examples from the second set. This definition is symmetrical in the same way that the definition of distance is: $D_\perp[m, E_1, E_2] = D_\perp[m, E_2, E_1]$. Also like the definition of distance, the definition of minimum distance is not transitive.

$$D_\perp[m,\ E_1,\ E_2] \triangleq E_1 \cap E_2 = \emptyset \ \wedge\ |E_1| > 0 \wedge |E_2| > 0 \ \wedge\ m > 0 \wedge \\ \exists e_1 : E_1,\ e_2 : E_2 \mid D[m,\ e_1,\ e_2] \wedge \\ (\forall e_1' : E_1,\ e_2' : E_2 \mid (\exists n : Int \mid D[n,\ e_1',\ e_2'] \implies m \leq n))$$

Consider an example with sets of strings. Suppose $E_1 = \{aa, bb\}$ and $E_2 = \{aab, b, aabb\}$. The minimum distance between these sets is one, which is the distance from aa to aab, and also the distance from bb to b.

Definition 14 ***(Near-hit example).*** We say that an example is a *near-hit* example if it is at the minimum possible distance to some near-miss example. Formally, predicate '$N_h[e_1,\ E,\ \mathcal{I}]$' is satisfied if e_1 is a near-hit example, E contains all examples of model M, and $\mathcal{I}$ includes all instances of model M.

$$N_h[e_1,\ E,\ \mathcal{I}] \triangleq |E| > 0 \ \wedge\ e_1 \in E \ \wedge\ E \subset \mathcal{I} \wedge \\ \exists\ e_2 : \mathcal{I} - E,\ m : Int \mid D[m,\ e_1,\ e_2] \ \wedge\ D_\perp[m,\ E,\ \mathcal{I} - E]$$

Consider the sets of strings example above, supposing that E here is equal to E_1 above, and that that $\mathcal{I}$ here equals $E_1 + E_2$ from above. We would say that aa and bb are both near-hit examples of E.

Definition 15 ***(Near-miss example).*** Similarly, a near-miss example is one that is at the minimum distance to a near-hit. Again, E is all examples, and $\mathcal{I}$ is all instances (of model M).

$$N_m[e_1,\ E,\ \mathcal{I}] \triangleq |E| > 0 \ \wedge\ e_1 \in I - E \ \wedge\ E \subset \mathcal{I} \wedge \\ \exists\ e_2 : E,\ m : Int \mid D[m,\ e_1,\ e_2] \ \wedge\ D_\perp[m,\ E,\ \mathcal{I} - E]$$

Consider the sets of strings example above, supposing that E here is equal to E_1 above,

and that that $\mathcal{I}$ here equals $E_1 + E_2$ from above. We would say that aab and b are both near-miss examples of E.

Definition 16 ***(Proximate Pair-Finder Formula).*** Since near-miss and near-hit need to be defined essentially with respect to each other, we introduce a definition of them together. Here e_1 and e_2 are individual examples drawn from (disjoint) sets E_1 and E_2, respectively. The examples e_1 and e_2 are a near-hit/near-miss pair when E_1 is all examples of model M and E_2 is all non-examples of model M. However, this definition is more general, since E_2 might be some other set of examples (that is disjoint from E_1). We discuss some other cases that might be interesting to users below.

$$PPFF[e_1,\ e_2,\ E_1,\ E_2] \triangleq e_1 \in E_1 \wedge e_2 \in E_2 \wedge E_1 \cap E_2 = \emptyset \\ \exists m : Int \mid D[m,\ e_1,\ e_2] \wedge D_{\perp}[m,\ E_1,\ E_2]$$

From the sets of strings example above, we would say that the pair $\langle aa, aab \rangle$ satisfies the PPFF. Similarly, the pair $\langle bb, b \rangle$ also satisfies the PPFF.

This PPFF definition forms the basis of our implementation. We next show, in several steps, how this definition is the guiding principle for our Alloy code generator.

4.3 Encoding the PPFF for Alloy*

The core of BORDEAUX generates variants of the PPFF [Definition 16], which it gives to Alloy* to solve. The input to the PPFF generation is two mutually inconsistent sets of constraints, C_1 and C_2, over the same set of relations, R. For generating these variants, BORDEAUX encodes modifications of Definition 16 in Alloy. Figure 4.3 shows the variants of PPFF definition that embody the minimum distance predicate, mutually inconsistent constraints, and distance predicate.

Figure 4.3 Proximate Pair-Finder Formula (PPFF) Variants

$$PPFF[e_1,\ e_2,\ E_1,\ E_2] \triangleq e_1 \in E_1 \wedge e_2 \in E_2 \wedge E_1 \cap E_2 = \emptyset \wedge$$
$$\exists m : Int \mid D[m,\ e_1,\ e_2] \wedge (\forall e_1' : E_1,\ e_2' : E_2 \mid \exists n : Int \mid D[n,\ e_1',\ e_2'] \implies m \leq n)$$

(a) A variant of PPFF [Definition 16] after embedding minimum distance predicate [Definition 13]

$$PPFF[E_1, E_2] \triangleq E_1 \cap E_2 \neq \emptyset \wedge \exists e_1 : E_1, e_2 : E_2,\ m : Int \mid D[m,\ , e_1,\ e_2] \wedge$$
$$\forall e_1' : \mathcal{I},\ e_2' : \mathcal{I} \mid \exists n : Int \mid D[n,\ e_1,\ e_2] \wedge C_1[e_1'] \wedge C_2[e_2']) \Rightarrow m \leq n)$$

(b) A variant of PPFF after changing parameters e_1, e_2 to existentially quantified variables.

$$PPFF[] \triangleq \exists e_1, e_2 : \mathcal{I},\ m : Int \mid C_1[e_1] \wedge C_2[e_2] \wedge D[m,\ , e_1,\ e_2] \wedge$$
$$\forall e_1' : \mathcal{I},\ e_2' : \mathcal{I} \mid \exists n : Int \mid D[n,\ e_1,\ e_2] \wedge C_1[e_1'] \wedge C_2[e_2']) \Rightarrow m \leq n)$$

(c) A variant of PPFF after changing parameters E_1, E_2, which are sets of examples, to externally defined predicates C_1, C_2. The predicates C_1, C_2 return true for all examples in E_1, E_2, respectively, and false for all others.

$$PPFF[] \triangleq \exists e_1, e_2 : \mathcal{I} \mid C_1[e_1] \wedge C_2[e_2] \wedge$$
$$\exists v : \mathcal{V} \mid (v = e_2 - e_1 \wedge e_1 \subset e_2) \wedge$$
$$\forall e_1' : \mathcal{I},\ e_2' : \mathcal{I},\ w : \mathcal{V} \mid$$
$$(C_1[e_1'] \wedge C_2[e_2'] \wedge w = e_2' - e_1' \wedge e_1' \subset e_2') \Rightarrow \#v \leq \#w$$

(d) A variant of PPFF after embedding the distance predicate [Definition 12]

Figure 4.3d demonstrates a definition of the PPFF that we have implemented in the core of BORDEAUX. A solution to the PPFF [Figure 4.3d] is a pair of instances, one of which (e_1) satisfies C_1, and the other of which (e_2) satisfies C_2. The key property of these two examples is that they are a minimum distance to each other. In the special case where C_2 is the negation of C_1, which the narrative of this section focuses on, then e_1 is a *near-hit* example of C_1 and e_2 is a *near-miss* example of C_1. For the sake of simplicity, this variant only considers distance from an instance to another instance by adding tuples. Hereafter, any reference to the PPFF without explicit qualification should be understood

as the formula in Figure 4.3d.

The PPFF is expressed with a top-level existential quantifier. Once the quantifier is skolemized by the Alloy Analyzer, the quantified variables contain a pair of instances e_1 and e_2. This PPFF contains two higher-order quantifiers: they are higher-order because they quantify over valuations of relations. The formula effectively says that there is no other pair of examples that are closer to each other than are e_1 and e_2. Valuation v in the PPFF is the difference $e_2 - e_1$. Valuation w in the PPFF is the difference $e'_2 - e'_1$. The relative minimization condition is that the size of w is not smaller than the size of v: $\#v \leq \#w$.

In the degenerate case where C_1 and C_2 are not mutually inconsistent, then the PPFF will always return $e_1 = e_2$, because any arbitrary example is at distance zero to itself. The PPFF is not designed to be meaningful when the constraints are not mutually inconsistent. The examples e_1 and e_2 are not necessarily absolutely minimal with respect to C_1 and C_2, respectively. That is, there might be smaller examples that satisfy C_1 and C_2. These two examples are *relatively* minimal with respect to each other: that is, the distance between them is small.

Alloy* supports higher-order quantifiers: *i.e.*, quantifiers over relations, which is required to solve PPFF. The user's model must be written in regular Alloy, with no higher-order quantifiers. Bordeaux transforms the user's Alloy model into an Alloy* model and adds a variant of the PPFF synthesized for the user's desired search. Bordeaux then transforms the Alloy* solution back into the terms of the user's original model.

While the Alloy* language is syntactically a superset of the regular Alloy language, so the user's model is a legal Alloy* model, simply taking the user's model as-is will not work for the PPFF. The transformation to prepare for solving the PPFF must bundle up all of the constraints of the original model (fact blocks, multiplicity constraints, *etc.*) into a single predicate. The rest of this section describes statements that Bordeaux generates for specifying *Instance*, *Distance*, and finally the PPFF in Alloy. We show Alloy templates for these statements and illustrate them with formulas that Bordeaux synthesizes for the example in Figure 4.1.

Instance. To encode the set of instances in the formula, *i.e.*, $\mathcal{I}$, we define a predicate that distinguishes instances of a model from other valuations. The input parameters of instance, as we call it, are all relations of the model. For a model having m unary relations and n non-unary relations, Figure 4.4 demonstrates three alternatives of what the predicate instance can look like. All alternatives check whether atoms in tuples of non-unary parameters already exist in the atoms of the corresponding unary parameters.

Figure 4.4 BORDEAUX generates the predicate instance to distinguish self-consistent valuations from the others. Three different approaches to check the inclusion of tuples of non-unary parameters are in tuples of the constituent unary relations.

```
1  pred instance[s1: S1, ..., sm: Sm,
2                r1: Si→...→Sj,...,
3                rn: Sk→...→Sl]{
4   // Join r1 to univ /r1/−1 times,
5   // where /r1/ is the arity of r1.
6   ((((r1.univ).univ)....).univ) in si
7    ...
8     (univ.(....(univ.(univ.r1)))) in sj
9   ...
10  ((((rn.univ).univ)....).univ) in sk
11   ...
12    (univ.(....(univ.(univ.rn)))) in sl
13 }
```

(a) Using *join* operators to check self-consistency of tuples passed in parameters

```
1  pred instance[s1: S1, ..., sm: Sm,
2                r1: Si→...→Sj,...,
3                rn: Sk→...→Sl]{
4    r1 in si → ... →sj
5    ...
6    rn in sk → ... →sl
7  }
```

(b) Using *product* operators for checking self-consistency of tuples in parameters

```
1  pred instance[s1: S1, ..., sm: Sm,
2                r1: Si→...→Sj,...,
3                rn: Sk→...→Sl]{
4   ((((r1.univ).univ)....).univ) in si
5   all x: si| x.r1 in si+1 → ... →sj
6   ...
7   ((((rn.univ).univ)....).univ) in sk
8   all x: sk| x.rn in sk+1 → ... →sl
9  }
```

(c) Checking self-consistency of tuples in parameters using join and product operators

To compare these alternatives, we synthesized all three and simulated them with a n-ary relation. We set n from two to ten and recorded the size of SAT-formula and the simulation time of each alternative. Measured in Table 4.1, the alternative that uses join

Table 4.1 Comparing use of (J)oin *vs.* (P)roduct *vs.* (C)ombination of join and product operators to implement instance operator explained in Figure 4.4. We simulated each alternative for relations with different arities, two to ten, in Alloy Analyzer and recorded the size of SAT formula and simulation time.

	SAT-Variables			SAT-Clauses			Trasnlation Time			Execution Time			Total Time		
Arity	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**	**J**	**P**	**C**
2	162	149	162	238	219	238	11	5	4	1	1	0	12	6	4
3	418	362	393	647	582	625	10	5	4	1	1	1	11	6	5
4	1,191	1,014	1,099	1,892	1,703	1,800	8	7	10	1	0	1	9	7	11
5	3,475	2,922	3,169	5,565	5,000	5,259	24	12	14	1	1	1	25	13	15
6	10,295	8,610	9,343	16,528	14,831	15,576	61	37	36	4	3	4	65	40	40
7	30,723	25,638	27,829	49,361	44,264	46,467	146	111	121	18	13	9	164	124	130
8	91,975	76,686	83,251	147,804	132,503	139,080	462	307	273	41	30	23	503	337	296
9	275,699	229,794	249,481	443,077	397,160	416,859	1,432	1,094	1,163	96	97	78	1,528	1,191	1,241
10	826,839	689,082	748,135	1,328,840	1,191,071	1,250,136	9,125	6,905	7,112	333	294	237	9,458	7,199	7,349

Figure 4.5 Sample of instance predicate for the model in Figure 4.1

```
1 pred instance [_reqs: Course→Course, _ece155: set ECE155, _ece240: set ECE240,
2                _ece250: set ECE250, _ece351: set ECE351, _program: set Program,
3                _courses: Program→Course] {
4   (all v1: one (_ece155 + _ece240 + _ece250 + _ece351) |
5     (v1._reqs in (_ece155 + _ece240 + _ece250 + _ece351))
6   (_reqs.univ) in (_ece155 + _ece240 + _ece250 + _ece351)
7   (all v1: one _program |
8     (v1._courses in (_ece155 + _ece240 + _ece250 + _ece351))
9   (_courses.univ in _program)
10 }
```

operators, Figure 4.4a, always needs more resources and takes longer. The other two alternatives perform very similarly. As Alloy Analyzer uses the combination alternative, Figures 4.4a and 4.4b, for translating theses kind of constraints from an Alloy model to a Kodkod model, we have chosen it in our implementation.

For illusteration, BORDEAUX generates the predicate in Figure 4.5 for the model in Figure 4.1. The parameters of the predicate are the model's relations. Since abstract unary relations, such as Course, do not have concrete valuations, BORDEAUX does not consider them for checking self-consistency.

Distance. As the valuations are encoded by the quantifier variables, we defined a predicate for each relation to find a set of tuples that should be added to the first relation valuation in order to get the second one. To do so, the predicates, prefixed by delta_, are synthesized for each relation. Figure 4.6 demonstrates the base definition of the predicate.

The delta predicate is synthesized for all relations that the user expects to see changes in their tuples. The parameters of the predicate respectively contain tuples for one relation of a near-hit example, near-miss example, and difference between them. Figure 4.7 shows the synthesized predicate for reqs relation in the course model. A summation of the size of the third parameter of the delta_ predicates is the distance between two instance.

PPFF. Figure 4.9 demonstrates a form of PPFF in Alloy. A run of a variant of this query with Alloy* skolemizes near-hit and near-miss examples. The quantifier spanned

Figure 4.6 Typical form of predicate delta_. BORDEAUX generates a predicate delta_ for each relation ri.

```
1 pred delta_r_i[r,r′,r″: S_j→...→S_k]{ r != r′ implies r″=r′−r and r′=r+r″ else no r″ }
```

Figure 4.7 Sample of delta_ predicate for encoding distance w.r.t. relation reqs in Figure 4.1

```
1 pred delta_reqs[_reqs: Course→Course, _reqs': Course→Course, _reqs'': Course→Course] {
2   _reqs != _reqs' implies (_reqs'' = _reqs' − _reqs and _reqs' = _reqs'' + _reqs) else no _reqs''
3 }
```

over Lines 9-16 in Figure 4.9 ensures that the quantified variables are skolemized to two instances with a minimum distance by adding tuples. C1 and C2 encompass two inconsistent constraints. All the referenced relations in the body of these constraints are replaced with the formal parameters. In the context of programming languages, this step is similar to refactoring a piece of code to replace global variables with local ones.

The distance that is encoded in quantified variables at Lines 6 in Figure 4.9 has to be minimum. The synthesized delta_ predicates ensure that these variables hold the tuples of difference between corresponding variables for near-hit and near-miss examples. The size of distance is the summation of the cardinality of quantified variables representing distances,

Figure 4.8 Sample of synthesized PPFF for Figure 4.1

```
1 pred PPFF_for_Courses_model[] {
2   some _reqs, _reqs', _reqs'': set Course→Course, _ece155, _ece155', _ece155'': set ECE155,
3        _ece240, _ece240', _ece240'': set ECE240, _ece250, _ece250', _ece250'': set ECE250, _ece351,
4        _ece351', _ece351'': set ECE351, _program, _program', _program'': set Program,
5        _courses, _courses', _courses'': set Program→Course | {
6        {instance[_reqs, _ece155, _ece240, _ece250, _ece351, _program, _courses]
7         instance[_reqs', _ece155', _ece240', _ece250', _ece351', _program', _courses']
8         delta_reqs[_reqs, _reqs', _reqs''] and deltaECE155[_ece155, _ece155', _ece155'']
9        deltaECE240[_ece240, _ece240', _ece240''] and deltaECE250[_ece250, _ece250', _ece250'']
10       deltaECE351[_ece351, _ece351', _ece351''] and deltaProgram[_program, _program', _program'']
11       delta_courses[_courses, _courses', _courses'']}
12     and
13       all _reqso, _reqso', _reqso'': set Course→Course, _ece155o, _ece155o', _ece155o'': set ECE155,
14            _ece240o, _ece240o', _ece240o'': set ECE240, _ece250o, _ece250o', _ece250o'': set ECE250,
15            _ece351o, _ece351o', _ece351o'': set ECE351, _programo, _programo', _programo'': set Program,
16            _courseso, _courseso', _courseso'': set Program→Course | {
17            {instance[_reqso, _ece155o, _ece240o, _ece250o, _ece351o, _programo, _courseso]
18            instance[_reqso', _ece155o', _ece240o', _ece250o', _ece351o', _programo', _courseso']
19            delta_reqs[_reqso, _reqso', _reqso''] and deltaECE155[_ece155o, _ece155o', _ece155o'']
20            deltaECE240[_ece240o, _ece240o', _ece240o''] and deltaECE250[_ece250o, _ece250o', _ece250o'']
21            deltaECE351[_ece351o, _ece351o', _ece351o''] and deltaProgram[_programo, _programo', _programo'']
22            delta_courses[_courseso, _courseso', _courseso'']}
23          implies
24            leq[sigma[#_reqs'', #_ece155'', #_ece240'', #_ece250'', #_ece351'', #_program'', #_courses''],
25                sigma[#_reqso'', #_ece155o'', #_ece240o'', #_ece250o'', #_ece351o'', #_programo'', #_courseso'']]
26       }
27   }
```

Figure 4.9 Encoding PPFF in Alloy. Given an Alloy model, including relations and two inconsistent constraints, the predicate finds two near border examples consistent with each constraint. Skolemization of the other existential quantifier represents both examples.

```
1  pred PPFF{
2   some s1: S1, ..., sm: Sm, r1: Si→...→Sj, rn: Sk→...→Sl,
3        s'1: S1, ..., s'm: Sm, r'1: Si→...→Sj, r'n: Sk→...→Sl|
4         instance[s1, ..., sm, r1, ..., rn] and instance[s'1, ..., s'm, r'1, ..., r'n] and
5         C1[s1, ..., sm, r1, ..., rn] and C2[s'1, ..., s'm, r'1, ..., r'n] and
6         some s''1: S1, ..., s''m: Sm, r''1: Si→...→Sj, r''n: Sk→...→Sl|
7          delta_S1[s1, s'1, s''1] and ... and delta_Sm[sm, s'm, s''m] and
8          delta_r1[r1, r'1, r''1] and ... and delta_rn[rn, r'n, r''n] and
9          all so1: S1, ..., som: Sm, ro1: Si→...→Sj, ron: Sk→...→Sl,
10             so'1: S1, ..., so'm: Sm, ro'1: Si→...→Sj, ro'n: Sk→...→Sl,
11             so''1: S1, ..., so''m: Sm, ro''1: Si→...→Sj, ro''n: Sk→...→Sl|
12             (instance[so1, ..., som, r1, ..., ron] and instance[so'1, ..., so'm, ro'1, ..., ro'n] and
13              C1[so1, ..., som, ro1, ..., ron] and C2[so'1, ..., so'm, ro'1, ..., ro'n] and
14              delta_S1[so1, so'1, so''1] and ... and delta_Sm[som, so'm, so''m] and
15              delta_r1[ro1, ro'1, ro''1] and ... and delta_rn[ron, ro'n, ro''n]) implies
16               leq[sigma[s''1, ..., s''m, r''1, ..., r''n], sigma[so''1, ..., so''m, ro''1, ..., ro''n]]
17 }
```

Line 16 in Figure 4.9. sigma is an additional synthesized Alloy function returning the summation of sizes.

Figure 4.8 shows a sample of synthesized Alloy model for the course example. Depending on the body of C1 and C2, a run on this formula with Alloy* would return a pair of instances and their distance.

4.4 Implementation: Extending Alloy

Based on PPFF, we have developed a prototype tool, Bordeaux that assists Alloy users to browse examples and non-examples of an Alloy model. While the user can already browse examples of an Alloy model, she can now also browse the model's near-miss examples. She can continue to browse other non-examples or ask for a near-hit example. In addition to the interaction with the user through the user interface, Bordeaux provides APIs to be called from another program, such as Margaux.

In the following, we explain how a user can interact with Bordeaux and what functionalities the prototype provides for the user and other programs in the form of APIs.

4.4.1 Special Cases of Potential User Interest

The user might be interested in some of the following special cases, which can all be easily accommodated by generating the PPFF with specific settings for C_1 and C_2 (some of these are not yet implemented in the current prototype [69]):

1. **Find a near-miss example and a near-hit example:** Set C_2 to be the negation of C_1 (as discussed above).
2. **Find a near-miss example close to an example:** Set C_1 to be a predicate that defines the example, and set C_2 to be the negation of the model's constraints.
3. **Find a near-hit example close to a non-example:** Set C_1 to be a predicate that defines the non-example, and set C_2 to be the model's constraints.
4. **Restrict the difference between the examples to certain relations:** The difference operation can easily be generated over a user-specified subset of the relations, rather than all of them.
5. **Smaller near-miss examples:** In PPFF, e_2 is bigger than e_1. If C_2 is the negation of the model's constraints, this will result in a near-miss example that is larger than the near-hit. To get a smaller near-miss example, simply set C_1 to be the negation of the model's constraints, and C_2 to be the model's constraints.
6. **Find a near-miss example for an inconsistent model:** If the original model is inconsistent, then it has no examples. A workaround for this situation is to set C_1 to be an empty example (no tuples), and set C_2 to be the negation of the model.

4.4.2 Interacting with Margaux

Bordeaux also assists Margaux, explained in Chapter 5, to produce focused near-hit and near-miss examples. Margaux needs to find a near-hit or near-miss example for two mutually inconsistent constraints. To do so, Margaux can interact with Bordeaux through an interface with two APIs:

1. **'`String Find_Near-hit (String C1, String C2, String M)`':** Given two constraints, Bordeaux returns an example consistent with `C2` and inconsistent with `C2`. From the implementation perspective, the constraints are enclosed in two Alloy **pred** blocks. Bordeaux tailors PPFF to refer to such constraints as $C1$ and $C2$. The parameters for

selecting relations and adding/removing tuples are passed implicitly. The return is an example encoded in the form of existential quantifier.

2. '`String Find_Near-miss (String C1, String C2, String M)`': BORDEAUX returns a near-miss example consistent with the second constraints.

4.5 Experiments

To study the idea of browsing near-hit and near-miss examples, we have developed BORDEAUX, a prototype that extends Alloy Analyzer. This study includes the experiments carried out to compare BORDEAUX with other tools. From this study, we also show paths that optimize the performance of BORDEAUX in finding near-miss examples. In this section, we explore the experiments revealing the position of BORDEAUX among other similar tools. The next section discusses our ideas to optimize the prototype.

Given an example, BORDEAUX can find a near-miss example. Users can browse more near-miss examples or ask for a near-hit example. To support this way of browsing, BORDEAUX performs a relative minimization; namely, minimizing a distance between an example and a non-example. Although users cannot browse near-hit and near-miss examples with Alloy Analyzer, they can manually modify models to produce examples and non-examples. Using Aluminum, the users can find minimal examples, and if they manually negate the model, they can browse minimal non-examples, too. Aluminum's concept of a minimal example, which we call *absolute* minimal, is an example with the smallest number of tuples.

The experiment includes five models that are shown in Table 4.2. We have used an Intel i7-2600K CPU at 3.40GHz with 16GB memory. All experiments are done with MiniSat. In what follows, we explain the experiments and discuss their contribution to answer the following research questions:

RQ 4.1. *What is the extra cost for producing relative minimal non-examples?*

RQ 4.2. *How many near-miss examples can* BORDEAUX *find in one minute?*

RQ 4.3. *How far are arbitrary non-examples from the near-miss?*

RQ 4.4. *How far are absolute minimum non-examples from the near-miss?*

To study the extra cost for finding near-miss examples with BORDEAUX, we used Alloy Analyzer to find arbitrary examples and non-examples and compared their costs to using BORDEAUX to find near-hit/near-miss example pairs (Table 4.2). To find the non-examples, we manually negated the studied models, *i.e.*, if C is a model's constraint, then

Table 4.2 Comparing BORDEAUX (B) and Alloy Analyzer (A) to find non-examples

			# SAT Variables			# SAT Clauses			Translation Time(ms)			Execution Time(ms)		
	Number of Relations	Size of Example	B	A	B/A	B	A	B/A	B	A	B/A	B	A	B/A
Singly-linked List	2	1	846	492	*1.72*	2,518	757	*3.33*	15	26	*0.58*	29	20	*1.45*
Doubly-linked List	3	7	20,531	1,909	*10.75*	56,358	4,580	*12.31*	39,700	141	*281.56*	121,664	111	*1,096.07*
Binary Tree	3	1	1,088	710	*1.53*	3,295	1,440	*2.29*	12	438	*0.03*	44	166	*0.27*
Graduation Plan	5	8	5,934	734	*8.08*	17,846	1,276	*13.99*	381	336	*1.13*	439	74	*5.93*
File System	10	8	8,154	2,605	*3.13*	27,672	4,690	*5.90*	3,883	571	*6.80*	13,366	308	*43.40*

$\neg C$ gives the negation of the model. In these experiments, for BORDEAUX, we set C_1 to be equal to the arbitrary example returned by Alloy.

In Table 4.2, it can be seen that BORDEAUX does not incur much additional cost for small models, but once the model gets larger the costs get significant (RQ 4.1). The small Binary Tree model is an exception where BORDEAUX appears to run faster than the stock Alloy Analyzer. Occasional anomalies such as this are common with technology based on SAT solvers.

For answering RQ 4.2, we have done another experiment to count the number of distinct near-miss examples that BORDEAUX generates in one minute. The results show how the prototype's performance degrades for the Alloy models with more relations or larger formula size. Given examples, BORDEAUX produces *27*, *4*, *31*, *15*, and *9* distinct near-miss examples respectively for Singly-linked List, Doubly-linked List, Binary Tree, Graduation Plan, and File System models in one minute. The performance descends because BORDEAUX reformulates and resolves the model per each distinct example and non-example. BORDEAUX returns more near-miss examples for Singly-linked List and Binary Tree models, as the given examples of both models are fairly simpler than the others. Therefore, the near-miss examples will have relatively fewer tuples. That is, smaller near-miss examples lead to smaller and relatively simpler formulas for excluding redundant near-miss examples.

To answer RQ 4.3 and RQ 4.4, we have performed another experiment to demonstrate how near-miss examples that BORDEAUX systematically produces differ from non-examples that other tools produce from manually modified models. To do so, using various sizes of examples of different models, we evaluated their distances to non-examples that each instance-finder produces. We have selected Alloy Analyzer and Aluminum for comparing with BORDEAUX. Although Alloy Analyzer and Aluminum do not provide capabilities for browsing non-examples, we have manually transformed the models and synthesized required statements.

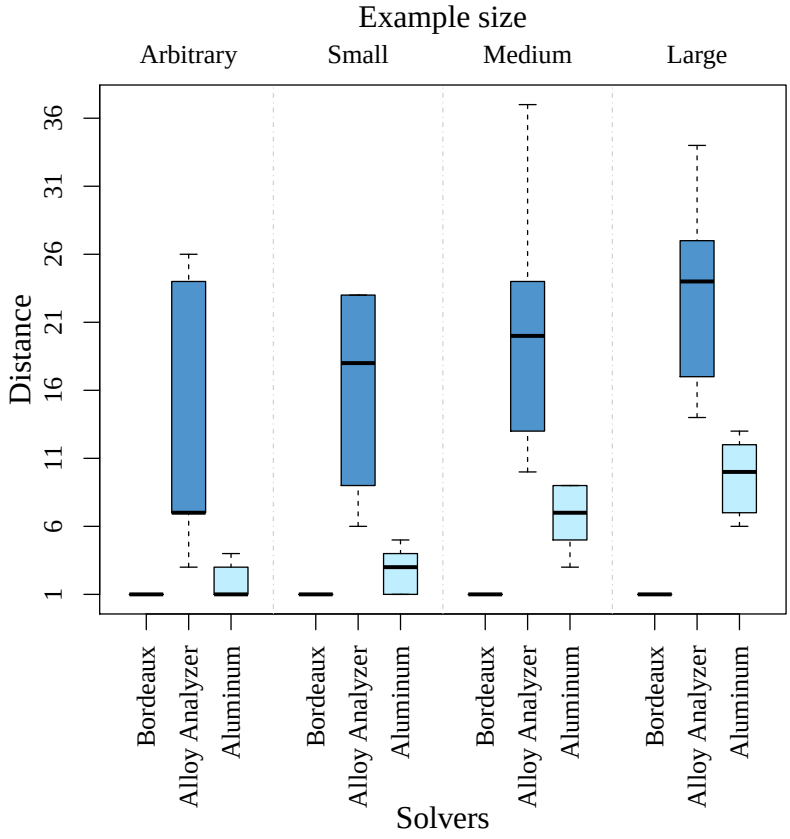
Table 4.3 Comparing the size of the first example and the first non-example generated by Bordeaux (B), Alloy Analyzer (Ay), and Aluminum (Am). The size of an example is its number tuples. Distance between the example and non-example is the number of tuples that are added to or removed from the example to make it identical to the non-example. Column 'Mean' shows the average of distances for the studied models per different sizes of examples.

		Size of Example			Size of Non-example			Added			Removed			Distance			Mean		
		B	Ay	Am	B	Ay	Am	B	Ay	Am	B	Ay	Am	B	Ay	Am	B	Ay	Am
Arbitrary	Singly-linked List	1	1	1	2	8	2	1	7	1	0	0	0	1	7	1			
	Doubly-linked List	7	0	0	8	26	2	1	26	2	0	0	0	1	26	2			
	Binary Tree	1	1	1	2	8	4	1	7	3	0	0	0	1	7	3	1	13.4	2.2
	Graduation Plan	8	8	8	9	12	4	1	3	0	0	0	4	1	3	4			
	File System	8	2	1	9	26	2	1	24	1	0	0	0	1	24	1			
Small	Singly-linked List	3	3	3	4	19	2	1	17	0	0	1	1	1	18	1			
	Doubly-linked List	4	4	4	5	17	2	1	20	1	0	3	3	1	23	4			
	Binary Tree	5	5	5	6	5	4	1	3	2	0	3	3	1	6	5	1	15.8	2.4
	Graduation Plan	5	5	5	6	9	4	1	7	0	0	2	3	1	9	1			
	File System	2	3	1	3	22	2	1	21	1	0	2	0	1	23	1			
Medium	Singly-linked List	5	5	5	6	19	2	1	17	0	0	3	3	1	20	3			
	Doubly-linked List	7	7	7	8	21	2	1	19	1	0	5	6	1	24	7			
	Binary Tree	9	9	9	10	5	4	1	3	2	0	7	7	1	10	9	1	20.8	6.4
	Graduation Plan	8	8	8	9	13	4	1	9	0	0	4	5	1	13	4			
	File System	9	9	9	10	34	2	1	31	1	0	6	8	1	37	9			
Large	Singly-linked List	9	9	9	10	19	2	1	17	0	0	7	7	1	24	7			
	Doubly-linked List	10	10	10	11	21	2	1	19	1	0	8	9	1	27	10			
	Binary Tree	13	13	13	14	5	4	1	3	2	0	11	11	1	14	13	1	23.2	9.6
	Graduation Plan	10	10	10	11	15	4	1	11	0	0	6	6	1	17	6			
	File System	12	12	12	13	34	2	1	28	1	0	6	11	1	34	12			

For comparing relative minimal, absolute minimal, and arbitrary non-examples, we have used the aforementioned tools to find non-examples given arbitrary, small, medium, and large size examples. In the case of arbitrary examples, each tool finds a pair of example and non-example without any extra constraints on the size of examples. With restricted-size examples, all the tools have to first generate the same size examples, then generate non-examples for them. Depending on the models, the size of the examples varies from *two* to *five* tuples in small size examples and *nine* to *thirteen* tuples for the large size examples. In Table 4.3, we have recorded the size of examples and non-examples that each tool produces, as well as the number of tuples that should be added or removed from an example to make an example identical with its paired non-example.

As Table 4.3 shows, Aluminum generates absolute minimal examples and non-examples once the example size is arbitrary. It also always produces minimal non-examples regardless

Figure 4.10 Comparing BORDEAUX, Alloy Analyzer, and Aluminum with respect to the number of tuples that differ between an example and a non-example.



of the size of given examples. Alloy Analyzer generates arbitrary examples close to absolute minimal size, but the sizes of non-examples do not follow any particular pattern. Although BORDEAUX produces examples in arbitrary sizes, it produces non-examples with one more tuple in all the models.

Depicted in Figure 4.10, BORDEAUX produces a non-example in a minimum distance from a given example. Answering RQ 4.3, Alloy Analyzer behaves arbitrarily to produce non-examples close to the examples. The distances from examples to non-examples increase for larger examples. Answering RQ 4.4, for arbitrary and small examples, Aluminum produces non-examples that are fairly close to the examples. Given medium and large examples, Aluminum finds non-examples with larger distances from the given examples. Although the distances between examples and non-examples, generated by Aluminum, do

not fluctuate like the distances between examples and non-examples produced by Alloy Analyzer, they show relative minimum distance similar to those found by BORDEAUX.

Moreover, finding a non-example by negating the model provides no direction for adding or removing tuples. Although we expected to see a near-miss example with extra tuples, as generated by BORDEAUX, Aluminum produced a non-example with fewer tuples for the Singly Linked-list model. Unlike BORDEAUX, Alloy Analyzer and Aluminum do not directly produce non-examples of a model. Simulating a model's negation does not necessarily cause that Alloy Analyzer and Aluminum produce non-examples in a minimum distance from given examples of the studied models.

4.6 Optimization

By reviewing the experiment resultsin Table 4.3, we have observed a trend in distances between examples and non-examples returned by BORDEAUX. In the studied models, with the addition of a single tuple, all examples and non-examples become identical. In the other words, the examples are already near-hit examples, and they can be pushed to be non-examples with the minimum number of changes, *i.e.*, a single tuple. This observation assists us to select tighter scopes and parallelize searches for examples and non-examples. Without choosing tight scopes, the analysis becomes infeasible. Using parallelization, the time to find near-miss examples improves to *2.2* seconds on average from several minutes without parallelization.

4.6.1 Selecting Tighter Scopes

If most examples are near-hit examples, as the case studies show, BORDEAUX can approximate the scope of each unary relation to be one more than the number of its tuples in the example when Alloy* is used for the underlying solver. As depicted in Table 4.4, we have rerun our experimental models by selecting scopes of *one* (+1), *two* (+2), and *three* (+3) more than the number of tuples of the example for each unary relation in the models. Note that these scopes limit the number of tuples only for unary relations. Non-unary relations still can have any tuples in difference between an example and a non-example.

When the scopes of unary relations increase by *one*, BORDEAUX can find a near-miss example for a studied model within *7.5* minutes on average. Provided the scopes increase by two, the time to find a near-miss example is inflated by the ratio of *8.43* on average. If the scopes increase by three, the time to find a near-miss example is *fifteen* times longer

Table 4.4 Showing how selecting different scopes affects the cost of analysis performed by Alloy*. The notations '*+1*', '*+2*', and '*+3*' show the records when the scopes of all unary relations in the studied models are set to one, two, and three more tuples than the number of tuples in the same relations of examples. The columns with '+1' in their headers contain the actual records. The other columns contain the increase ratios.

		SAT Variables			SAT Clauses			Translation Time(ms)			Execution Time(ms)			Total Time(ms)		
		+1	+2/+1	+3/+2	+1	+2/+1	+3/+2	+1	+2/+1	+3/+2	+1	+2/+1	+3/+2	+1	+2/+1	+3/+2
Arbitrary	Singly-linked List	846	1.8842	1.5558	2,518	1.9682	1.5672	15	1.0667	1.2500	29	0.828	1.458	44	0.9091	1.3750
	Doubly-linked List	20,531	2.4871	T/O	56,358	2.7368	T/O	39,700	11.8532	T/O	121,664	1.172	T/O	161,364	3.7998	T/O
	Binary Tree	1,088	1.9743	1.6294	3,295	2.0859	1.6770	12	1.1667	1.2857	4	3.500	1.429	16	1.7500	1.3571
	Graduation Plan	5,934	1.9821	1.4840	17,846	2.0284	1.5026	381	7.4436	1.7585	439	3.146	1.547	820	5.1427	1.6891
	File System	8,154	2.1634	T/O	27,672	2.2664	T/O	3,883	12.4074	T/O	13,366	15.239	T/O	17,249	14.6013	T/O
Small	Singly-linked List	1,862	1.5397	1.6993	5,858	1.5683	1.7648	12	1.4167	1.4118	19	0.947	2.222	31	1.1290	1.8286
	Doubly-linked List	3,204	1.8121	2.0541	10,922	1.9736	2.2635	31	1.7742	2.8364	174	0.632	2.036	205	0.8049	2.3030
	Binary Tree	5,371	2.0646	2.2615	17,404	2.1485	2.3097	105	5.5429	7.3849	141	13.652	3.484	246	10.1911	4.3893
	Graduation Plan	4,342	1.7725	1.5654	13,244	1.7882	1.5503	292	7.7671	1.7637	503	5.865	2.271	795	6.5635	2.0502
	File System	2,890	1.8218	1.5470	8,910	1.9274	1.5852	60	1.0500	1.2540	138	3.732	1.450	198	2.9192	1.4291
Medium	Singly-linked List	3,537	1.8476	1.3770	10,887	1.9555	1.4716	66	4.3030	0.7007	49	3.490	2.146	115	3.9565	1.2440
	Doubly-linked List	6,175	1.9869	2.1946	20,109	2.1358	2.3287	178	5.8427	7.6663	388	1.284	57.504	566	2.7173	23.8036
	Binary Tree	13,153	2.4004	T/O	44,972	2.4702	T/O	20,804	14.7966	T/O	9,873	11.299	T/O	30,677	13.6711	T/O
	Graduation Plan	4,862	1.8375	1.5640	14,838	1.8684	1.5456	291	8.0584	1.7693	155	23.587	0.963	446	13.4552	1.2781
	File System	6,946	2.2469	T/O	21,008	2.2076	T/O	12,256	15.8204	T/O	13,174	59.415	T/O	25,430	38.4043	T/O
Large	Singly-linked List	45,668	T/O	T/O	91,495	T/O	T/O	1,040,121	T/O	T/O	260	T/O	T/O	1,040,381	T/O	T/O
	Doubly-linked List	20,549	2.4846	T/O	56,436	2.7332	T/O	35,275	14.9564	T/O	163	1.227	T/O	35,438	14.8932	T/O
	Binary Tree	24,149	T/O	T/O	79,127	T/O	T/O	1,860,944	T/O	T/O	924,176	T/O	T/O	2,785,120	T/O	T/O
	Graduation Plan	16,528	T/O	T/O	34,806	T/O	T/O	7,814	T/O	T/O	35	T/O	T/O	7,849	T/O	T/O
	File System	14,215	T/O	T/O	41,246	T/O	T/O	2,839,697	T/O	T/O	2,018,781	T/O	T/O	4,858,478	T/O	T/O

than the scopes with one more unary tuple. Moreover, except for one model, BORDEAUX did not terminate within *90* minutes if the example size is large, and the scopes increase by two. Such a lack of results within the time-limit is more frequent once the scope increases by three. Selecting the tightest scope increase can make the problem tractable for BORDEAUX. If BORDEAUX cannot find a near-miss example with the least scope increase, it can increase the scopes and search in a larger universe of discourse.

4.6.2 Parallelization

Increasing the number of atoms exponentially elevates the size of the SAT-formula, the translation time to generate it, and its solving time. In some cases, such as the Binary Tree model with a large size example, if the scope is not properly selected, Alloy* cannot find a

Table 4.5 Parallelizing PPFF can improve the efficiency of BORDEAUX. Columns show the ratio of metrics measured from solving without breaking PPFF, recorded in columns labeled by '+1' in Table 4.4, to different approaches solving PPFF for each relation. The columns *Min-R* and *Max-R* show the improvement ratio for using parallelization. For *Min-R*, the first process finds a near-miss example, and for *Max-R*, all processes finish their searches. Columns *Seq-R* shows the differences while all processes run sequentially.

		SAT Variables			SAT Clauses			Translation Time			Execution Time			Total Time		
		Min-R	**Max-R**	**Seq-R**	**Min-R**	**Max-R**	**Seq-R**	**Min-R**	**Max-R**	**Seq-R**	**Min-R**	**Max-R**	**Seq-R**	**Min-R**	**Max-R**	**Seq-R**
Arbitrary	**Singly-linked List**	4.43	1.25	*0.55*	5.07	0.74	*0.60*	1.67	1.50	*0.52*	5.80	4.83	*1.81*	3.14	2.75	*0.98*
	Doubly-linked List	3.24	1.89	*0.45*	3.81	0.48	*0.51*	72.58	45.74	*10.79*	10,138.67	322.72	*170.40*	288.67	129.61	*36.72*
	Binary Tree	5.04	1.27	*0.84*	5.84	0.74	*0.92*	1.33	1.20	*0.43*	1.00	1.00	*0.33*	1.23	1.14	*0.40*
	Graduation Plan	2.05	1.49	*0.28*	2.11	0.65	*0.28*	2.01	1.37	*0.26*	54.88	36.58	*13.30*	4.14	2.83	*0.55*
	File System	2.98	2.22	*0.27*	3.28	0.41	*0.30*	23.25	14.93	*1.96*	1,336.60	636.48	*230.45*	97.45	61.38	*8.44*
Small	**Singly-linked List**	2.25	1.32	*0.51*	2.44	0.71	*0.55*	1.33	1.20	*0.43*	4.75	3.17	*1.27*	2.38	1.94	*0.72*
	Doubly-linked List	1.45	1.45	*0.34*	1.57	0.64	*0.37*	1.15	1.07	*0.24*	2.68	2.38	*1.23*	2.23	2.01	*0.76*
	Binary Tree	1.49	1.38	*0.47*	1.57	0.70	*0.49*	2.76	2.39	*0.88*	20.14	4.15	*2.82*	5.47	3.15	*1.45*
	Graduation Plan	2.36	1.67	*0.31*	2.48	0.57	*0.33*	2.25	1.36	*0.29*	83.83	71.86	*22.86*	5.85	3.58	*0.77*
	File System	5.49	2.30	*0.34*	6.27	0.40	*0.38*	1.25	1.11	*0.12*	1.05	1.02	*0.15*	1.10	1.05	*0.14*
Medium	**Singly-linked List**	1.95	1.18	*0.46*	2.07	0.80	*0.48*	4.40	1.53	*0.70*	9.80	6.13	*2.45*	5.75	2.25	*1.01*
	Doubly-linked List	2.19	1.49	*0.34*	2.42	0.62	*0.37*	3.71	2.66	*0.60*	5.54	4.04	*0.92*	4.80	3.47	*0.79*
	Binary Tree	3.07	2.07	*0.88*	3.34	0.46	*0.94*	73.25	44.45	*19.87*	580.76	46.79	*38.72*	101.92	45.18	*23.56*
	Graduation Plan	2.27	1.60	*0.30*	2.35	0.61	*0.31*	2.22	1.21	*0.29*	19.38	3.30	*2.25*	3.21	1.55	*0.41*
	File System	2.80	1.93	*0.23*	2.73	0.53	*0.23*	3.44	2.11	*0.27*	823.38	268.86	*75.71*	7.11	4.35	*0.55*
Large	**Singly-linked List**	2.52	1.31	*0.52*	3.19	0.62	*0.64*	98.98	19.26	*13.85*	8.67	4.56	*1.90*	98.73	19.24	*13.83*
	Doubly-linked List	3.23	1.88	*0.45*	3.78	0.48	*0.51*	69.71	37.89	*9.72*	7.76	0.40	*0.20*	67.24	26.45	*7.96*
	Binary Tree	2.81	1.75	*0.78*	3.16	0.55	*0.84*	312.76	182.68	*83.34*	3,513.98	337.54	*168.55*	448.27	215.48	*100.14*
	Graduation Plan	2.55	1.91	*0.35*	2.31	0.51	*0.32*	2.26	1.42	*0.31*	1.84	1.59	*0.51*	2.26	1.42	*0.31*
	File System	3.94	2.79	*0.32*	4.03	0.34	*0.34*	155.98	54.37	*9.71*	112,154.50	151.16	*149.89*	266.60	74.08	*15.88*

near-miss example within several hours. Another factor that influences on the magnitude of the SAT-formula is the number of integer atoms that BORDEAUX incorporates into the formula to prevent the integer overflow that might occur for distance calculations.

Observing that most examples are near-hit examples and can become near-miss examples by adding or removing a single tuple, we make new formulas so that each one applies PPFF on individual relations. Solving each formula, BORDEAUX may find a near-miss example for a given example regarding a particular relation of the model.

Finding near-miss examples for each relation has the benefit of avoiding additional integers in the universe of discourse. Depending on how many relations a model has, BORDEAUX can solve a PPFF per each relation that leads to a relatively smaller universe of discourse. As BORDEAUX can independently find near-miss examples per each relation, one approach is to parallelize the search so that each process searches for a near-miss

example per each relation.

The parallelization applies to all relations in the model. The parallelization has no particular restriction on the scopes of the model's relations. However, selecting proper scopes increases the performance. If a process tries to find a minimum distance with respect to a unary relation, increasing the scope of the relation by one causes a found instance to be in the distance of one, provided adding tuples is requested. Since Alloy only allows restricting scope for unary relations, no increase is the tightest scopes for a process that tries to find a minimum distance with respect to a non-unary relation; more than one tuple of a non-unary relation can also change.

In this approach, if a process finds a near-miss example first that is at a distance of one from the given example, then all other processes can stop their searches. Otherwise, all processes should continue. In the end, either a) the last process returns a near-miss example with the distance of one from the given example, b) all processes return nothing, c) some processes return near-miss examples with a distance of two, or d) some processes return near-miss examples with a distance of three or more.

If PPFF can find a near-miss example in distance one from the given example, then one of the processes must be able to find it too. Clearly, the near-border examples finder formula found the example because only one relation gets or loses a tuple, so running the formula over that relation will get the same result. If such a near-miss example exists, one process has to return it. In case (a), the last finished process returns such a near-miss example.

If no process returns a near-miss example, *i.e.*, case (b), either such an instance does not exist at any distance from the given example or adding or removing more than one tuple from two or more relations turns the given example into a non-example. In the first situation, PPFF also returns no near-miss example; however, PPFF returns a near-miss example once tuples of more than a single relation need to be changed.

In case (c), some processes return near-miss examples with distance two from the given example. Then two is the true minimum distance. If there were a closer near-miss, it would be at distance one, and one of the other processes would have found it. Since that didn't happen, two is the minimum distance.

If a process returns a near-miss example with a distance of three from a given example and all other processes return no shorter distances, the PPFF might find a near-miss example in a closer distance which is exactly two. The distance might be two if simultaneously altering tuples of two relations makes a near-miss example; therefore, the individual processes cannot find such a near-miss example. If there was a near-miss example with

distance one, processes should have returned it. In this case, distance three might be a local minimum. The same argument is valid for a distance of four or more in case (d).

As the case studies show, distances between an example and its paired near-miss example is highly likely to be one; therefore, parallelizing BORDEAUX would often give the near-miss examples. As discussed, the process could also provide a good approximation of the minimum distance. In our practiced cases, all near-miss examples are found in distance one from given examples.

As Table 4.5 shows, parallelization improves the search for near-miss examples. In all studied models, regardless of the sizes of the given examples, parallelization decreases the size of the SAT-formula, the translation time to generate it, and the solving time. We have measured this improvement by recording the time and resources taken to find the first near-miss example, as well as the time and resources taken to finish all parallel processes. Compared to using non-broken PPFF with the least scope increase, the time to concurrently find the first near-miss example decreases by the ratio of *70.9* on average. Waiting for the termination of all processes' results changes the ratio to *30.2*.

If there are not enough resources available for parallelization, sequentially running the decomposed processes still has value. The studied models show that the sum of the process times is often less than the general time when the size of an example is large. Since most of the Alloy statements synthesized for each process are the same, the translation time might be saved by reusing some parts that have already been translated for the formula of another relation. Full assessment of this idea is left for future work.

4.7 Related Work

Both Nelson et al. [78] and Cunha et al. [20] have proposed techniques to guide Alloy Analyzer's example generation facility towards more interesting examples. The Alloy Analyzer 4.2 generates arbitrary examples, which might or might not be interesting, and might or might not help the user discover underconstraint bugs.

The Nelson et al. [78] extension of Alloy, called Aluminum, generates *minimal* examples. We say that this approach produces *absolute minimum* examples, because it finds the smallest examples that satisfy a given model. By contrast, our technique looks for *relatively minimum* pairs of examples: one example (near-hit) and the other non-example (near-miss) that are at a minimum distance from each other; they might not be absolutely minimal from Aluminum's perspective. Aluminum also has a facility for growing the minimal example, called *scenario exploration*.

Cunha et al. [20] used PMax-SAT [15] to enhance Kodkod [102] to find examples that are close to a target example. They discussed applications in data structure repair and model transformation. Perhaps this technique could be modified to replace our usage of Alloy* in Bordeaux.

When the model is completely overconstrained (*i.e.*, inconsistent), then no examples are possible. Shlyakhter et al. [96] enhanced Alloy to highlight the *unsatisfiable core* of such models; Torlak et al. [103] further enhanced this functionality. This tells the user a subset of the model (*i.e.*, formula) that needs to be changed, but does not give an example (because none satisfy the model).

Browsing desirable non-examples might help the user understand what is wrong with the model [4]. In an empirical user study, Zayan et al. [115] evaluated the effects of using examples and non-examples in model comprehension and domain knowledge transfer. The study demonstrated evidence of the usefulness of non-examples in understanding the models, but did not state any preferences for particular examples. Browsing (desirable) non-examples might also help the user understand partially overconstrained models (in which some, but not all, desirable instances are possible).

Batot [6] designed a tool for automating MDE tasks, such as model transformation or well-formedness rule extraction. The tool generates examples from partial or complete metamodels to be evaluated or corrected by an expert. The minimality and coverage of examples are two major criteria for generating useful examples. The coverage is defined based on the MDE automation task, some frequent OCL templates, and model slicing. Mottu et al. [70] proposed a mutation analysis technique to improve model transformation testing. Their technique mutates the model w.r.t. four abstract model transformation operators and generates mutants for evaluating test-suites. Macedo and Cunha [60] proposed a tool for analyzing bidirectional model transformations based on least changes using Alloy. The tool tries different number of changes to find the least number. Selecting proper scopes for Alloy Analyzer is a major obstacles to scaling the tool.

In his seminal work Winston [110], introduced using near-miss examples in learning classification procedures as well as explaining failures in learning unusual cases. Gick and Paterson [34] studied the value of near-miss examples for human learning. They found that contrasting near-miss examples were the most effective examples for learning. Popelínsky [83] used near-miss examples for synthesizing normal logic programs from a small set of examples. Seater [94] employed the concepts of near-miss and near-hit examples to explain the role, *i.e.*, restricting or relaxing, of a constraint in a given model. Modeling By Example [64] is an unimplemented technique used to synthesize an Alloy model using near-miss and near-hit examples. The technique synthesizes an initial model from a set

of examples; it learns the borders by generating near-miss and near-hit examples to be reviewed by the user. The near-hit and near-miss examples are from a slightly modified model. The concepts of near-hit and near-miss examples are conceptually similar to near-hit and near-miss examples, respectively. Apart from the contextual difference, our idea to generate near-hit and near-miss examples is based on slight changes in the instance versus the model.

ParAlloy [91] and Ranger [92] realize parallel analysis of models written in Alloy. Both tools partition a given Alloy model and make multiple calls to the underlying SAT-solver. The idea of parallelization in Bordeaux relies on selecting proper scopes as opposed to partitioning the model.

4.8 Summary

Bordeaux is a tool for finding near-hit and near-miss example pairs that are close to each other, as stated in **Contribution 2**. The near-hit example is an example of the model the user wrote. The near-miss example is almost consistent with the user's model except for one or two crucial details. Others have found near-miss examples to be useful [4, 34, 110, 115]. In particular, Gick and Paterson [34] found that pairing a near-miss example with a similar near-hit example increased human comprehension of the model. We posit that such pairs might be particularly helpful for discovering and diagnosing partial over-constraints in the model. Tool support for this task is currently limited.

The Bordeaux prototype has been built to work with ordinary Alloy models. It works by transforming the user's Alloy model and synthesizing a query with higher-order quantifiers that can be solved by Alloy* [65]. Through experiments we have observed that near-hit and near-miss examples often differ in no more than one tuple. We have based two optimizations on this observation: scope tightening and parallelization. Together, they significantly reduce the cost of searching. The formalization of the idea, the PPFF (Figure 4.3d), is more general than the specific use-case that our narrative has centred on. The formalization works from a pair of inconsistent constraints. The use-case narrative in this chapter has focused on the specific circumstance when one constraint is the negation of the other, and sometimes even more narrowly on when the first constraint is a specific example. In Chapter 5, we develop a technique to produce particular near-hit and near-miss examples.

Chapter 5

Margaux: A Pattern-based Approach for Debugging Underconstraint and Overconstraint

The need to debug arises because the *expressed* meaning of a model differs from the *intended* meaning of the model, but the user does not know where or why [54]. In this chapter, we explain **Contribution 1**, an approach to produce *discriminating examples* , focused examples and non-examples encoding hypotheses on whether the expressed model might differ from the user's intention. One hypothesis is to check whether bugs happen close to borders; therefore, near-hit and near-miss examples are the candidates to be encoded as discriminating examples. Another hypothesis to check using discriminating examples is whether or not the model has particular semantics. Based on the semantics encoded in the form of examples, the user decides whether a constraint should be strengthened or weakened (or stay as is). We propose *pattern-based debugging* as a semi-automatic technique to assist with localization and understanding of these differences. Using the discriminating examples, our technique can focus on particular constraints for evaluating whether certain one causes a total or partial overconstraint bug.

Pattern-based debugging comprises *pattern-based simulacrum inference* and *semantics mutation* to produce discriminating examples. In this context, a *pattern* is a general idea such as acyclicity. When a pattern is instantiated with respect to a particular relation, we call it a *property*, *e.g.*, r is acyclic. *Pattern-based simulacrum inference* is the process of discovering which properties the model implies, is consistent with, and is inconsistent with. Such properties comprise *synonyms* and *antonyms* of a model or its constituent constraints. These properties form the simulacrum of the model or its constraints.

Simulacrum mutation is the process of changing this inferred simulacrum by either strengthening or weakening individual properties, listed as **Contribution 3**. Such mutation is done by two operators: *Weakener Mutation Operator* and *Strengthener Mutation Operator*. The former mutation operator relaxes the model and may lead to revealing a overconstraint bug, whereas the latter mutation operator might reveal underconstraint bugs in the model. We have computed both an *implication graph* and a *conjunction graph* that are used to guide this reasoning process. The implication graph records which patterns strengthen or weaken to other patterns. The conjunction graph records which patterns are mutually satisfiable, and which patterns are in conflict (mutually unsatisfiable). The debugger uses the mutations to produce discriminating-examples for the user to accept or reject. Through this dialogue, the mutation will move closer to the user's intentions.

To orchestrate the debugging procedure, we have developed a procedure to analyze the simulacrum of a given model, mutate its simulacrum, and produce discriminating examples. The base-line procedure makes some non-deterministic decisions for selecting the parts of the model that should be mutated and the mutation operators that should be applied. The non-deterministic choices can lead to many discriminating examples for the user to review before the one revealing the bug. We have developed a set of heuristics for regulating the decisions so as to produce fewer discriminating examples. As we evaluated the procedure with different Alloy models, the heuristics dramatically decreased the number of discriminating examples. Some heuristics add extra analysis cost so that the procedure takes longer to produce fewer discriminating examples.

For assessing the idea of debugging with discriminating examples, we have developed Margaux. Given an Alloy model, Margaux generates discriminating examples by mutating the simulacrum of the model using predefined patterns. By reviewing focused discriminating-examples, the user explores different corners of the model and finds out whether the model suffers from under or over-constraint issues. Using Margaux in Chapter 6, we demonstrate how the pattern-based debugging can guide the user to find a bug, understand it, and fix corresponding statements. Section 5.1 contains the fundamental definitions for pattern-based debugging approach. In Section 5.2.1, we explain the details of the debugging procedure and mutation operators. Section 5.3 includes the related works and section 5.4 summerizes the chapter.

5.1 Foundations

In this section, we review and define the fundamental concepts and definitions used in debugging and discriminating example search procedures. First, we briefly review Alloy as

the language of models that MARGAUX takes to debug. Then, we define the concepts for analyzing and mutating the model in order to generate examples and non-example near borders.

5.1.1 Discriminating Example

Given an Alloy model, MARGAUX assists its user to ensure that the model is consistent with her intention. The actual expressed model as M_E [Definition 2] and the conceptual intended model as M_I [Definition 3]. MARGAUX assists its users in debugging expressed Alloy models by producing its particular examples [Definition 10] and non-examples [Definition 11]. The user then only should review these instances [Definition 9] and provide her feedbacks by accepting or rejecting them. The debugger might reveal inconsistencies between the intended model and expressed model based on acceptance/rejection responses. We define these type of examples and non-examples as:

Definition 17 ***(Discriminating Example).*** is an instance that either **(a)** satisfies M_E but not M_I or **(b)** satisfies M_I but not M_E.

With this definition, MARGAUX can distinguish the type of a bug that an expressed model faces with. Provided the user rejects an instance from case-a, then the expressed model has an underconstraint bug. There is an overconstraint bug in the expressed model if the user accepts an instance from case-b. The challenge for producing discriminating examples from its definition is that intended models are unspecified. MARGAUX resolves this challenge by proposing some hypotheses on how the intended model (as a whole or part of) might resemble. Therefore, the debugger tests each hypothesis by producing a *discriminating example* from a *discrimination formula*:

Definition 18 ***(Discrimination Formula).*** Given a constraint hypothesis H of the user's intended model, an instance satisfying $\neg H \wedge M_E$ realizes a case-a discriminating example, and an instance satisfying $H \wedge \neg M_E$ realizes a case-b discriminating example.

In the following sections, we discuss using near border examples and simulacrum mutation to generate H in the discrimination formula and thereby produce discriminating examples.

5.1.2 Borders and Discriminating examples

A general hypothesis is that mistakes happen near borders. MARGAUX uses discriminating examples to test such hypothesis. As discussed in Chapter 4, a border is a concept sepa-

rating examples from non-examples in this context. Near-hit examples are instances that become non-example by adding or removing a minimum number of tuples; likewise, near-miss examples are instances that become example after losing or gaining a least number of tuples.

MARGAUX considers near-hit and near-miss examples of an expressed model (M_E) to be discriminating examples. If the user rejects a near-hit example of M_E, the model faces an underconstraint issue. M_E has an overconstraint bug provided the user accepts a near-miss example. MARGAUX constructs discrimination formula to produce this type of discrimination example. For the expressed model, we call $N^h_{M_E}$ as a constraint that all near-hit examples of M_E satisfy. Likewise, $N^m_{M_E}$ is a constraint that all near-miss examples of M_E satisfy.

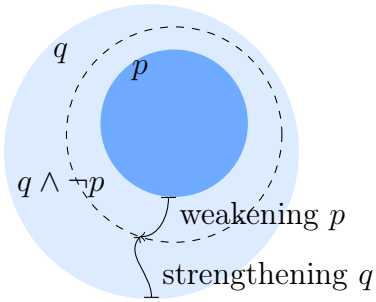
Considering these two constraints for specifying near-hit and near-miss examples of M_E, H in the discrimination formula for producing a discriminating example revealing an underconstraint is $(\neg N^h_{M_E})$. Also, H for producing a discriminating example exposing an overconstraint is $(N^m_{M_E})$. As we have implemented BORDEAUX [§4] for finding near-hit and near-miss examples, MARGAUX does not directly constructs a discrimination formula and just calls BORDEAUX for finding these examples.

5.1.3 Mutation and Discriminating examples

Using discriminating examples, MARGAUX assesses different hypotheses of how the expressed model should be modified to become identical with the user's intent. One of these hypotheses is that the user may intend a model to be stronger or weaker than the expressed model. The debugger assesses such a hypothesis by mutating the model and producing discriminating examples consistent or inconsistent with them. Depicted in Figure 5.1, accepting or rejecting such a discriminating example reveals whether the model has an underconstraint or overconstraint bug.

One can imagine such discriminating examples as test-cases that check whether the model complies with her intentions. Such a test-suite for evaluating the given model provides the user with a rough guide to whether a model may satisfy her intentions in critical aspects. Methods like Mutation Testing [22] help the user to evaluate a test-suite's capability to reveal unpredicted bugs. Mutation Testing relies on two hypotheses [11]: one that programmers are competent, *i.e.*, they develop code very similar to their intention; and the other related to coupling effect *i.e.*, detecting minor issues could implicitly reveal a more complex issue. According to these hypotheses, MARGAUX uses the concept of mutation to generate discriminating examples.

Figure 5.1 The dark centre circle represents instances satisfying property p, whereas the lighter outer circle represents instances satisfying property q. The containment relationship indicates that p implies q. Discriminating examples are inside the dotted line, but outside the dark centre circle: *i.e.*, they satisfy $q \wedge \neg p$. If the user accepts such an example then p must be weakened towards the dotted line, whereas rejecting such an example means that q must be strengthened towards the dotted line.



To generate different discriminating examples, MARGAUX constructs various discrimination formulas from weaker or stronger mutants of the expressed model. The two mutation operators are *strengthener* and *weakener*.

Definition 19 ***(Strengthener Mutation Operator).*** The strengthener mutation operator produces a mutant (M_S) of the expressed model such that some examples of M_E do not satisfy M_S. The operator then creates a discriminating formula in the form of $\neg M_S \wedge M_E$ (case-b in Definition 18).

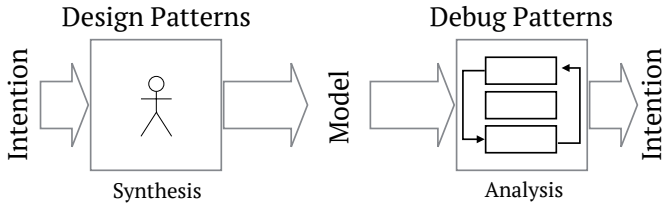
If the user rejects a discriminating example that satisfies this discrimination formula, then the expressed model faces an underconstraint.

Definition 20 ***(Weakener Mutation Operator).*** The weakener mutation operator creates a mutant (M_W) from the expressed model such that some non-examples of M_E satisfy M_W. The operator constructs a discriminating formula in the form of '$M_W \wedge \neg M_E$' and produces a discriminating example (case-a in Definition 18).

Provided the user accepts this discriminating example, then the expressed model has an overconstraint bug.

MARGAUX shows to users only discriminating examples that are generated from surviving mutations. It kills mutations generating already-reviewed discriminating examples. If the weakener mutation operator creates a mutation consisting of a previously rejected discriminating example, the mutation is also killed.

Figure 5.2 Debug patterns *VS.* Design patterns. Design patterns [32] (or specification patterns [36]) assist users in converting their ideas to software artifacts. Debug patterns help machines to extract intentions behind software artifacts.



Debugging Relational Declarative Models with Discriminating Examples 24 / 38

In the following parts, we explain the foundations for strengthening and weakening the simulacrum of a given expressed model.

5.1.4 Debug Patterns

In this work, the term *pattern* defines general concepts such as monotonicity:

Definition 21 ***(Debug Pattern).*** A debug pattern is a predicate over a relation, optionally informed by ordering information about the relation's atoms.

The idea of using patterns in debugging a formal model does not completely follow the classical approach to design or specification patterns [32, 36]. Unlike with design patterns, the debugger uses patterns in reverse. Instead of assisting practitioners to convert their intentions into formal artifacts, a debugger helps them to find whether a formal artifact complies with their intentions. However, debug patterns are directly used by the debugger, and a practitioner does not necessarily directly deal with them. Therefore, these patterns

need not always be understandable to humans, but should be be accurate and compact. A library of debug patterns suitable for the debugger is machine readable and facilitates reasoning procedures. Also, libraries of design patterns aim to be useful, but not necessarily comprehensive. Over time, practitioners and researchers might discover new debug or design patterns that are also useful.

Definition 22 ***(Synonym).*** If a given constraint c restricts all valuations of a relation r according to a pattern p (c$\Rightarrow$ p[r]), we say p is a synonym of c with respect to r.

For example, if a constraint c implies the acyclic pattern, from structural binary patterns, over a relation r, then we say that acyclicity of r is a synonym of c.

Definition 23 ***(Antonym).*** If a given constraint c prevents valuations of a relation r according to pattern q ($\neg$c$\vee\neg$q[r]), we say q is an antonym of c with respect to r.

Definition 24 ***(Partial-synonym).*** If a given constraint c partially restricts valuations of a relation r according to a patterns p (c$\wedge$p[r]$\models$SAT), we say p is a partial-synonym of c.

Definition 25 ***(Simulacrum).*** A simulacrum of a constraint c, with respect to relation r, is a triple $\mathcal{S}_c^r = \langle s, a, p\rangle$, where s is the set of synonyms of c, a is the set of antonyms for c, and p is the set of partial-synonyms for c (all with respect to r). Likewise, a simulacrum of a model M (with respect to relation r) is a triple $\langle s, a, p\rangle$ that is equal to the simulacrum of M's constraint C (with respect to relation r).

Consider that M's constraint C is a conjunction of constraints c_i: $C = \wedge_i c_i$. The simulacrum of M can be partially computed compositionally from the simulacra of the constituent constraints c_i. Specifically, M's synonyms are the union of the constituent synonyms: $s_M = \cup_i s_i$. The proof of this compositionality is simple: recall that each constituent set of synonyms contains formulas of the form $c_i \Rightarrow p_j[r]$, for however many properties are implied by c_i (with respect to relation r); since C is the conjunction of all of the constituent constraints c_i ($C = \wedge_i c_i$), then all c_i must be true for C to be true; so whatever is implied by c_i is also implied by C. Likewise, M's antonyms are the union of the constituent antonyms: $a_M = \cup_i a_i$. M's set of partial synonyms cannot be computed compositionally.

Definition 26 ***(Pattern-based Simulacrum Inference).*** Given a constraint c, a relation r, and a set of patterns, the pattern-based simulacrum inference is to find simulacrum of c with respect to r.

The pattern-based simulacrum inference is guided by both an *implication graph* and a *conjunction graph* of patterns. The implication graph, an acyclic and directed graph, records which patterns imply others, whereas the conjunction graph is undirected and records which patterns are mutually satisfiable. These graphs are computed once for the

library of patterns and then stored for use in analyzing individual models. If pattern $p \Rightarrow q$, and the constraint $c \Rightarrow p[r]$ (that is, the constraint implies pattern p applied to relation r), then there is no need to check $c \Rightarrow q[r]$. Similarly, if pattern p and pattern x are not mutually satisfiable (*i.e.*, if there is no edge between them in the conjunction graph), and the constraint $c \Rightarrow p[r]$ then there is no need to check $c \Rightarrow x[r]$. Likewise, if c is inconsistent with p, it becomes inconsistent with y, once $y \Rightarrow p$. The constraint c is consistent with p, so it is definitely consistent with properties implying p, such as q. All in all, having an implication and conjunction graphs, MARGAUX avoids the need to check all properties.

Table 5.1 The *Binary Implication Lattice* comprises *23* patterns, each represented as a node. An edge of the lattice encodes an implication relation between two patterns. The strongest patterns, *i.e.*, sources in the lattice, are distinguished in **bold** with an overline. Patterns in underlined *italic* are the weakest patterns or sinks of the lattice.

From	To
bijection	bijective, function
empty	acyclic, *functional*, *injective*,*symmetric*, *transitive*
equivalence	preorder, *symmetric*
rootedAll	stronglyConnected
rootedOne	*weaklyConnected*
totalOrder	complete, partialOrder
bijective	*injective*, *surjective*
function	*functional*, *total*
acyclic	*antisymmetric*, *irreflexive*
preorder	reflexive, *transitive*
stronglyConnected	*weaklyConnected*
complete	*weaklyConnected*
partialOrder	*antisymmetric*, preorder
reflexive	*surjective*, *total*

Figure 5.3 Parametric Temporal Patterns' structure for a ternary relation of the form r:Left→Middle→Right

$$Inclusion \times Ordering \times Column \times Staticness \times Emptiness$$

$$\left[\begin{array}{|c|}\hline Expand \\ Expandd \\ Contract \\ Contractt \\ Mutate \\ \hline\end{array} \times \begin{array}{|c|}\hline HeadOf \\ HeaddOf \\ TailOf \\ TaillOf \\ \varnothing \\ \hline\end{array} \times \begin{array}{|c|}\hline Middle \\ Right \\ \hline\end{array} \times \begin{array}{|c|}\hline \_MiddleStatic \\ \_RightStatic \\ \varnothing \\ \hline\end{array}\right] \times \left[\begin{array}{|c|}\hline \_FirstLeftEmpty \\ \_LastLeftEmpty \\ \varnothing \\ \hline\end{array}\right]$$

5.1.5 Library of Debug Patterns

A library of debug patterns comprises of a set of *patterns* and two graphs encoding *implication* and *consistency* relations between those patterns. The library of debug patterns is used for inferring the simulacrum of a constraint, and killing mutations. These graphs are computed once for the library of debug patterns and then stored for use in analyzing individual models.

Currently, MARGAUX includes two libraries of debug patterns: one for structural and the other for temporal patterns. The structural patterns encode properties for expressing how entities are structurally connected to one another. The structural patterns are for binary relations and are based on properties defined by Mendel [64]. The binary relational patterns and the implication relation between them are depicted in Table 5.1.

We present a library of ternary patterns, listed as **Contribution 4**. A previous study of Alloy models showed that ternary relations are often used in temporal models [86]: a third column is added, at either end, to index a binary relation through time.

The library of debug patterns is generated from a five-dimensional cross-product (Figure 5.3). There are 450 patterns produced by this cross-product. An analysis with Alloy reveals that only 180 of these are satisfiable. These can be grouped into 160 equivalence classes (two patterns are in the same equivalence class if they are equivalent). The implication graph for this library has 12 source nodes, 6 sink nodes, and a maximum path length from source to sink of 6. Some nodes are reachable only from a subset of the source nodes.

For the ternary relation r, with column Left, Middle, and Right: **sig** Left{r: Middle→Right}. Table 5.2 shows example values for this relation that conform with four different patterns.

Consider the pattern ContracttTailOfRight_MiddleStatic. We see that $L_0 \rightarrow M_0 \rightarrow$

$\{R_0, R_1, R_3\}$; in the next time step we see that the tail of the right column (atom R_0) has been removed (contracted), while the middle column remains the same: $L_1 \rightarrow M_0 \rightarrow \{R_1, R_3\}$. The double 't' on 'Contractt' indicates that the contraction must happen at each time step.

By contrast, the pattern ExpandHeadOfRight_MiddleStatic does not have a double 'd' at the end of 'Expand', so the expansion does not need to happen at every time step: things might stay the same, as they do from L_0 to L_1. When atoms are added to the right column, they must be greater than the existing atoms in that column. For example, state L_2 adds R_1, and L_3 adds R_2. The MiddleStatic part of the pattern indicates that once an association is made between a middle atom and a right atom, that association persists. For example, the association $M_0 \rightarrow R_0$ exists in the first state (L_0) and in all subsequent states. Similarly, in state L_2 the association $M_1 \rightarrow R_1$ is made, and then persists for all future states.

Pattern ContracttMiddle_LastLeftEmpty strictly shrinks the number of atoms in the middle column as the states progress, ending in the empty set. For example, at state L_0 we see $\{M_0, M_1, M_2\}$. Then at state L_1 this has been reduced to $\{M_0, M_2\}$. There is no *ordering* constraint in the pattern, which is why M_1 could be removed. The LastLeftEmpty constraint in the pattern enforces that in the last state, L_3, there are no associated middle or right atoms.

The example for the MutateHeadOfMiddle pattern shows that the head of the middle column changes from M_1 to M_2 as the state progresses from L_0 to L_1. The transition from L_1 to L_2 reveals some subtleties of this pattern. First, the head of the middle is not required to change at each time step, because 'head' does not have a double 'd', and from L_1 to L_2, the middle column does not change. Second, the contents of the right column do change from L_1 to L_2, which at first might seem confusing. This change is permitted because the pattern does not include RightStatic.

Table 5.2 Examples satisfying four samples from temporal patterns expressed in Figure 5.3. Each example is represnted as a set of tuples of a relation of the form r:Left→Middle→Right

ExpandHeadOfRight_MiddleStatic			ContracttTaillOfRight_MiddleStatic			ContracttMiddle_LastLeftEmpty			MutateHeadOfMiddle		
Left	Middle	Right	Left	Middle	Right	Left	Middle	Right	Left	Middle	Right
L0	M0	R0	L0	M0	R0	L0	M0	R0	L0	M0	R0
L1	M0	R0			R1		M1	R1		M1	R1
L2	M0	R0			R3		M2	R0	L1	M0	R0
	M1	R1	L1	M0	R0	L1	M0	R0		M2	R0
L3	M0	R0			R3		M2	R0	L2	M0	R1
	M1	R1	L2	M0	R3	L2	M2	R0		M2	R0
		R2	–	–	–		∅	∅	–	–	–

Meta-programming techniques are used to generate the formulas for patterns in this library, as in Figure 5.4 for ExpandHeadOfRight_MiddleStatic. The highlighting shows which parts of the formulas are generated from which concepts in the pattern. Full details of the library of ternary patterns is in Appendix A.

Figure 5.4 Examples of how formulas for patterns are generated. The different colours of highlighting show which parts of the formula in the property correspond to which concepts in the pattern.

```
1 open utils/ordering[Left] lo
2 open utils/ordering[Middle] mo
3 open utils/ordering[Right] ro
4
5 pred ExpandHeadOfRight_MiddleStatic{
6   all l: Left − lo/last | let l'= lo/next[l] | all m: Middle |
7     let i= m.(l.r) | let j= m.(l'.r) | let delta = j - i |
8       (i in j) and ( some delta implies ro/lte[ ro/min[i], ro/min[delta] ] )
9 }
```

5.1.6 Synonyms and Antonyms Variants

The variations of *synonyms* and *antonyms* as well as their formal definitions are presented next, as a precursor to explaining mutation operators.

Given a library of patterns and a model, synonyms of a constraint with respect to a relation defined in the model are properties implied from the constraint. The synonyms and antonyms of a constraint can be strengthened or weakened. With an implication graph, a property that implies a synonym of the constraint is called a *stronger* synonym. *Weaker* synonyms are the properties implied from the synonym. Weaker and stronger antonyms of a constraint are defined in the same way, unless a weaker antonym might not be necessary to an antonym any more.

The strongest synonyms of a constraint are those directly implied from the constraint. In Algorithm 1, given two constraints, c and c', and a relation, r, *consistent strongest synonyms* are the strongest synonyms of c consistent with c'.

The *Weakest partial synonyms* of a constraint are those properties consistent only with the constraint, but not implied from it. The *Weakest antonyms* of a constraint with respect to a relation are patterns inconsistent with the constraint, yet any properties weaker than that antonym do constitute an antonym. The *Consistent weakest antonyms* of constraints c and c' are the weakest antonyms of c that are consistent with c'.

Algorithm 1: Variations of Synonyms and Antonyms referred to in mutation operators. Each variation is a set returned according to function definition. As the functions have common inputs, they are declared upfront. All functions have access to implicit parameters. The constraints c and c' may be formulas consisting of the constraints belonging to C.

Implicit: Model $N = \langle C, R, B \rangle$
Input: Library of debug patterns $\pi = \langle \Omega, \mathcal{U}, \mathcal{G} \rangle$. For each graph, V and E refer to Nodes and edges.
Input: A constraint c
Input: A constraint c'
Input: A relation $r \in R$
Input: A property $p \in \Omega$
Output: A set of synonyms and antonyms depending on which function is called.

`/* Strongest Synonyms of` c `Consistent with` c' `*/`

Function `SS_C`(π, c, c', r)**return**
  $\{\omega \in \pi.\Omega | ((c \Rightarrow \omega[r]) \wedge$ `Satisfiable(`$c' \wedge \omega[r]$`)` $) \wedge \forall \omega' \in \pi.\Omega - \omega | (\omega' \rightarrow \omega \in \pi.\mathcal{U}.E) \Rightarrow \neg((c \Rightarrow \omega'[r]) \wedge$ `Satisfiable(`$c' \wedge \omega'[r]$`)`$)\}$
end

`/* Weakest partial Synonyms of` c `*/`

Function `WpS`(π, c, r)**return**
  $\{\omega \in \pi.\Omega |$ `Satisfiable(`$c \wedge \omega[r]$`)`
  $\wedge \neg(c \Rightarrow \omega[r]) \wedge \forall \omega' \in \pi.\Omega - \omega | (\omega \rightarrow \omega' \in \pi.\mathcal{U}.E) \Rightarrow (c \Rightarrow \omega'[r])\}$
end

`/* Weakest Antonyms of` c `Consistent with` c' `*/`

Function `WA_C`(π, c, c', r)**return**
  $\{\omega \in \pi.\Omega | \neg$`Satisfiable(`$c \wedge \omega[r_j]$`)` $\wedge$ `Satisfiable(`$c' \wedge \omega[r_j]$`)`
  $\wedge \forall \omega' \in \pi.\Omega - \omega | (\omega \rightarrow \omega' \in \pi.\mathcal{U}.E) \Rightarrow$ `Satisfiable(`$c \wedge \omega'[r]$`)` $\}$
end

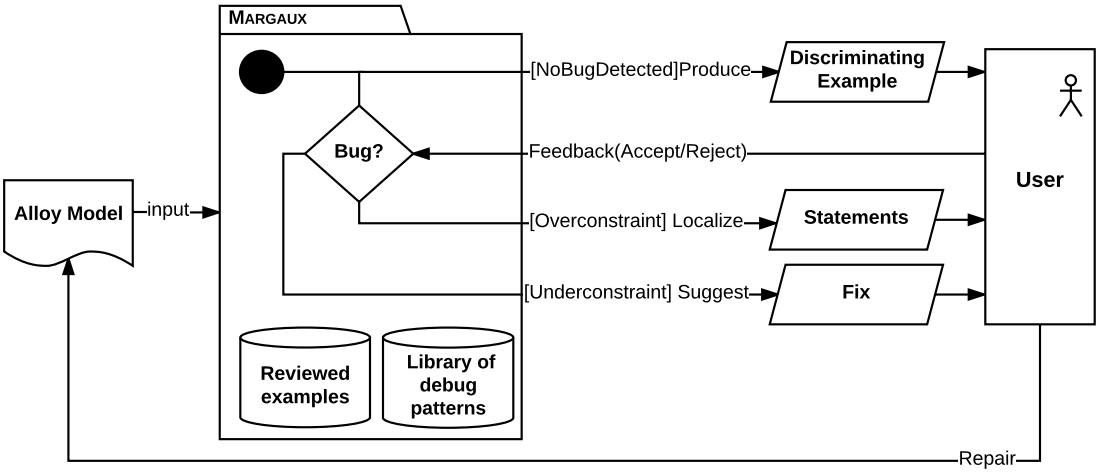
`/* partial Synonyms of` c `Stronger than` p `*/`

Function `pS_S`(π, c, p, r)**return**
  $\{\omega \in \pi.\Omega | (\omega \rightarrow p \in \pi.\mathcal{U}.E) \wedge$ `Satisfiable(`$c \wedge \omega[r]$`)`
  $\wedge \forall \omega' \in \pi.\Omega - \omega | (\omega' \rightarrow p \in \pi.\mathcal{U}.E) \Rightarrow (\omega \rightarrow \omega' \notin \pi.\mathcal{U}.E)\}$
end

As needed, mutation operators may ask for only weaker or stronger synonyms or antonyms. *Stronger consistent partial synonyms* of a synonym are the immediate predecessors of the synonym in the implication graph.

So far, we have defined four specific variations of synonyms and antonyms. These definitions are directly used in mutation operators. With other mutation operators, one might define different variations of synonyms and antonyms.

Figure 5.5 User interaction with MARGAUX. The libraries of debug patterns and reviewed examples are shared between MARGAUX's components.



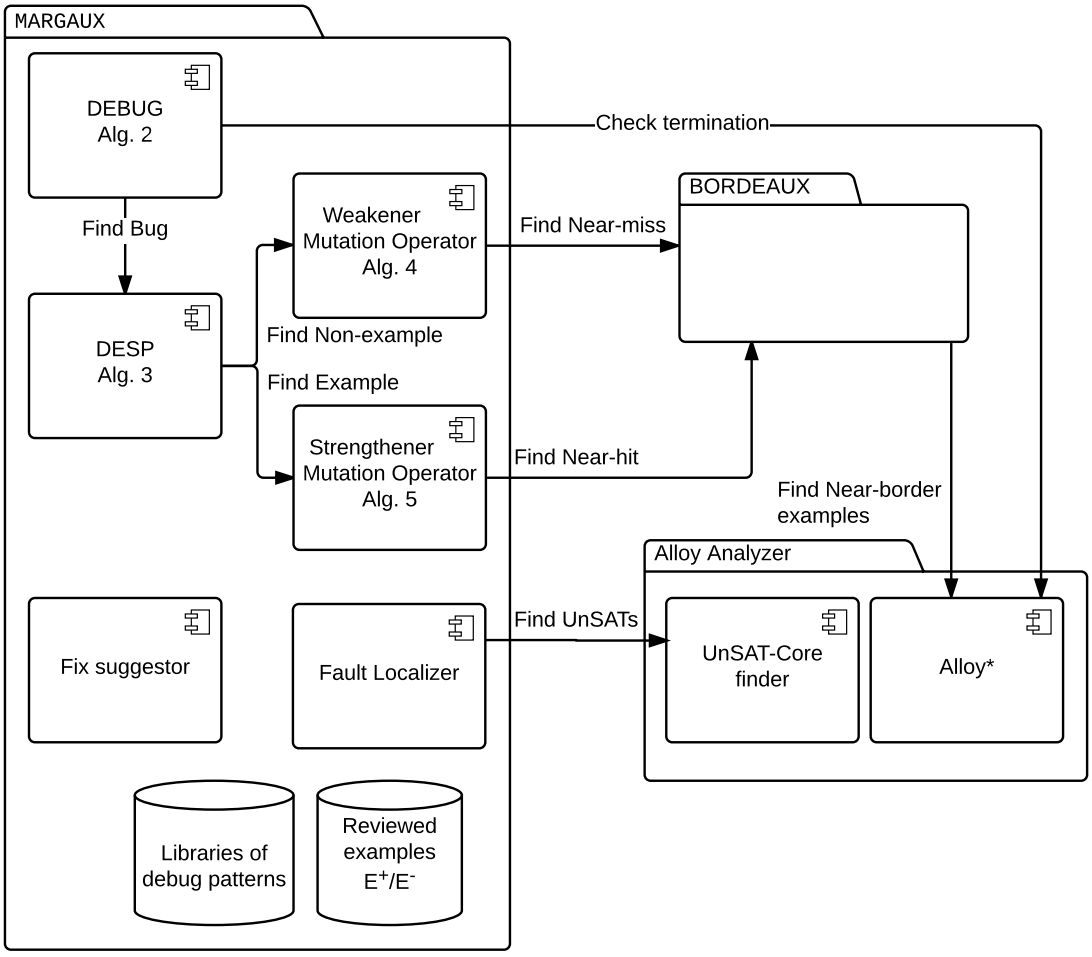
5.2 A Design for MARGAUX

MARGAUX assists the user to debug Alloy models by producing discriminating examples. By judging such examples, the user can understand different aspects of the model that might be improperly specified. As MARGAUX emphasizes on helping its users in understanding the model, its main mission is to produce discriminating examples revealing potential divergence from intended idea.

To begin debugging, MARGAUX guesses some hypotheses for testing whether the engineer's intention might differ from the expressed model. As depicted in Figure 5.5, the debugger tests each hypothesis by producing discriminating examples and inquiring them from the engineer. By producing non-examples, the debugger tests the hypothesis that a model should be weaker than its current state. If the user accepts this discriminating example, the debugger alerts for an overconstraint bug and localizes the statements that cause such a discriminating example to be excluded. The debugger produces examples of a model to test another type of hypotheses that the model might have underconstraint issue. If the user rejects any of these examples, the debugger reports the bug and tries to suggest fixes for mitigating it.

MARGAUX uses a library of patterns for finding and representing simulacrum of the model. When the debugger analyzed the simulacrum of the model, it weakens or strengthens the simulacrum for mutating the model. Each result of these mutations carries a hypothesis that the model should be weakened or strengthened in order to become consis-

Figure 5.6 Major components of MARGAUX and other extensions. The libraries of debug patterns and reviewed examples are shared between MARGAUX's components.



tent with the user's intention. The debugger tests these hypotheses by producing examples and non-examples from the mutations.

At each discriminating example generation iteration, MARGAUX checks all previously reviewed discriminating examples to see if the changes align with her previous answers. The major components of MARGAUX are depicted in Figure 5.6. In this section, we explain MARGAUX's components and their interactions with other tools.

Algorithm 2: DEBUG

Input: Libraries of debug patterns, Π. Each library in the form of $\langle \Omega, \mathcal{U}, \mathcal{G} \rangle$, where Ω is a set of of patterns. $\mathcal{U}$, $\mathcal{G}$ are implication and consistency graphs, respectively.

Input: Model $M = \langle F, R, B \rangle$, where F is in the form of $Y \Rightarrow P$ for checking mode and Y for simulating mode, Y and P are two Alloy formulas. $R = \{r_1, \ldots, r_m\}$ is a set of relations and $B = \{b_1, \ldots, b_m\}$ represents their bounds.

Output: A list of tuples that each includes accepted and rejected discriminating examples, the models before and after repair, and a bug detection result $(NoBugDetected, Underconstraint, Overconstraint)$.

```
1  E+, E- ← ∅                                   ▷ All discriminating examples the user rejected/accepted.
2  M_R ← M
3  Σ ← ⟨E+, E-, M_R, ∅, ∅⟩                      ▷ Initializing the return list of tuples.
4  repeat
5  |   M ← M_R
6  |   N ← ⟨ convert_Y(M.F), convert_P(M.F), M.R, M.B⟩   ▷ convert_Y() extracts a set of
   |     constraints from Y part in F.
7  |   ρ ← DESP(Π, N, E+, E-)                   ▷ Call to discriminating example generation Algorithm 3
8  |   if ρ[3] ≠ NoBugDetected then             ▷ Third entry of the returned tuple should be the bug name.
9  |   |   M_R ← repair(M)                     ▷ The user repairs M such that M_R.R = M.R and M_R.B = M.B
10 |   end
11 |   Σ ← Σ + ⟨ρ[1], ρ[2], M, M_R, ρ[3]⟩      ▷ Append the result to the list.
12 until !((E+ ∪ E-) ≠ ∅ ⇒ ∀e : E+|e ⊨ M_R ∧ ∀e : E-|e ⊭ M_R) ⋀      ▷ Condition-1
13 ⟨E+, E-⟩ kills all mutations of M generated by Algorithm 3 ⋀      ▷ Condition-2
14 (Provided M is a property checking form, i.e. Y ⇒ P, then ¬Satisfiable(M))      ▷ Condition-3
15 return Σ
```

5.2.1 Debugging Procedure

DEBUG, depicted in Algorithm 2, is the main procedure of MARGAUX. Given an Alloy model, DEBUG decomposes it into two sets containing constraints and relations. The model decomposition for extracting the constraints set differs slightly for simulating and checking modes. A typical Alloy model executed in the checking mode follows the form of $Y \Rightarrow P$; in the simulating, it is Y. Regardless of the execution mode, DEBUG converts Y into a set of constraints. The conjunction of constraints in the set is equivalent to Y. The constraints in the set should have no conjunctions, unless enclosed by quantifiers. However, if Y contains a predicate or function call, the debugger does not inline and decompose its body at this step. The extracted set and the library of patterns are implicitly shared between DEBUG and succeeding procedures called from DEBUG.

DEBUG iteratively calls the DESP algorithm, discussed in Section 5.2.2, to generate discriminating examples and check them with the user's intentions. Once DESP finds a bug,

it terminates and returns the bug type as well as any discriminated examples evaluated by the user. During the next interactions, this information is used to prevent redundant discriminating example generation, kill further mutations, and check termination conditions.

The simulacrum of a constraint includes three finite sets of properties, and the debugger avoids making redundant mutations; therefore, DEBUG has to terminate after generating limited number the discriminating examples at each iteration. In the end, DEBUG should ensure that the model is equivalent with the user 's intention in accordance with the library of patterns, *i.e.*, $M_E \Leftrightarrow M_I$. Once a repaired model passes the following termination conditions, DEBUG returns a list of tuples recording revealed bugs, reviewed discriminating examples, and the repairs done by the user while interacting with the debugger.

Condition-1 The last fixed model, *i.e.*, M_R, should accordingly accept and reject all previously reviewed discriminating examples.

Condition-2 No more mutation is possible. Every mutation is killed by a previously reviewed discriminating example.

Condition-3 If the model is in checking mode and follows the form of $Y \Rightarrow P$, the check must not return any counter-example.

Since all mutations are done by mutation operators called from DESP, DEBUG implicitly confirms that all mutations are killed once it gets NoDetectedBug from DESP.

5.2.2 Search Procedure

Discriminating examples search procedure (DESP) is the heart of MARGAUX. Given libraries of patterns and a model as a set of constraints and a set of relations, DESP iterates over the relations, finds an applicable library of patterns with respect to the relation, and interchangeably calls two mutation operators. The operators mutate the model, generate discriminating examples, and interpret the review responses in order to discover any deviation from intentions. After each call to the operators, DESP receives a ternary tuple, consisting of accepted and rejected discriminating examples, as well as a name for any bug discovered. If the mutation operator does not find a bug, reviewed discriminating examples are kept aside for further functions, such as killing new mutations or checking termination conditions. DESP returns NoBugDetected, if neither mutation operator finds a bug.

DESP checks two special cases at the beginning and end. Over Lines 1-4, it checks whether the model is consistent with previous examples reviewed by the user. Inconsistency

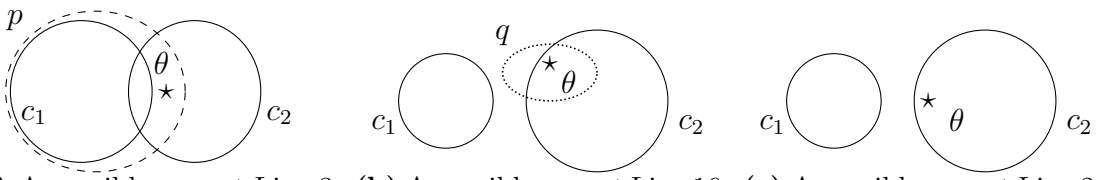
Algorithm 3: DESP(Discriminating Examples Search Procedure)

Input: List of libraries of debug patterns Π, each library is a tuple $\langle \Omega, \mathcal{U}, \mathcal{G} \rangle$.
Input: Model $N = \langle C, P, R, B \rangle$, where Constraints $C = \{c_1, \ldots, c_n\}$, Property P, Relations $R = \{r_1, \ldots, r_m\}$, and Bounds $B = \{b_1, \ldots, b_m\}$.
Output: A tuple including accepted discriminating examples, rejected discriminating examples, and a potentially discovered bug.

```
1  if ∃e⁻ ∈ E⁻|e⁻ ⊨ ⋀C then
2  |  return ⟨E⁺, E⁻ ∪ e⁻, Underconstraint⟩
3  end
4  if ∃e⁺ ∈ E⁺|e⁺ ⊭ ⋀C then
5  |  return ⟨E⁺ ∪ e⁺, E⁻, Overconstraint⟩
6  end
7  for r_j ∈ R do
8  |  for π ∈ Get_Applicables(Π, r_j) do   ▷ Get_Applicables finds which libraries are applicable
   |   to this relation.
9  |  |  α, β ← ⟨E⁺, E⁻, Initialized⟩                                  ▷ Initialization
10 |  |  repeat
11 |  |  |  strengthening ← Magic()   ▷ Magic() randomly returns True/False in this context.
12 |  |  |  if strengthening then
13 |  |  |  |  α ← Strengthener_Mutation_Operator(π, r_j, C, E⁺, E⁻)
14 |  |  |  |  if α[3] = Underconstraint then
15 |  |  |  |  |  return ⟨α[1], α[2], Underconstraint⟩
16 |  |  |  |  end
17 |  |  |  |  E⁺ ← E⁺ ∪ α[1]
18 |  |  |  else
19 |  |  |  |  β ← Weakener_Mutation_Operator(π, r_j, C, E⁺, E⁻)
20 |  |  |  |  if β[3] = Overconstraint then
21 |  |  |  |  |  return ⟨β[1], β[2], Overconstraint⟩
22 |  |  |  |  end
23 |  |  |  |  E⁻ ← E⁻ ∪ β[2]
24 |  |  |  end
25 |  |  until ¬(α[3] = NoBugDetected ∧ β[3] = NoBugDetected)
26 |  end
27 end
28 if P ≠ ∅∧Satisfiable(¬P ∧ ⋀C) then
29 |  return ⟨E⁺, E⁻∪Find_Near-hit ((⋀C) ∧ ¬P, P) , Underconstraint⟩
30 end
31 return ⟨E⁺, E⁻, NoBugDetected⟩
```

might occur if the repair of a bogus model is imperfect and violates previous repairs. Finally, `DESP` checks a special case for the checking mode. If no previous mutations have generated a discriminating example to reveal the invalidity of the check statement, then

Figure 5.7 Showing how BORDEAUX should find examples for each mutation done by `Weakener_Mutation_Operator` . The mentioned notations at each line are accordingly reflected in the corresponding line. Briefly speaking, a given model consists of c_1 and c_2. p and q, which are outlined by dashed and dotted circles, represent the strongest synonym and weakest antonym of c_1. Star signs, labeled by θ, represent the instances found by BORDEAUX. Except in the last diagram, c_1 is picked for analysis.



(a) A possible case at Line 3 in Algorithm 4

(b) A possible case at Line 16 in Algorithm 4

(c) A possible case at Line 26 in Algorithm 4

`DESP` retrieves an example consistent with the premise but inconsistent with the conclusion. If such an example exists, `DESP` reports an underconstraint bug.

Weakener Mutation Operator

`DESP` uses the weakener mutation operator in order to test hypotheses whether a given model has overconstraint bug. The operator changes the model to generate discriminating examples inconsistent with the current model. If the user accepts such discriminating examples, MARGAUX reveals an overconstraint bug as well as the constraints causing the bug. Using the synonyms and antonyms, the debugger is able to report the meaning of unintentionally excluded instances.

To generate such a discriminating example, `DESP` selects a relation and calls the weakener mutation operator. Having the relation, the operator, depicted in Algorithm 4, picks one constraint of the model and finds all of its strongest synonyms that are consistent with the rest of the model's constraints. The operator calls BORDEAUX to find examples close to borders distinguishing instances having the same synonym but consistent or inconsistent with the constraint.

Figure 5.7a shows the border and a discriminating example expected by the operator. Once the user accepts or rejects the discriminating example, the operator returns, regardless of her answer, so that `DESP` continues other mutations or terminates. If the operator does not find a surviving mutation that generates a discriminating example, it tries

Algorithm 4: Given a set of constraints, `Weakener_Mutation_Operator` finds the simulacrum of each one, mutates the simulacrum, generates discriminating examples, and likely reveals an overconstraint bug if the useri does not want the example. Using the history of reviewed discriminating examples, the operator kills mutations. `Find_Near-miss` is a call to BORDEAUX [§4].

Implicit: Model $N = \langle C, R, B \rangle$
Input: A library of debug patterns $\pi = \langle \Omega, \mathcal{U}, \mathcal{G} \rangle$
Input: A relation r, where $r \in R$
Input: A set of constraints C, where $C \in N.C$
Input: A set of accepted Discriminating Examples E^+
Input: A set of rejected Discriminating Examples E^-
Output: A tuple including accepted discriminating examples, rejected discriminating examples, and a potentially discovered bug.

for $c_i \in C$ **do**
  for $p \in$ `SS_C(`$\pi, c_i, \bigwedge(C - c_i), r$`)` **do**
    $\theta \leftarrow$`Find_Near-miss (`$\bigwedge(C), \bigwedge(C - c_i) \wedge \neg c_i \wedge p$`)`
    if $\theta \in E^+ \cup E^-$ **then**
      continue
    else if $\theta \in M_I$ **then**`/* Ask the user */`
      return $\langle E^+ \cup \theta, E^-, Overconstraint \rangle$
    else
      return $\langle E^+, E^- \cup \theta, NoBugDetectedSoFar \rangle$
    end
  end
  if ¬`Satisfiable(`$\bigwedge C$`)`∧ `Satisfiable(`$\bigwedge(C - c_i)$`)` **then** `/*` c_i `is inconsistent with the rest of constraints */`
    for $q \in$ `WA_C(`$\pi, c_i, \bigwedge(C - c_i), r$`)` **do**
      $\theta \leftarrow$ `Find_Near-miss (`$c_i, \bigwedge(C - c_i) \wedge q$`)`
      if $\theta \in E^+ \cup E^-$ **then**
        continue
      else if $\theta \in M_I$ **then**
        return $\langle E^+ \cup \theta, E^-, Overconstraint \rangle$
      else
        return $\langle E^+, E^- \cup \theta, NoBugDetectedSoFar \rangle$
      end
    end
  end
  $\theta \leftarrow$ `Find_Near-miss (`$c_i, \bigwedge(C - c_i) \wedge \neg c_i$`)`
  if $\theta \in E^+ \cup E^-$ **then**
    continue
  else if $\theta \in M_I$ **then**
    return $\langle E^+ \cup \theta, E^-, Overconstraint \rangle$
  else
    return $\langle E^+, E^- \cup \theta, NoBugDetectedSoFar \rangle$
  end
end
return $\langle E^+, E^-, NoBugDetected \rangle$

antonyms. Depicted in Figure 5.7b, the operator considers antonyms of the constraints that are consistent with other constraints. Provided the user accepts such a discriminating example that is consistent with the antonym and other constraints means that the model has an overconstraint bug and the constraint is the root of the problem. That is, a repair has to turn the antonym into at least a partial synonym. Last but not least, one way to make a mutation is to weaken the constraint by excluding it. The operator removes the constraint and reviews with the user a discriminating example that conceptually looks like the one depicted in Figure 5.7c.

Ultimately, the operator returns NoBugDetected once all discriminating examples are killed by previous examples or rejected by the user. Before inquiring about a discriminating example, the operator also checks whether the example has already been reviewed. In this way, the operator kills mutations that would otherwise generate redundant examples.

Strengthener Mutation Operator

For finding underconstraint bugs, DESP uses the strengthener mutation operator, depicted in Algorithm 5, to generate discriminating examples consistent with the given model. However, the examples are intentionally consistent with a synonym of the model but inconsistent with its stronger forms. If the user rejects such a discriminating example, the model should be repaired in order to exclude instances with characteristics similar to those of the synonym, not its stronger forms.

The operator navigates the implication graph in order to make a mutation consistent with the model and one of its partial synonyms, but inconsistent with a property stronger than the synonym.

As diagram Figure 5.8a shows, the operator starts from the weakest partial synonym of the model. For the next mutation, the operator picks a partial synonym of the model stronger than the previous one, as in the case depicted in Figure 5.8b. The operator generates a sequence of discriminating examples by traversing through the implication graph. It starts from sinks, or close to them, and walks towards the graph sources. At each step, the operator checks whether a synonym is enough or should be strengthened into a stronger synonym. We found Depth-First-Traversal quickly approaches stronger patterns, thereby causing bug reveal with fewer interactions.

Algorithm 5: Once `Strengthener_Mutation_Operator` finds the simulacrum of a given constraint, it generates discriminating examples by systematically tightening the partial synonyms. If the user does not want the example, the operator returns *Underconstraint* bug; otherwise, it returns *NoBugFoundSoFar*. The operator returns *NoBugFound* provided it cannot generate any discriminating example. `Find_Near-hit` is an interface to BORDEAUX [§4].

Implicit: Model $N = \langle C, R, B\rangle$
Input: A library of debug patterns $\pi = \langle \Omega, \mathcal{U}, \mathcal{G}\rangle$
Input: A relation r, where $r \in R$
Input: A set of constraints C, where $C \in N.C$
Input: A set of accepted Discriminating Examples E^+
Input: A set of rejected Discriminating Examples E^-
Output: A tuple including accepted discriminating examples, rejected discriminating examples, and a potentially discovered bug.

```
1  if Satisfiable(⋀C) then
       /* Initializing a list.                                                    */
2      ℒ ← (⋀C)
       /* A set to keep done properties.                                          */
3      Ψ ← ∅
4      while length(ℒ)≠ 0 do /* More mutations are left with properties implying p  */
5
6          p ← tail(ℒ)
7          if length(ℒ) = 1 then
               /* Choose a property from the set of weakest partial synonyms of the constraint. */
8              choose q ∈ WpS(π, ⋀C,r) −Ψ
9          else
10             choose q ∈ pS_S(π, ⋀C, p, r) −Ψ
11         end
12         if q = ∅ then /* No stronger property than p or all were visited      */
13
14             Ψ ← Ψ∪ remove_tail(ℒ)
15         else
16             ℒ ← ℒ + q
17             θ ← Find_Near-hit (⋀C ∧ ¬q ∧ p, ⋀C ∧ q)
18             if θ ∈ E+ ∪ E− then
19                 continue
20             else if θ ∈ M_I then
21                 return ⟨E+, E− ∪ θ, Underconstraint⟩
22             else
23                 return ⟨E+ ∪ θ, E−, NoBugFoundSoFar⟩
24             end
25         end
26     end
27 end
28 return ⟨E+, E−, NoBugDetected⟩
```

Regulating nondeterministic choices

DESP and the mutation operators make some non-deterministic decisions to generate discriminating examples. We categorize the nondeterministic choices into the following groups.

Figure 5.8 `Strengthener_Mutation_Operator` systematically navigates the implication graph to generate different discriminating examples consistent with a given model. The Venn diagrams represent possible areas where the operator can find discriminating examples close to their borders at Line 8. The notations at each line are accordingly reflected in the corresponding line. The left diagram shows a situation where the operator picks C to be p at Line 10. The right diagram shows the next mutations, where p is not the weakest partial synonym.

(a) A possible case at Line 8 in Algorithm 5 **(b)** A possible case at Line 10 in Algorithm 5

Some are applicable only for one selection, some for more:

Relation selection `DESP` has to sequentially choose relations to find simulacra.

Constraint selection `Weakener_Mutation_Operator` selects a constraint to weaken at each call.

Strengthening or weakening Provided both mutation operators are applicable, `DESP` has to choose one of them at each iteration.

Synonym/Antonym selection If multiple synonyms and antonyms exist, the mutation operators must pick one so as to generate a discriminating example.

Although, the number of mutations is finite, choosing appropriate choices at each decision point affects how soon MARGAUX generates a discriminating example revealing a bug. In particular cases, procedures can reason about the next steps and generate a discriminating example revealing a user's intention. We call these types of reasoning *heuristics* and apply them whenever possible. Provided heuristics are not hit; the procedures nondeterministically choose their paths to generate examples. After running experiments, we found some heuristics that lead to a quicker bug reveal (summarized in Table 5.3). In the following sections, we frequently refer to these heuristics and show how they affect the search procedure.

Table 5.3 The way MARGAUX decides to deal with its non-deterministic choices affects how many discriminating examples should be reviewed until one reveals a bug. For different decision points, we have found heuristic techniques that could improve the debugger's effectiveness.

Heuristic	Affected Decision	Description
Heuristic-1	Strengthening or weakening	If a given model is unsatisfiable, then select weakening
Heuristic-2	Relation selection	Although a model is satisfiable, some of its relations might not have any tuple. Pick them first.
Heuristic-3	Relation selection	The relation that is referenced more in a model or a constraint should be analyzed first.
Heuristic-4	Strengthening or weakening	Constraints with many synonyms might cause an over-constraint issue.
Heuristic-5	Synonym/Antonym selection	If the user rejects an example from the weakener mutation operator, the synonym or antonym used in the mutation should be selected after other synonyms for the next mutations.
Heuristic-6	Strengthening or weakening	If a constraint is inconsistent with approximations of another constraint, then the former constraint might be overconstrained and need to be weakened. Hence, weakening the former constraint has a higher priority than the latter one.
Heuristic-7	Strengthening or weakening	If a relation is always empty, then weaken constraints.
Heuristic-8	Synonym/Antonym selection	For selecting a property from a set of properties implying a given temporal ternary property, properties with emptiness constraints are selected first, then properties staticness, strictness expansion/contraction, and finally the direction of expansion.
Heuristic-9	Constraint selection	If the model is unsatisfiable, work only on the constraint existing in UNSAT-CORE.
Heuristic-10	Constraint selection	If a selected constraint does not lead to a bug reveal, move on to another constraint.

Soundness and Completeness

The debugging iteration is sound but not complete in terms of the generated examples and bugs that are found through interactions with users.

Soundness. `DEBUG`, depicted in Algorithm 2, is sound to generate examples and non-examples. A instance produced as an example should be consistent with the model. The instance should be inconsistent with the model if the algorithm finds it is a non-example. The algorithm interchangeably calls the mutations operators to produce near-border examples for the mutations. The debugger is sound, if `Weakener_Mutation_Operator` produces non-

examples and `Strengthener_Mutation_Operator` produces examples of a given model. `Weakener_Mutation_Operator` calls BORDEAUX to produce `Find_Near-miss` examples with respect to the mutations. The returns of these calls are consistent with the second parameter passed to `Find_Near-miss`. As the second arguments of these calls are inconsistent with the model, all the calls return non-examples. That is, as depicted in Figure 5.7, the `Find_Near-miss` examples are always inconsistent with one of the conjoined constraints. On the other hand, the call to BORDEAUX in `Strengthener_Mutation_Operator` returns a `Find_Near-hit` example that is consistent with the first parameter. As the first argument is conjoined to the model's constraints, the result is an example consistent with the model. Shown in Figure 5.8, such discriminating examples are always consistent with the conjunction of all constraints in the model.

Completeness. `DEBUG` is not complete for finding all `Find_Near-hit` and `Find_Near-miss` examples as well as revealing all bugs. Using BORDEAUX, MARGAUX picks only near-border examples, so that examples that are far from the conceptual border are not generated and checked by the user. Moreover, MARGAUX limits its generation of `Find_Near-miss` examples to only those consistent or inconsistent with the given model's mutations. Therefore, if the mutations do not cover all possible `Find_Near-miss` examples, then MARGAUX will not logically cover all `Find_Near-miss` examples unless a mutation or a combination of mutations cover all `Find_Near-miss` examples. Since `DEBUG` does not generate all possible examples, it might skip examples showing underconstraint or overconstraint issues. MARGAUX is also not complete to find all bugs.

Although `DEBUG` is not complete and does not generate all possible examples, it tries to find the most likely examples for revealing bugs. To illustrate the compromise that MARGAUX makes, consider Alloy Analyzer, which generates all examples. The examples reveal only underconstraint issues, but the user has to continue exploring them until she finds an appropriate one.

5.2.3 Localization

When the user finds that a discriminating example shows a divergence between her intention and the expressed model, she repairs the model by adding constraints to it to fix underconstraints or removing constraints from it to eliminate overconstraints. MARGAUX uses the reviewed discriminating examples and the Minimum Unsatisfiable Core (MUC) finder [101] to localize the constraints that need to be fixed for an overconstraint bug. Given an unsatisfiable model, MUC returns a subset of constraints that removing any of them makes the model satisfiable.

To find the part of the model causing an intended discriminating example to be excluded from the model, the first step is to turn the example into some form of constraint. MARGAUX turns the example into the existential form. In the existential form, a statement with an existential quantifier ensures that the atoms corresponding in the tuples of the example's relations exist in the universe of discourse, and that all interpretations of the relations will be the same as the example. For instance, given an example of a typical model of a singly-linked list as '$\mathsf{Node}=\langle \mathsf{Node0},\mathsf{Node1}\rangle,\mathsf{next}=\langle \mathsf{Node0}\rightarrow\mathsf{Node1}\rangle$', the existential form becomes: '**some disj** Node0,Node1:Node| Node=Node0+Node1 **and** next=Node0 →Node1'. That is, any valuations of relations Node and next will be a list with two connected nodes.

Next, the debugger finds statements that prevent the discriminating example. As the model excludes such an example, there must be some statements that are inconsistent with the existential form of the reviewed discriminating example. If the model has partial overconstraint, then the debugger conjoins the model with the existential form. Given such a conjoined model, MUC finder returns statements inconsistent with the statement expressing the example. On the other hand, if the model has a total overconstraint, the debugger finds the statements in the MUC and iterates over them to check whether one is inconsistent with the existential form of the example.

As Alloy Analyzer approximates a MUC within a given inconsistent model, the granularity of the reported statements depend on the complexity of the model. For further localizations, MARGAUX might use two syntactical techniques: *quantifier unrolling* and *function and predicate inlining*. The former technique transforms the corresponding universal quantifiers to conjunction statements, and existential quantifiers to disjunction statements [101]. The latter technique replaces the corresponding calls of predicates and functions with their body. As Alloy models are analyzed in a finite universe of discourse and recursions are forbidden, then both techniques terminate in a finite number of steps.

As the user needs to add constraints to eliminate unintended discriminating examples, MARGAUX does not provide a particular functionality to pinpoint what statements should be fixed.

5.2.4 Fix suggestion

As discriminating examples assist the users to understand the model and find underconstraint or overconstraint bugs, MARGAUX can also suggest some fixes for particular kinds of bugs. Provided the debugger finds synonyms and antonyms for a given model or its constraints, then it might be able to suggest fixes. In a case of an overconstraint bug, if the debugger has generated a discriminating example using the synonym of a constraint,

i.e., the case sketched in Figure 5.7a, then a fix would be replacing the constraint with its synonym. For an underconstraint bug, provided the user rejects a discriminating example, then one fix suggestion is to conjoin the partial synonym, what q in Algorithm 5 refers, to the model. For both bugs, the debugger can also suggest what property should be strengthened or weakened from the model to fix the bug. For example, a constraint that implies total order property should be relaxed to enforce partial order. The major development of the fix suggestion is planned as future work.

5.3 Related Work

The concept of design patterns goes back to Christopher Alexander and his colleagues, who worked on using patterns to structure buildings and cities [2]. Gamma et al. [32] applied the same concept in object-oriented software design. Software design patterns capture the elegant practices experts use in addressing particular problems and convey them in a simple and understandable form that others can follow. The structural form of design patterns facilitates mapping and understanding the best path from a user's intention to a reasonable solution.

In their seminal work, Dwyer et al. [26], proposed a pattern-based approach for encoding property specifications for finite-state verification. The patterns are temporal properties that are claimed to be useful for practitioners with a reasonable level of knowledge in formal specification and analysis. Konrad and Cheng [56] and Gruhn and Laue [36] extended Dwyer's original specification patterns to the context of real-time property specifications. Raimondi et al. [84], Kallel et al. [47], and Halle et al. [38] formulated particular specification patterns for service-based applications. Bianculli et al. [9] did a survey to evaluate how specifications patterns for service based architecture are practical in industry. Liu et al. [59] re-used the idea of specification patterns to analyze functional requirements.

By mutating a program code, the mutation testing checks whether a given test-suite is able to detect intentional changes in the code [11]. The feasibility of mutation testing relies on two hypotheses: competent programmers and coupling effect [45]. Having similar hypothetical basics, MARGAUX slightly alters a given model to discriminating examples. MARGAUX not only relies on the syntactical changes to mutate a model, it infers simulacra and systematically alters them to create mutations.

Alloy Analyzer finds arbitrary examples of a model. The sequence of the returned instances could vary from time to time. Recently, researchers [20, 78, 93] proposed techniques for mitigating the arbitrary nature of example generation. Nelson et al. [78] proposed an

extension of Alloy Analyzer, called Aluminum, that helps the user to track a scenario of examples rather than a random series of them. Given a model, their solver finds an example with a least number of tuples consistent with the model. Having augmentation operations, the user explores different examples, starting from the first example. MARGAUX differs from such exploration tools since it systematically assists users to explore examples and non-examples. Also, unlike Aluminum, the examples MARGAUX returns are not necessary absolute minimum, although they become consistent or inconsistent with a minimum number of changes.

Shapiro [95] proposed a useful debugging technique for Prolog that interactively communicates with the user to reveal three types of errors: termination with incorrect output, termination with missing output, and nontermination. His inductive algorithm corrects some revealed faults. According to the survey by Silva [97], many researchers have proposed techniques for selecting the minimum number of meaningful debugging questions mainly by focusing on strategies for traversing a Prolog program's execution tree. Similar to the algorithmic debugging technique and its extensions, MARGAUX assists Alloy users to find incorrect or insufficient constraints. Both debuggers rely on interactions with an oracle to reveal potential bugs. Since the execution of an Alloy model is abstracted from independent solvers, MARGAUX does not use the execution tree concept in the way the other does. Instead, its way of decomposing constraints and traversing them to find underconstraint and overconstraint issues is conceptually related to top-down and bottom-up execution tree traversals.

Shlyakhter et al. [96] proposed using unsatisfiable core constraints for debugging Alloy models. Once an Alloy model is translated into a Boolean formula, their technique, implemented in Alloy Analyzer 3, first uses a SAT-solver to find an unsatisfiable subset of the Boolean formula, then translates the subset back into the model's statements. Not only is finding a minimum unsatisfiable core of an inconsistent Boolean formula complex, [23, 39, 63, 108], translating from an unsatisfiable core to Alloy constraints is challenging.

The technique by Torlak et al. [103] improved Alloy Analyzer 4 to approximate the minimum unsatisfiable core in an inconsistent Alloy model. The technique minimizes the unsatisfiable core at a higher level than with Boolean logic. It also recycles inferences that the underling SAT-solver made in finding the core. These inferences are recognized from Boolean proof provided by the solver. Even though having an unsatisfiable core helps users to determine which constraints are in conflict, the code does not exactly clear up where and why a problem originated. In practice, the root of the problem may be a statement other than the constraints reported as an unsatisfiable core. MARGAUX frequently uses the unsatisfiable core, if it exists, to narrow its search for discriminating examples in heuristics.

Reiter [90] proposed the theory of diagnosis from first principles. A diagnosis depends on the conflict between observation of how a system actually behaves and description of how a system's components are expected to behave. A minimum diagnosis is a conjecture for removing the least faulty components in order to make the whole system description consistent. Reiter proposed an algorithm for determining minimal diagnosis. The algorithm relies on the minimal conflict set in a given formula. Felfernig et al. [29] improved the diagnosis algorithm by putting a total order on the given inconsistent constraints. Xiong et al. [111] designed and implemented an algorithm that automatically generates a range of fixes for an inconsistent configuration specified by quantifier-free constraints. The algorithm uses Reiter's theory of diagnosis for selecting the minimal set of variables to be fixed. Nentwich et al. [79] proposed a repair framework for inconsistent constraints such as in UML and EJB deployment description. The constraint language is First-Order Logic but restricted to $=$ and $\neq$ comparison operators. Reder and Egyed [89] proposed formulating a specification fixing transformation as a set of fix rules that show some change scenarios. A fix will be interactively applied after any requested changes. They used heuristics to manage many fixes in practice. Van Der Straeten et al. [107] used Kodkod [102] for automatically generating consistent models for an inconsistent specification that needs to be resolved. Junker [46] designed an approach for revealing a minimal conflict set based on a divide and conquer search strategy. The approach is used in a tool for fault explanation and repair. Groce et al. [35] proposed a technique to make use of valid solutions close to counter-examples for assisting users in understanding and isolating errors. In the techniques that are based on, or similar to, Reiter's theory of diagnosis, the debugger knows of the problem and tries to explain its roots as well as suggesting repairs. Margaux takes one step back and assists Alloy users in finding potential issues. To better understand potential faults, examples are the core of debugging interactions in Margaux.

Researchers have developed different debugging techniques for analyzing models expressed in temporal logic. Könighofer et al. [55] developed a debugger for LTL formal specifications using a counterstrategy that is a finite state strategy whereby a specification cannot be fulfilled if the environment adheres to it. The tool first provides input; the user tries to provide output; then it uses counterstrategy to find inputs such that the system does not fulfill the specification. The interaction is continued until the user fails and understands the problem. The debugger suggests no fix, but helps clarify the problem. Beer et al. [8] and Chechik and Gurfinkel [17] proposed an interactive visualization technique for understanding counterexamples. In [8], the technique uses the structural causality between LTL properties to display information on a failed property and related constraints. The debugging technique, explained in [17], incorporates the structure of a specification and its expected CTL properties to annotate counter-examples with proofs

that explain the model checking result. When debugging database queries, users are interested in non-answer tuples that were expected to be returned by the database engine but which were not. In this context, a provenance defines a piece of information stating why a query execution does not return a tuple. Huang et al. [40] developed a method to generate non-answer tuples and the provenances explaining their failure to appear as output.

Chan [16] proposed a technique to find a propositional formula for a temporal logic query. Gurfinkel et al. [37], extended the technique to find propositional formulas for multiple placeholders in a query. Using different temporal logical queries, the user can explore different temporal aspects of a model, expressible in Kripke structures.

5.4 Summary

Using discriminating examples, Alloy users review focused instances; accepting or rejecting them can lead to bug detection. This way, Margaux tests different hypotheses whether the model differs from the user's intention. Interacting with the user, the debugging procedure can find subtle partial over constraint bugs that other tools such as UnSAT-Core are difficult to find.

To produce discriminating examples, Margaux infers simulacra of a given model' constraints and generates near-border examples for the model's mutant. We have used the concept of patterns to abstract the simulacra of a given model. With the predefined relations between patterns, Margaux mutates the model and produces examples and non-examples of the model. By producing examples for strengthened or weakened properties, Margaux can test whether the model suffers from a overconstraint or underconstraint bug. We have designed the search procedure to automatically perform these steps and generate discriminating examples. The search procedure is plugged with heuristics to produce fewer discriminating examples for finding the bug.

Chapter 6

Dining Philosophers Case Study

In this chapter, we demonstrate core evidence for the feasibility of Margaux in producing insightful examples and non-examples and utility of discriminating examples in revealing bugs. The evidence stems from studying Margaux to debug two non-trivial bogus Alloy models; therefore, we have developed a prototype of Margaux to infer its simulacrum and constituent constraints. It implements *Discriminating Example Search Procedure* (`DESP`) for producing discriminating examples. To show the feasibility, we show how effective the prototype finds discriminating examples. For indicating the utility of Margaux, we provide evidence on how Margaux differs from other state-of-the-art tools to produce discriminating examples revealing bugs in an important historical case study.

The first studied model focuses on encoding and simulating a Singly-linked List in Alloy. This model has been used for years to teach modeling with Alloy to engineering students. Notwithstanding the simplicity of the concept, many students commonly make similar mistakes without noticing the consequences. By studying a typical model that such students usually specify, we will show how Margaux can reveal such a bug.

The other case is a model of Dijkstra's dining philosopher's problem that was shipped with Alloy Analyzer for several years. This model had a partial over-constraint bug that went undetected until 2012. The bug was originally discovered — and fixed — when a translator from Alloy to the KeY theorem proving system was written [105]. However, the depth of reasoning required to fix this bug was not adequately explored in previous publications. In Section 6.2, we demonstrate that pattern-based debugging can guide the user through this complex and multi-step reasoning.

In addition to demonstrating how Margaux can assist Alloy users to reveal bugs, we use the studied models to evaluate whether using heuristics decreases the number of

discriminating examples. Also, we analyze the extra time needed for analyzing the simulacrum of the model's constraints and selecting the best heuristics for non-deterministic choices in MARGAUX's procedures. The results, presented in Section 6.3, show us how the extensions and implementations should be directed in the next steps.

In another study, Section 6.4, we investigate how Alloy Analyzer [43] and Aluminum [78], two state-of-the-art tools for Alloy models, could produce insightful discriminating examples. Dijkstra's dining philosophers model is our base case to study the efforts that users should spend to reach the examples revealing bugs.

Sections 6.1 and 6.2 contain case studies on Singly Linked List and Dijkstra's dining philosophers models. Section 6.3 shows the effectiveness of MARGAUX's heuristics. Section 6.4 includes comparisons between MARGAUX and two state-of-the-art Alloy analyzing tools. Section 6.5 summarizes this chapter.

6.1 Singly Linked List

We present an example of how a bug could emerge in an Alloy model, yet not be easily detected by convention analyzing tools. Then, we show how MARGAUX assists users to effectively examine the model in order to detect unintended corner-sides. We frequently use this example in teaching Alloy modeling to engineering students. We ask them to model a Singly Linked List data structure and check some of its properties. In our experience, the Alloy model, expressed in Figure 6.1, is typical of what students come up with when first learning Alloy. Although the model looks to be short and straightforward, there is a subtle bug in the entities' declarations and corresponding constraints. The Alloy Analyzer claims that the model satisfies the property at Line 5, and the user is happy with her first exercise.

A typical model of Singly Linked Lists, such as the one depicted in Figure 6.1, consists of a set of nodes and connections between them, represented by the signature Node and the relation next. A list has a head, and the user encodes it as a special node, called Head, at Line 2. Such modeling includes any form of linked lists, such as cyclic or acyclic. To exclude unwanted forms of linked lists, *i.e.*, any form other than Singly Linked Lists, the user adds the noLoop constraint at Line 4. Clearly, from the constraint name, there is no loop in the list, provided no path exists to connect an atom of Node to itself. The user expects such a constraint to suffice in expressing a Singly Linked List, but she might be asked whether a disconnected list exists or all nodes are reachable from one node. She writes another constraint, at Line 5, and checks whether having noLoop implies the reachability.

Figure 6.1 A bogus Alloy code trying to express modeling of a *Singly Sorted Linked List* in Alloy.

```
1 sig Node {next: Node}
2 one sig Head extends Node {}
3
4 pred noLoop {all n: Node| !n in n.^next))}
5 pred reachable {one n: Node| Node = n.*next)}
6
7 check {noLoop implies reachable}
```

Alloy Analyzer checks the statement at Line 7, and no counter-example means the hypothesis is valid in a finite universe. Astute users might be skeptical about the hypothesis at the beginning and be surprised by the result. Indeed, their understanding is correct and an acyclic Singly Linked List is not necessarily a connected list. The model faces an overconstraint issue, and it partially expresses what the user intended to model. The antecedent at Line 7 avoids any instance of a Singly Linked List that contains a tuple of next. Since the antecedent models only a part of the user's intentions, the property in the conclusion is partially implied from the antecedent. The user might be able to avoid such an illusive result by simulating antecedents first. For this particular example, the antecedent models only a Singly Linked List with no atoms. Now, the question is how the user should find such a subtle bug and identify the root of the problem.

Before stepping into DEBUG algorithm, we would like to clear up the syntactic desugaring happening for structural declarations. Besides connecting entities, a *relation* definition also contains declarative constraints that formulate multiplicities. Such constraints are defined as *declarative constraints* [42] and are paired with the Alloy formula syntax, called *declarative formula*. From the model in Figure 6.1, the declaration constraint next restricts every atom of Node to be in a relation with exactly one atom of Node. The declarative formula for next is defined below. We consider declarative formulas as inputs to DEBUG algorithm and mention them as needed.

pred declarativeFormulaForNext {**all** n: Node| **one** n.next}

At the first step, DEBUG algorithm takes the model, as shown in Figure 6.1, and converts it into the form ⟨{noLoop, declarativeFormulaForNext}, {Node, next}, *default*⟩. The first part is a set of constraints; the next is a set containing relations, and the last carries the information about the scopes of unary relations. In addition to the model, DEBUG passes the libraries of debug patterns, reviewed examples–at this point empty– to Discriminating

Example Search Procedure, *i.e.*, DESP, to generate discriminating examples and reveal potential bugs.

At the beginning, DESP picks next as the only choice for relations. Considering Heuristic 7, DESP decides to call Weakener_Mutation_Operator in order to find an overconstraint bug. Given the set of constraints, Weakener_Mutation_Operator randomly chooses declarativeFormulaForNext for weakening. At Line 2, Weakener_Mutation_Operator finds that the partialFunction pattern over the relation next is the strongest synonym implied from declarativeFormulaForNext, which is consistent with other constraint, *i.e.*, noLoop. It weakens the constraint declarativeFormulaForNext by replacing it with partialFunction. At Line 3, DESP asks BORDEAUX to generate a discriminating example, and the example finder returns the following discriminating example:

Node	next
Head_0	- -

The discriminating example is a Singly Linked List with one node. It looks like a legitimate list, and the user marks it as *intended*, as it was kept away by declarativeFormulaForNext. At Line 20, Weakener_Mutation_Operator returns an *overconstraint* bug as well as the reviewed discriminating example. We expect that the user will notice the root of the problem once she reviews the discriminating example and sees the constraint causing the issue.

In a few words, declarativeFormulaForNext enforces every atom of Node to connect to exactly *one* atom of Node. On the other hand, noLoop does not allow any cyclic path. In a finite universe, an atom of Node cannot exist without pointing to another atom. The discriminating example confirms this hypothesis, and the user repairs the issue by weakening the declarative constraint of next as follows:

sig Node {next: **lone** Node}

Once the user finds the bug, she could repair it and stop MARGAUX. Or, she could let DEBUG examine whether the repair aligns with her intentions. Termination conditions, expressed in Section 5.2.1, are responsible for the examination. They confirm property checking; if it exists, properly passes, and all reviewed examples are consistent with the new changes. Once the user has repaired the declarative constraint, DEBUG finds that the new model passes Condition-1, but fails on Condition-2. The latter condition suspects that a surviving mutation might reveal another bug. The bug might occur become a constraint

is stronger than the intention or the repair lets unintended instances accidentally emerge. Therefore, DEBUG calls DESP another time to see if any surviving mutation reveals another bug.

At this call, DESP can select either mutation operator. If it decides to weaken the model, Weakener_Mutation_Operator has to weaken one of the constraints. The only Synonym for declarativeFormulaForNext is functional. Since it is a sink in the implication graph, it cannot be weakened. Alternatively, noLoop's synonyms is acyclic. Mutating the synonym to its weaker forms, irreflexive and antisymmetric, leads to different discriminating examples that are rejected by the user. For instance, a mutation with antisymmetric generates the following example:

Node	next	
$Head_0$	$Head_0$	$Head_0$

At this step, the call to Weakener_Mutation_Operator leads to, at most, one discriminating example being rejected by the user. This example is added to the other rejected examples, and DESP continues for the next mutations. Unlike noLoop, DESP cannot weaken declarativeFormulaForNext, because the constraint's approximation is functional which has no successor in the implication graph.

As the model is consistent, DESP can also strengthen the model. The operator explores various sides of the model consistent with different synonyms. The weakest partial synonyms consistent with the model are {antisymmetric, irreflexive, functional, injective, weaklyConnected, transitive}. However, only the last two survive, and the others are unsatisfiable or killed by previously reviewed discriminating examples. For either synonym, the mutation reveals different bugs. As the operator returns after finding a bug, the next bug should be revealed in the next call. The following example is the result of mutating the model using weaklyConnected:

Node	next	
$Head_0$	-	-
$Node_0$	-	-

The user rejects the discriminating example because the list is not connected. She struggled a little and finds the following constraint in order to ensure all nodes are connected.

pred connected {**one** n:Node| **no** n.next **and all** n':Node−n| **one** n'.next}

Once the user repairs the disconnected list issue, DEBUG algorithm checks the termination conditions another time. Similar to the previous iteration, Condition-2 fails and DEBUG calls DESP in order to review any surviving mutations. They might have survived from the last iteration or been introduced by the last repair. At this iteration, DESP finds that the new constraint's (*i.e.*, connected) synonym is the functional pattern over the relation next. The weakened mutation over the new model part is rejected by the user .

In the call to Strengthener_Mutation_Operator, out of six synonyms, only injective leads to a surviving mutation. The operator generates the following discriminating example.

Node	next	
$Head_0$	$Head_0$	$Node_0$
$Node_0$	$Node_1$	$Node_0$
$Node_1$	-	-

The discriminating example does not parallel with the user's intention. It shows a list with two heads. The user adds another constraint to block such unintended examples. The constraint forces all nodes except the head to have exactly one incoming link.

pred singleHead {**no** Head.~next **and all** n': Node−Head| **one** n'.~next}

After the repair, DEBUG calls DESP in order to make sure there is no surviving mutation. Through this call, all calls to Weakener_Mutation_Operator are killed, so that the operator returns NoBugDetected. Still, Strengthener_Mutation_Operator has one more surviving mutation. It finds that the model is a partial synonym of the complete pattern. The operator mutates the model by strengthening the synonym to totalOrder. The discriminating example is a valid list. None of the mutation operators mutate any more, and return noBugDetected to DESP.

At the last iteration, DEBUG finds that all reviewed discriminating examples are consistent with the last repair, so that Condition-1 is satisfied. There are no surviving mutations; thus Condition-2 also becomes true. The **check** at Condition-3 fails to find any counterexample. The user can be confident that she has browsed many meaningful discriminating examples, so that the model is correct with respect to the structural binary patterns.

6.2 Case-study: Dining Philosophers

The Alloy Analyzer ships with a model of Dijkstra's dining philosophers problem. The purpose of the model is to show that Dijkstra's criterion for ordering the mutexes prevents deadlock. Previously, another research group [105] used a theorem prover to discover that the original model (which shipped with Alloy for years) was over-constrained: deadlock was prevented because no instances were possible. They provided a fix, but did not document the complex and subtle reasoning required to create that fix. Their approach does not produce examples, and we could not find evidence of how the approach assists users in understanding, localizing, and fixing the bug [54].

Dijkstra's well-known idea is that processes will not deadlock if they grab mutexes in order. In Figure 6.2, the Alloy model is defined with the entities Process, Mutex, and State, and the relations holds and waits.

The absence of a counter-example reported at Line 46 persuades its designer that the model is correct and the solution should not lead to any deadlock. Through a number of interactions, we will show how MARGAUX can assist the user to explore various corner-sides and make sure that the model aligns with her intentions. We also briefly demonstrate how she could eventually come up with a correct and logically equivalent (although syntactically different) code to the solution previously presented in the literature [105].

The First Bug

As the given model (Figure 6.2) is in the checking mode, `DEBUG`, the main procedure of MARGAUX, splits the antecedent into a set of constraints and a set of properties. That is $C = \{$GrabbedInOrder, GrabOrRelease, lowerBoundProcess$\}$ and $R = \{$holds, waits$\}$. The relations are ternary with ordering, so that the library of temporal patterns (Section 5.1.5) is the only applicable library of debug patterns in `DEBUG` and its succeeding calls to `DESP`.

At first, call `DESP` picks holds to analyze considering Heuristic 3. Although the model is satisfiable, holds and waits are always realized to an empty set of tuples. Enforcing them to have at least one tuple makes GrabbedInOrder and GrabOrRelease inconsistent. Considering Heuristic 1, `DESP` chooses to call `Weakener_Mutation_Operator`. Due to Heuristic 9, only the two inconsistent constraints participate in the current call for mutations. The operator finds no synonyms for the constraints, but does find that the weakest antonyms of GrabbedInOrder consistent with the other constraint are ExpandRight_FirstLeftEmpty and ExpanddRight. By applying Heuristic 8, the operation picks the former antonym and generates the following discriminating example for review by the user:

Figure 6.2 A bogus Alloy model expresses modeling a solution to the classical Dinning Philosophers problem proposed by Edsger W. Dijkstra. The Alloy model has been shipped with the Alloy Analyzer module for years.

```
1  open util/ordering [State] as so
2  open util/ordering [Mutex] as mo
3
4  sig Process, Mutex {}
5  sig State {holds, waits: Process →Mutex }
6
7  pred Initial [s: State] { no s.holds + s.waits }
8  pred IsFree [s: State, m: Mutex] { no m.~(s.holds) }
9  pred IsStalled [s: State, p: Process] { some p.(s.waits) }
10 pred GrabMutex [s: State, p: Process, m: Mutex, s': State] {
11   not s.IsStalled[p]
12   m !in p.(s.holds)
13   s.IsFree[m] implies {
14     p.(s'.holds) = p.(s.holds) + m and no p.(s'.waits)
15   } else {
16     p.(s'.holds) = p.(s.holds) and p.(s'.waits) = m
17   }
18   all otherProc: Process − p | otherProc.(s'.holds) = otherProc.(s.holds) and otherProc.(s'.waits) = otherProc.(s.waits)
19 }
20 pred ReleaseMutex [s: State, p: Process, m: Mutex, s': State] {
21   not s.IsStalled[p]
22   m in p.(s.holds)
23   p.(s'.holds) = p.(s.holds) − m
24   no p.(s'.waits)
25   no m.~(s.waits) implies {
26     no m.~(s'.holds) and no m.~(s'.waits)
27   } else {
28     some lucky: m.~(s.waits) | { m.~(s'.waits) = m.~(s.waits) − lucky and m.~(s'.holds) = lucky }
29   }
30   all mu: Mutex − m | mu.~(s'.waits) = mu.~(s.waits) and mu.~(s'.holds)= mu.~(s.holds)
31 }
32 pred GrabOrRelease {
33   Initial[so/first]
34   all pre: State − so/last | let post = so/next [pre] | (post.holds = pre.holds && post.waits = pre.waits) ||
35     (some p: Process, m: Mutex | pre.GrabMutex [p, m, post]) ||
36     (some p: Process, m: Mutex | pre.ReleaseMutex [p, m, post])
37 }
38 pred GrabbedInOrder {
39   all pre: State − so/last | let post = so/next[pre] |
40     let had = Process.(pre.holds), have = Process.(post.holds) | let grabbed = have − had |
41       some grabbed implies grabbed in mo/nexts[had]
42 }
43 pred lowerBoundProcess { some Process }
44 pred Deadlock{ some Process and some s: State | all p: Process | some p.(s.waits) }
45
46 check{ (lowerBoundProcess and GrabOrRelease and GrabbedInOrder)
47   implies not Deadlock } for 5 State, 5 Process, 4 Mutex
```

	holds			waits	
$State_0$	-	-	$State_0$	-	-
$State_1$	$Process_0$	$Mutex_0$	$State_1$	-	-

The user accepts the discriminating example, so that the operator returns an *Overconstraint* issue along with the reviewed discriminating examples. Since the discriminating example is in conflict with GrabbedInOrder, the user has to repair the code.

The debugger now applies the quantifier unrolling to GrabbedInOrder, which has a universal quantifier over a set of states. The UnSAT-Core of the unrolled GrabbedInOrder with the discriminating example further localizes the problem: GrabbedInOrder is inconsistent with the first state of the discriminating example. GrabbedInOrder comprises a single implication. For that implication to be in conflict with the discriminating example it must be the case that the antecedent is satisfied and the consequent is where the conflict is. So the problem has been localized to grabbed **in** mo/nexts[had]. As had is empty, the user realizes that the problem is that the nexts function will not return any successors for the empty set (had). The user replaces this faulty expression with a conditional that returns all mutexes when had is empty and mo/nexts[had] otherwise. A repair choice is represented in Figure 6.3.

Figure 6.3 A fix for mitigating the overconstraint bug in the model from Figure 6.2

```
1 pred GrabbedInOrderFixed1 {
2   all pre: State − so/last |
3     let post = so/next[pre] |
4     let had = Process.(pre.holds), have = Process.(post.holds) |
5     let grabbed = have − had |
6       (some grabbed) ⇒
7         (grabbed in (no had implies Mutex else mo/nexts[had]))
8 }
```

Alternatively, if `Weakener_Mutation_Operator` picked GrabOrRelease first, MutateRight and ContractRight are its weakest antonyms. The operator selects the former pattern; the call to Bordeaux returns the follow discriminating example:

	holds			waits	
$State_0$	$Process_0$	$Mutex_0$	$State_0$	$Process_0$	$Mutex_0$
$State_1$	$Process_0$	$Mutex_0$	$State_1$	-	-

As the user does not want this discriminating example, the operator adds the example to the rejected examples set and searches for another one. Using Heuristic 10, the operator

switches to GrabbedInOrder, and reveals the bug. All in all, after at most reviewing *two* discriminating examples, the user finds the bug. On the other hand, if the heuristics are not considered, on average, it takes three discriminating examples for the operator to find a relevant discriminating example.

The Second Bug

`DEBUG` takes the repaired model, represented in Figure 6.3, and checks whether termination conditions are passed. Because some mutations have survived, `DESP` creates more discriminating examples for exploring other parts of the model and finding other bugs. Considering Heuristic 2, `DESP` selects the relation holds first. The model is satisfiable, so `DESP` can call both operators. If it calls for `Weakener_Mutation_Operator`, the operator finds non-examples of each constraint to be reviewed. The user rejects them, so `DESP` may call `Strengthener_Mutation_Operator` in the middle or after all calls for weakening. The user also accepts discriminating examples, once the partial synonyms ExpandRight and ExpandHeadOfRight are consecutively strengthened. Considering Heuristic 8, the next mutation strengthens the partial synonym ExpandHeadOfRight_MiddleStatic to ExpandHeaddOfRight_MiddleStatic and generates the following discriminating example:

holds			waits		
$State_0$	-	-	$State_0$	-	-
$State_1$	$Process_0$	$Mutex_0$	$State_1$	-	-
$State_2$	$Process_0$	$Mutex_0$	$State_2$	-	-
$State_2$	$Process_0$	$Mutex_2$	-	-	-
$State_3$	$Process_0$	$Mutex_0$	$State_3$	-	-
$State_3$	$Process_0$	$Mutex_1$	-	-	-
$State_3$	$Process_0$	$Mutex_2$	-	-	-

The user rejects the discriminating example, because in the fourth state, Process0 grabbed a mutex with an order lower than the order of the mutex that it grabbed in the third state. Since the order of grabbing is localized in grabbedInOrder, the code in Figure 6.3 is the first place to be repaired.

The user again repairs his previous repair in order to ensure that the order of the grabbed mutex is higher than the order of mutexes that it already has. The concept basically is conveyed by the meaning of the stronger synonym. Figure 6.4 is the user 's repair mitigating the overconstraint bug. On the other hand, without using the heuristics, a bug might be revealed only after 16 or more interactions with random choices.

Figure 6.4 Repairing the underconstraint bug in the model from Figure 6.3

```
1 pred GrabbedInOrderFixed2 {
2   all pre: State − so/last |
3     let post = so/next[pre] |
4     let had = Process.(pre.holds), have = Process.(post.holds) |
5     let grabbed = have − had |
6       (some grabbed) ⇒ (no (grabbed & mo/prevs[had]))
7 }
```

The Third Bug

Once the user has repaired the model, `DEBUG` finds that the termination conditions are not satisfied yet, and more mutations to be checked. `DESP` calls `Weakener_Mutation_Operator`, yet the surviving mutations do not generate discriminating examples acceptable to the user. The first five calls to `Strengthener_Mutation_Operator` are killed by reviewed discriminating examples. In the next two calls, the operator generates discriminating examples by strengthening the model's partial synonyms, ExpandRight and ExpandRightHeadOfRight, to their mutations ExpandRight_MiddleStatic and ExpandRightHeadOf_MiddleStatic, respectively. The user accepts a discriminating example returned by Bordeaux for the former mutation, but rejects the one, as in the following example, for the latter mutation:

holds			waits		
$State_0$	-	-	$State_0$	-	-
$State_1$	$Process_0$	$Mutex_0$	$State_1$	-	-
$State_2$	$Process_0$	$Mutex_0$	$State_2$	-	-
$State_2$	$Process_1$	$Mutex_1$	-	-	-
$State_3$	$Process_0$	$Mutex_0$	$State_3$	$Process_1$	$Mutex_0$
$State_3$	$Process_1$	$Mutex_1$	-	-	-
$State_4$	$Process_1$	$Mutex_0$	$State_2$	-	-
$State_4$	$Process_1$	$Mutex_1$	-	-	-

The user rejects the discriminating example since a process grabbed a mutex with an order lower than the order of a mutex that it already has.

This Bug occurred because the order of grabbed mutexes is not bound per each process. By quantifying over atoms of Process and restricting their mutex grabbing, the user can repair the model, making it similar to that in Figure 6.5. With heuristics, the user reviews at least two discriminating examples to identify the bug. Alternatively, she has to review 4 examples at most. If Margaux does not employ heuristics, the desired discriminating

Figure 6.5 Repairing the underconstraint bug in the model from Figure 6.4

```
1 pred GrabbedInOrderFixed3 {
2   all p: Process |
3     all pre: State − so/last |
4       let post = so/next[pre] |
5       let had = p.(pre.holds), have = p.(post.holds) |
6       let grabbed = have − had |
7         (some grabbed) ⇒ (no (grabbed & mo/prevs[had]))
8 }
```

example is found after 4 mutations on average.

The Fourth Bug

After the user has repaired the model, `DEBUG` again checks the termination conditions for the model in Figure 6.5. Four mutations still survive for the relation holds. Once the user accepts or rejects them without any unexpected answer, `DESP` starts mutating over waits. If it chooses to call `Weakener_Mutation_Operator`, the operator generates one discriminating example; otherwise, it makes two examples by strengthening. Indeed, the second call to `Strengthener_Mutation_Operator` generates the following discriminating example by strengthening the partial synonym ExpandRight to ExpandHeadOfRight over the relation waits.

holds			waits		
$State_0$	-	-	$State_0$	-	-
$State_1$	$Process_0$	$Mutex_0$	$State_1$	-	-
$State_2$	$Process_0$	$Mutex_0$	$State_2$	-	-
$State_2$	$Process_1$	$Mutex_1$	-	-	-
$State_3$	$Process_0$	$Mutex_0$	$State_3$	$Process_0$	$Mutex_1$
$State_3$	$Process_1$	$Mutex_1$	-	-	-
$State_4$	$Process_0$	$Mutex_0$	$State_4$	$Process_0$	$Mutex_1$
$State_4$	$Process_1$	$Mutex_1$	$State_4$	$Process_1$	$Mutex_0$

The user rejects the discriminating example since not only does a process have to grab a mutex in order, it also has to wait for a mutex of a higher order than its grabbed mutexes.

Depicted in Figure 6.6, she could repair the previous repair by constraining the tuples of waits similar to holds. Using heuristics, the bug could be revealed after reviewing 7 discriminating examples. Without using them, the number rises to 19 discriminating examples.

Figure 6.6 Repairing the underconstraint bug in the model from Figure 6.5

```
1 pred GrabbedInOrderFixed4 {
2   all p: Process |
3     all pre: State − so/last |
4       let post = so/next[pre] |
5       let had = p.(pre.holds), have = p.(post.(waits+holds)) |
6       let grabbed = have − had |
7         (some grabbed) ⇒ (no (grabbed & mo/prevs[had]))
8 }
```

Given the last repair, `DEBUG` finds that six mutations still survive. Once the user reviews them, the termination Condition-2 is satisfied. As the given model is in checking mode, the analyzer checks the property but finds no counter-example, which satisfies termination Condition-3. The final model is equivalent to the fixed model shipped with Alloy Analyzer 4.2 but is syntactically different from it.

6.3 Margaux's Heuristics are Better than Random

The case studies have shown how Margaux can generate discriminating examples in order to assist the user in exploring whether the written model properly expresses her intent. In the studies, some discriminating examples informed the user of bugs where the expressed model deviated from the intended meaning. In the following sections we analyze the effectiveness of Margaux's heuristics and compare it with Alloy Analyzer and Aluminum, two tools for analyzing Alloy models, in terms of generating discriminating examples.

In this section, we assess the effectiveness of Margaux's heuristics. We want to know whether the heuristics reduce the number of discriminating examples that the user must inspect before she discovers the bug. We also want to know the cost of heuristics to reveal the bugs as well as the additional cost of the simulacrum inferences that the heuristics need.

RQ 6.1. *How many examples should the user inspect before finding an an insightful discriminating example with* Margaux*?*

RQ 6.2. *How well do heuristics find discriminating examples?*

RQ 6.3. *What is the dominant component of the cost for producing discriminating examples by* Margaux*?*

To answer these questions, we have replaced the user with an oracle and run Margaux in two configurations to produce discriminating examples. In the first configuration,

Table 6.1 Effect of using heuristics in MARGAUX

Model	Bug	# Examples to reveal bugs		Find the Bug$_{(ms)}$ *with simulacrum inference*		Find the bug$_{(ms)}$ *without simulacrum inference*	
		Heuristic	Arbitrary	Heuristic	Arbitrary	Heuristic	Arbitrary
Dijkstra	Overconstraint	1	3	741,362	166,749	4,385	9,878
DijkstraFixed1	Underconstraint	3	16	753,236	424,441	13,906	97,570
DijkstraFixed2	Underconstraint	2	4	773,989	415,350	12,235	116,155
DijkstraFixed3	Underconstraint	7	19	898,324	472,490	115,425	93,942
DijkstraFixed	Termination	6	21	930,242	997,392	122,203	172,956
List	Overconstraint	2	3	92,381	65,385	2,636	9,355
ListFixed1	Underconstraint	2	3	103,872	74,248	12,261	16,380
ListFixed2	Underconstraint	2	4	133,090	97,519	13,768	22,213
ListFixed	Termination	1	1	157,849	111,499	11,697	21,630

MARGAUX employs heuristics to decide its procedures' non-deterministic choices; yet, it decides randomly in the second configuration. As we have already known the fixes for each bug, we have exploited them to build oracles to emulate the user responses. This way, we replace the user interactions for confirming or rejecting discriminating examples. We have evaluated experiments, with and without heuristics, using an Intel i7-2600K CPU at 3.40GHz with 16GB memory. All experiments are done with MiniSat.

Answering RQ 6.1, MARGAUX asked *13* discriminating examples, and *four* of them exposed overconstraint and underconstraint bugs, as depicted in Table 6.1. If MARGAUX does not incorporate heuristics, but instead arbitrary, then the number of reviewed discriminating examples rises to *63*. On average, after reviewing *two* discriminating examples, an exposing discriminating example is generated. MARGAUX without the heuristics needs to produce *2.5* times on average more examples until the bug is revealed.

Regarding RQ 6.2, if MARGAUX uses heuristics for finding fewer discriminating examples, it takes *13.66* minutes on average to find the bug. Without heuristics, the time decreases to *8.25* minutes on average. The gap between these two times is taken by the heuristics for analyzing the model so that MARGAUX produces fewer discriminating examples. MARGAUX with the heuristics takes *1.8* times on average longer than without heuristics to reveal bugs.

To clarify the time for inferring simulacra for heuristics, *i.e.*, RQ 6.3, we have separated the time of simulacrum inference from the time that MARGAUX spends on another tasks such as mutations and near-border example generation. The last two columns in Table 6.1 show the time for finding bugs with and without using heuristics. On average, simulacrum inference takes *93%* of analyzing time once heuristics are used and *81%* if MARGAUX uses no heuristics. From another perspective, if the time for the simulacrum inference, for both

mutation and some heuristics, is excluded, MARGAUX without heuristics performs *55%* slower.

6.4 Debugging the Dining Philosophers with Other Tools

In this section, we describe how other tools could assist a user to debug the issues explained in section 6.2. Alloy Analyzer [43] and Aluminum [78] are two major tools for simulating Alloy models at the time that we have written this dissertation. Neither of these tools explicitly support the techniques similar to MARGAUX, such as non-example generation, localization, and fix suggestion.

Similar to the technique explained in section 4.5, we negate the model and run it with Alloy Analyzer or Aluminum to get non-examples. Alloy Analyzer provides all non-symmetrical examples for a model and all non-symmetrical non-examples for the negation of the model. Aluminum provides scenario-exploration for a given model. The first scenario to explore a model is its minimal example. To explore different scenarios, Aluminum provides the augmentation operator. The operator adds a minimum number of tuples to produce a new scenario from an existing one.

Through this study, we want to assess how quick other tools can produce insightful discriminating examples. Due to many possibilities of examples and non-examples, we have based the study on the models and discriminating examples described in the previous section. That is, our studied models are the initial Dining Philosophers model and three following fixes. For each model, we attempt to produce the same example or non-example that MARGAUX produces. All in all we want to know that:

RQ 6.4. *How many examples should the user inspect before she can assume that she has found an insightful discriminating example with Alloy Analyzer?*

RQ 6.5. *How many examples should the user inspect before she can assume that she has found an insightful discriminating example with Aluminum?*

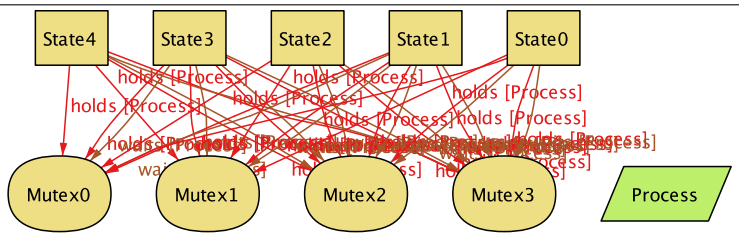
MARGAUX *VS.* Alloy Analyzer

Alloy Analyzer produces arbitrary sequences of examples. There might be possible that Alloy Analyzer produces a discriminating example in its first try similar to what MARGAUX produces. To assess how quick Alloy Analyzer produces an insightful discriminating example, we have reviewed the first 25 examples that Alloy Analyzer produces for each

model. By using Alloy Analyzer in this comparison, we want to find how likely one finds an insightful discriminating example from arbitrary sequence of examples.

For the model in Figure 6.2, Alloy Analyzer returns five examples. By running the negation of the model, Alloy Analyzer returns arbitrary non-examples, such as the one in Figure 6.7. Having 40 tuples, such a non-example looks to be complex to be an insightful discriminating examples. Within the next 25 non-examples, we could not find any instance similar to the one that MARGAUX generates to reveal the bug.

Figure 6.7 An arbitrary non-example that Alloy Analyzer produces for the model in Figure 6.2



By running the negation of the model, Alloy Analyzer produces arbitrary size non-examples.

To assess the tool with respect to a different bug, we consider the model in section 6.2, which is a fix for the previous bug. Using Alloy Analyzer, we could not find any example within the first 25 examples that reveals the bug.

For the next two models, depicted in Figures 6.4 and 6.5, Alloy Analyzer does not produce any interesting discriminating examples in its *25* tries.

Answering RQ 6.4, Alloy Analyzer produces arbitrary examples, as well as non-examples, that do not lead to any bug reveal in the discussed cases. Even arbitrary exploration of the first 25 (non-)examples shows that such (non-)examples are too complex to be insightful.

MARGAUX *VS.* Aluminum

Aluminum provides scenario-exploration by producing a minimum example of a given model and yielding the augmentation operator. By using Aluminum, we want to see whether absolute minimal (non-)examples are discriminating example. In the other words, we want to know whether absolute minimal (non-)examples can be useful to reveal the bugs. If a minimal example does not reveal bugs, we want to know how much a scenario

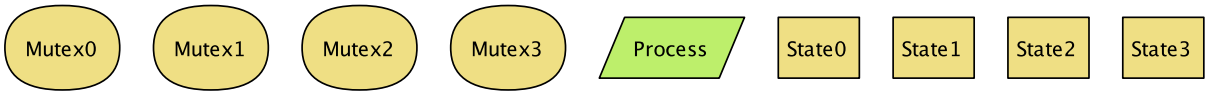
exploration should be proceeded to find the bugs. As the generated discriminating examples by MARGAUX are shown to be insightful in finding bugs, we direct the scenario exploration to find similar examples. By also counting the number of different possible augmentations at each step, we will have an understanding of the user's efforts in directing the tool.

For the model in Figure 6.2 Aluminum also finds the same examples as Alloy Analyzer has found. The first non-example returned by Aluminum has tuples for Mutex and State (Figure 6.8). The tool also provides 41 choices to augment the minimal non-example. Some augmentations lead to unintended non-examples. For instance, the result of augmenting the minimal example with an atom of Process and a tuple like ⟨State0,Process,Mutex⟩ is unintended. A few number of augmentations lead to non-examples similar to the one produced by MARGAUX.

Figure 6.8 A minimum example that Aluminum produces for the negation of the model in Figure 6.2

Aluminum initially generates an unintended minimal non-example and provides 41 choices for augmenting it; however, only very few of these augmentation choices lead to the bug reveal.

Figure 6.9 A minimal example that Aluminum produces for the model in Figure 6.3



For the model in Figure 6.3, Aluminum produces a minimal example with no valuation of holds and waits (Figure 6.9). The minimal example is not insightful in terms of revealing the bug. Aluminum suggests *57* different sets of tuples to augment the minimal example. After five augmentations, the user can see the same example as MARGAUX produces for this model. The next augmentations have to be selected among *40*, *39*, *22*, and *19* augmentations choices. This means that the user has to find the proper example from roughly 37 million ($\simeq 57 \times 40 \times 39 \times 22 \times 19$) possible examples.

For the models in Figures 6.4 and 6.5, Aluminum starts from a minimum example similar to Figure 6.9. To produce insightful discriminating examples similar to MARGAUX,

the user has to apply the augmentation operator three and six times, respectively. The size of the search space to find the same discriminating example for the former model is $57 \times 40 \times 39$ with respect to the number of augmentations at each step. The insightful discriminating example for the latter model is in the search space of size $57 \times 40 \times 39 \times 33 \times 29 \times 8$.

Regarding RQ 6.5, the minimal (non-)examples by Aluminum are too small to be insightful. These minimum examples are not useful to reveal the bugs in all cases. Our experiments, explained in section 4.5 , also showed that the same situation for non-examples. Using the augmentation operator, we have produced (non-)examples similar to the ones by MARGAUX. However, such (non-)examples have to be found within roughly *180* million possible (non-)examples on average. There are also *53* exploration choices on average to augment the minimal example. At each scenario-exploration step, there are roughly *37* choices for augmenting the current (non-)example.

All in all, MARGAUX can produce some insightful discriminating examples after less than three tries on average. Moreover, users are not involved in selecting from many choices to explore the model. Even if other insightful examples might exist within shorter sequences of augmentations, a manual exhaustive exploration to find them is almost impossible. We have demonstrated that MARGAUX helps in this important historical case study where the existing tools do not, despite the fact that the tool might be tuned for the case-study to find the discriminating examples in a fewer interactions with users.

6.5 Summary

In this chapter, we have presented two models that we have studied to evaluate MARGAUX, introduced in Chapter 5. Using Singly Linked List and Dijkstra's Dining Philosophers models, we have explained how MARGAUX infers constituent constraints of a model's simulacra. The inferences are based on the patterns, implication lattice, and consistency graph. With respect to the simulacrum, we have shown how MARGAUX mutates the model and produces various discriminating examples.

With two studied cases, we have demonstrated the feasibility of MARGAUX for the problems that users might face in debugging Alloy models. We have also shown how using heuristics directs the debugger to produce particular discriminating examples that detect a model's divergence from its actual form. Generating fewer discriminating examples needs more analysis, which certainly takes more time and resources. However, spending machine time and resources to detect subtle bugs in models usually saves the time and resources that would otherwise be needed later to deal with such bugs and their effects.

Doing these two case studies has helped us to develop our simulacrum inference framework and formulate debugging patterns for ternary relations. They have also guided us to take the next steps in improving the performance of the simulacrum inference stages. As the cases has shown, simulacrum inference is the dominant effort; therefore, distributing MARGAUX over a network of processing machines would be useful directions for future implementations.

We have also demonstrated the effectiveness of MARGAUX in producing insightful discriminating examples in comparison with existing state-of-the-art tools for analyzing Alloy models. To do so, we have studied how Aluminum and Alloy Analyzer could produce discriminating examples that reveal bugs in the Dijkstra's Dining Philosophers Alloy model. In this case study, MARGAUX can assist users in finding the bugs by producing three (non-)examples on average. Alloy Analyzer does not produce such (non-)examples within the first 25 (non-)example that it generated. In the scenario exploration by Aluminum, absolute minimal examples were not useful to reveal bugs. For augmenting such (non-)examples, users face 53 choices for augmenting the minimal (non-)example on average.

Chapter 7

Conclusion

In this dissertation, we have discussed different bugs that a relational first-order model might have: *partial overconstraint*, *total overconstraint*, and *underconstraint* bugs. We have proposed a system to generate *discriminating examples*, which helps engineers identify and understand these bugs in their logical specifications. We have demonstrated that this system is both *feasible* [§2, §3, §4, §5, §6] and *useful* [§6]. In particular, we have demonstrated that it is more useful than the current state of the art on the Dijkstra's dining philosophers problem.

Our position on the utility of discriminating examples in debugging relational first-order models is based on three well-established premises: concrete examples are more understandable than abstract models [4, 42, 64]; near-border examples are practical for learning [34, 110]; and debugging is a hypothesis-driven activity [57, 117]. The premises are not radical and are well established in other applications. For instance, the hypotheses-driven debugging approaches are well established in imperative debugging [54]. In Artificial Intelligence, near-border examples are used for learning corner-cases [52, 83].

In our proposed system, the debugger constructs a discrimination formula to encode a hypothesis about a potential bug. It then generates a discriminating example for the user to judge. By either accepting or rejecting the discriminating example, the user affirms or refutes the hypothesis. Our tools formulate hypotheses of two general kinds: that bugs happen near borders [§4]; and that the user might have expressed a logical property that is either slightly stronger or slightly weaker than what they intended [§5]. In support of the latter, we have developed a library of debug patterns for the debugger to explore, and heuristics to guide its search therein.

We have developed prototype extensions of Alloy Analyzer to assess the feasibility and

utility of discriminating examples in debugging relational first-order models. We have developed BORDEAUX [§4] for finding near-border examples. Using the extension, the user can review non-examples that can become examples with minimum changes. Another extension, MARGAUX [§5], finds the simulacra of a given model and its constituent constraints, mutates them, and produces discriminating examples and non-examples to affirm or refute the user's intention.

To realize the ideas behind BORDEAUX and MARGAUX, we have developed other extensions of Alloy Analyzer. LEVURE [§2] equips Alloy Analyzer to explicitly access the performance benefits of Kodkod's partial instance features. Using partial instance blocks, one can efficiently express examples and test different aspects of the model. With LEVURE, we have also developed BENTONITE [§3], to solve optimization queries searching for near-border examples. As measured with CLOC [21], we have developed 2,615, 3,114, 5,497, and 25,580 lines of Java code to implement LEVURE, BENTONITE, BORDEAUX, and MARGAUX, respectively.

To demonstrate how using discriminating examples can assist in understanding and localizing bugs, we have done two case-studies. One of these case-studies [68] is debugging subtle bugs in a model of Dijkstra's dining philosophers [43, 105]. Within these case-studies, we have illustrated how MARGAUX reasons to produce discriminating examples in order to reveal overconstraint and underconstraint bugs. In the case-studies, we have demonstrated the utility of discriminating examples in identifying and understanding different types of bugs and the feasibility of MARGAUX in generating insightful discriminating examples.

Since being published, both the LEVURE [66] and BENTONITE [67] tools have attracted attention and been used for solving problems with various applications. LEVURE stimulates researchers to use partial instances for test-case generation [49] and multi-objective optimization [81]. Analyzing model-checking properties in [24] is another application of BENTONITE. A year after we developed BENTONITE in 2014, Milicevic et al. [65] proposed a different approach for solving a similar type of problems. As we have shown in this dissertation, their approach works more efficiently for our application, so that we have based our following prototypes on the newer implementation. However, BENTONITE works better for other applications, *e.g.* [24].

In conclusion, discriminating examples are both feasible and useful for first-order relational logic languages: they can be computed by machines and used by people, and they advance the state of the art in debugging logical models.

7.1 Future work

The major techniques described in this dissertation offer multiple directions for the future work:

Reduced computation time by leveraging previous results. We have shown the potentials in enhancing example finders to create meaningful variants of the user's model, and to explain those variants to the engineer semantically, syntactically, and by example [§5]. The proposed debugger can process a broader range of models by having richer libraries of debug patterns. The debug patterns can reflect the experts' experiences in facing problems or encode frequent cases in different models. Also the debugger can analyze patterns over combinations of relations and take into account combinatorial cases of patterns. Localizing underconstraint bugs and suggesting fixes for such bugs are another avenue to extend the debugger. As the search algorithm relies on heuristics to direct its decisions, providing knowledge from previous interactions with the user is a way to bring forward bug detections.

Syntactical Mutation Operators. MARGAUX currently supports mutation operators based on debug patterns. In another direction to extend, MARGAUX could support syntactical mutation operators. Some candidates for syntactical mutation operators are:

Type	Change	
	From/To	To/From
Quantifier	**some**	**one/lone/all**
	all	**one/lone/some**
	one	**lone**
Local/Global	**pred** p[a:S]{p'[a]}	**pred** p[a:S]{**all** a':S | a'**in** a **implies** p'[a']}
Arithmetic	int + int	plus[int,int]
	int **-** int	minus[int,int]
	sum[A]	**sum** a:A' | a.r
Ordering	next[]	nexts[]
	prev[]	prevs[]

Due to the expressiveness of Alloy, many libraries of debug patterns are required to support debugging for different applications and forms of models. Applying the concepts of discriminating examples and library of debug patterns for a domain specific language (based on Alloy) with less complexity would be more practical.

Providing Discriminating Examples for Domain Experts. In addition to Alloy experts, MARGAUX might assist domain experts to understand and validate formal models utilized in various software activities, such as requirements analysis. Despite their specialized business knowledge, a lack of modeling skill often prevents them from understanding and working with formal models [4]. MARGAUX might assist domain experts in benefiting from formal models by generating focused discriminating examples. A case for studying this idea is an extension of the model explained in Section 4.1. The model is about using BORDEAUX to produce non-examples for an academic graduation plan. An extension of the model can be used in a system to assist academic program advisors for checking whether a student's completed courses comply with the program rules.

In this case, declarative models are appropriate for encoding graduation rules and revealing inconsistencies between them. From our conversation with academic advisors, as domain experts, there are valuable benefits in such a system although working and understanding logical models is not straightforward to them. However, they can understand examples for the model and validate whether they desire these examples. Using MARGAUX, or its extension, academic advisors would review discriminating examples that might help them to focus on particular aspects of the model and find potential partial overconstraint and underconstraint issues.

Possible Usability Testing. To test the usability of MARGAUX, we can perform *exploratory* and *evaluative* studies [5]. By an exploratory, or formative, study, we can find information to recognize and describe the problem from users' perspective. Exploratory studies are often performed by interviews and contextual inquiries. Evaluative studies examine the effectiveness of the idea and its realization with respect to usability objectives. They also allow generalizing the results in a broader domain. To evaluate particular use cases of a subject, the study has to be done in a controlled environment. By examining the usability in two controlled environments, the study generates information for comparing effects of a subject.

A *controlled user study* for evaluating the utility of discriminating examples in debugging relational first-order models can direct us to know how well typical users understand the cause of an issue with and without MARGAUX. Since the premises of our fundamental ideas have already been evaluated in similar applications, it would not be surprising to expand their consequences into our context. On the other hand, from an *uncontrolled evaluative user study*, we will learn how users debug their models in different applications. By analyzing the user interactions in two different releases, we can also identify other hypotheses that users might find interesting in practice.

From *exploratory user studies*, we want to know the other types of examples that the users find insightful. We also want to explore the usefulness of MARGAUX's heuristics in ranking different discriminating examples. Similar to the user study by Ko and Myers [53], we also want to know what other hypotheses Alloy users might find helpful for debugging Alloy models. As we know well from carrying out a previous *controlled evaluative study* in another application of Alloy models [88], finding a group of Alloy users is the most challenging part of similar empirical studies. An account of an unsuccessful exploratory user study before developing Aluminum also confirms this challenge [75].

As users with different levels of knowledge in Alloy have different hypotheses for debugging a model or any similar artifact, such as academic graduation plan, distinct exploratory and evaluative user studies should be designed. By performing different exploratory user studies with domain experts having less knowledge in logical modeling, we can learn the type of debugging hypotheses that they might ask while reviewing a relevant logic-based artifact. This study also would help us to adjust library of debug patterns with respect to the particular problem domain. For assessing the usefulness of the debugging technique in assisting domain experts, a controlled evaluative user study needs two groups of users with fairly equal knowledge in domain: study group + control group.

Near-border examples Our prototype demonstrates the computational feasibility of finding relative minimum distances between examples and non-examples [§4]. The implementation is based on Alloy*, as we have found it an effective way to realize the prototype with respect to time and resources. Using another solver for improving the performance is one way to extend the prototype. As the experiments have shown, translating step by Kodkod is a bottleneck for some studied cases. In addition to using the mentioned parallelization techniques, recycling the already translated parts of the model is one way to increase the performance. As we have defined, the relative minimal distance is based on the number tuples between two instances. Another direction of research is to assess another interpretation of the concept of distance.

Staged solving technique As we needed to run queries in the form of $\exists\forall$, we have extended Alloy Analyzer to automatically perform additional scope computations. We have used a stage evaluation technique to decrease the size of universe of discourse for running such queries [§3]. Although we have shown a posterior tool can perform more efficient in our applications, the experiments show values of our prototype in checking the queries. A hybrid technique from our staging approach and counterexample guided inductive synthesis technique is another future work. As our preliminary study showed, dropping non-isomorphic can lead to unsound instances satisfying $\exists\forall$ queries. Since the

number of non-isomorphic objects can, in some cases, be exponentially smaller than the number of objects [102], there will be a dramatic performance improvement for some class of $\exists\forall$ queries to exclude isomorphic objects [62].

References

[1] P. Abad, N. Aguirre, V. Bengolea, D. Ciolek, M. F. Frias, J. P. Galeotti, T. Maibaum, M. Moscato, N. Rosner, and I. Vissani. Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. In *ICST '13*, 2013.

[2] C. Alexander. *A pattern language: towns, buildings, construction.* Oxford University Press, 1977.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, Jan. 2004.

[4] K. Bak, D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wasowski, and D. Rayside. Example-driven modeling: model = abstractions + examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1273–1276. IEEE Press, 2013.

[5] C. M. Barnum. *Usability testing essentials: ready, set... test!* Elsevier, 2010.

[6] E. Batot. Generating examples for knowledge abstraction in MDE: a multi-objective framework. In M. Balaban and M. Gogolla, editors, *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015, Ottawa, Canada, September 29, 2015.*, volume 1503 of *CEUR Workshop Proceedings*, pages 1–6. CEUR-WS.org, 2015.

[7] K. Beck. *Test-Driven Development.* Addison-Wesley, 2003.

[8] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining Counterexamples Using Causality. In *Proc.21st CAV*, volume 5643 of *LNCS*, pages 94–108. Springer-Verlag, July 2009.

[9] D. Bianculli, C. Ghezzi, C. Pautasso, and P. Senti. Specification patterns from research to industry: a case study in service-based applications. In *Proceedings of the 34th International Conference on Software Engineering*, pages 968–976. IEEE Press, 2012.

[10] A. Biere. *Handbook of Satisfiability*, volume 185. Ios PressInc, 2009.

[11] T. Budd, R. DeMillo, R. Lipton, and F. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *7th POPL*, Las Vegas, NV, Jan. 1980.

[12] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.

[13] Y. Cai and K. Sullivan. Modularity analysis of logical design models. In S. Easterbrook and S. Uchitel, editors, *Proc.21st ASE*, Tokyo, Japan, Sept. 2006.

[14] Y. Cai, S. Huynh, and T. Xie. A framework and tool supports for testing modularity of software design. In A. Egyed and B. Fischer, editors, *Proc.22nd ASE*, pages 441–444, Atlanta, GA, Nov. 2007.

[15] B. Cha, K. Iwama, Y. Kambayashi, and S. Miyazaki. Local search algorithms for partial maxsat. *AAAI/IAAI*, 263268, 1997.

[16] W. Chan. Temporal-logic queries. In *International Conference on Computer Aided Verification*, pages 450–463. Springer, 2000.

[17] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. In *Proc.8th FASE*, pages 220–236. Springer-Verlag, Apr. 2005.

[18] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[19] A. Cunha. Bounded model checking of temporal formulas with Alloy. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 303–308. Springer, 2014.

[20] A. Cunha, N. Macedo, and T. Guimaraes. Target oriented relational model finding. In *Fundamental Approaches to Software Engineering*, pages 17–31. Springer, 2014.

[21] A. Danial. Count lines of code, 2006. URL `https://github.com/AlDanial/cloc`.

[22] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[23] C. Desrosiers, P. Galinier, A. Hertz, and S. Paroz. Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *Journal of Combinatorial Optimization*, 18:124–150, 2009.

[24] D. Dietrich, P. Shaker, J. Atlee, D. Rayside, and J. Gorzny. Feature Interaction Analysis of the Feature-Oriented Requirements-Modelling Language Using Alloy. In *MoDeVVa Workshop at MODELS Conference*, 2012.

[25] D. Dietrich, P. Shaker, J. Gorzny, J. Atlee, and D. Rayside. Translating the Feature-Oriented Requirements Modelling Language to Alloy. Technical Report CS-2012-12, University of Waterloo, David R. Cheriton School of Computer Science, 2012.

[26] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, may 1999. ACM.

[27] J. Edwards, D. Jackson, and E. Torlak. A type system for object models. In R. N. Taylor and M. B. Dwyer, editors, *Proc.12th FSE*, Newport Beach, CA, USA, Nov. 2004.

[28] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[29] A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *Artif. Intell. Eng. Des. Anal. Manuf.*, 26(1):53–62, Feb. 2012.

[30] B. Fraikin, M. Frappier, and R. St-Denis. Modeling the supervisory control theory with Alloy. In J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *LNCS*, pages 94–107. Springer, Springer-Verlag, June 2012.

[31] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Analysis of Invariants for Efficient Bounded Verification. In P. Tonella and A. Orso, editors, *Proc.19th ISSTA*, pages 25–36. ACM, 2010.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[33] S. Ganov, S. Khurshid, and D. E. Perry. Annotations for Alloy: Automated Incremental Analysis Using Domain Specific Solvers. In T. Aoki and K. Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *LNCS*, pages 414–429. Springer, 2012.

[34] M. L. Gick and K. Paterson. Do contrasting examples facilitate schema acquisition and analogical transfer? *Canadian Journal of Psychology/Revue canadienne de psychologie*, 46(4):539, 1992.

[35] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, 2006.

[36] V. Gruhn and R. Laue. Specification patterns for time-related properties. In *Temporal Representation and Reasoning, International Syposium on*, pages 189–191. IEEE Computer Society, 2005.

[37] A. Gurfinkel, B. Devereux, and M. Chechik. Model exploration with temporal logic query checking. *ACM SIGSOFT Software Engineering Notes*, 27(6):139–148, 2002.

[38] S. Halle, R. Villemaire, and O. Cherkaoui. Specifying and validating data-aware temporal web service properties. *TSE*, 35(5):669–683, 2009.

[39] F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting mucs from constraint networks. In *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy*, pages 113–117, 2006.

[40] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment*, 1(1):736–747, 2008.

[41] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, pages 249–256, 1997.

[42] D. Jackson. *Software Abstractions: Logic, Language, and Anlysis*. MIT Press, 2011.

[43] D. Jackson. Alloy, 2012. URL `http://alloy.mit.edu/alloy/`.

[44] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21–22, Apr. 1996.

[45] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[46] U. Junker. Quickxplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th national conference on Artifical intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004.

[47] S. Kallel, A. Charfi, T. Dinkelaker, M. Mezini, and M. Jmaiel. Specifying and monitoring temporal properties in web services compositions. In *Web Services, 2009. ECOWS'09. Seventh IEEE European Conference on*, pages 148–157. IEEE, 2009.

[48] S. Katz, O. Grumberg, and D. Geist. 'Have I written enough Properties?' — A Method of Comparison between Specification and Implementation. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 280–297. Springer-Verlag, 1999.

[49] H. Keramati and S.-H. Mirian-Hosseinabadi. Generating semantically valid test inputs using constrained input grammars. *Information and Software Technology*, 57: 204–216, 2015.

[50] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.

[51] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A Case for Efficient Solution Enumeration. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 272–286. Springer, 2004.

[52] K. Kira and L. A. Rendell. A practical approach to feature selection. In *Proceedings of the ninth international workshop on Machine learning*, pages 249–256, 1992.

[53] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1):41–84, 2005.

[54] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Computing Surveys*, 43(3):21:1–21:44, Apr. 2011.

[55] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *International Journal on Software Tools for Technology Transfer*, pages 1–21, 2011.

[56] S. Konrad and B. H. Cheng. Real-time specification patterns. In W. Griswold and B. Nuseibeh, editors, *Proc.27th ICSE*, pages 372–381, NYC, 2005. ACM. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062604.

[57] J. F. Krems. Expert strategies in debugging: experimental results and a computational model. In *Cognition and computer programming*, pages 241–254. Ablex Publishing Corp., 1994.

[58] L. Lamport. The TLA Home Page, 2010. URL `http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html`.

[59] S. Liu, X. Wang, and W. Miao. Supporting requirements analysis using pattern-based formal specification construction. In *Formal Methods and Software Engineering*, pages 100–115. Springer, 2015.

[60] N. Macedo and A. Cunha. Implementing QVT-R bidirectional model transformations using Alloy. In *International Conference on Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.

[61] N. Macedo and A. Cunha. Least-change bidirectional model transformation with qvt-r and atl. *Software & Systems Modeling*, pages 1–28, 2014.

[62] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proc.16th ASE*, pages 22–31, San Diego, CA, Nov. 2001.

[63] J. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications. In *Multiple-Valued Logic (ISMVL), 2010 40th IEEE International Symposium on*, pages 9 –14, may 2010.

[64] L. Mendel. Modeling by example. Master's thesis, Massachusetts Institute of Technology, sep 2007.

[65] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A General-purpose Higher-order Relational Constraint Solver. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 609–619. IEEE Press, 2015.

[66] V. Montaghami and D. Rayside. Extending Alloy with Partial Instances. In *Proceedings of the Third international conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 122–135. Springer-Verlag, 2012.

[67] V. Montaghami and D. 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.

[68] V. Montaghami and D. Rayside. Pattern-based debugging of declarative models. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 322–327. IEEE, 2015.

[69] V. Montaghami, O. Odunayo, B. Guntoori, and D. Rayside. Bordeaux prototype. `https://github.com/drayside/bordeaux`, 2016.

[70] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 376–390. Springer, 2006.

[71] G. Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[72] J. P. Near. An Imperative Extension to Alloy and a Compiler for its Execution. Master's thesis, Massachusetts Institute of Technology, 2010.

[73] J. P. Near. From relational specifications to logic programs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[74] J. P. Near and D. Jackson. An imperative extension to Alloy. In *International Conference on Abstract State Machines, Alloy, B and Z*, pages 118–131. Springer, 2010.

[75] T. Nelson. *First-order Models For Configuration Analysis*. PhD thesis, Brown University, 2013.

[76] T. Nelson, C. Barratt, D. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration*, pages 1–8. USENIX Association, 2010.

[77] T. Nelson, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Toward a more complete Alloy. In *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pages 136–149. Springer, 2012.

[78] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: principled scenario exploration through minimality. In B. Cheng and K. Pohl, editors, *Proc.35th ICSE*, pages 232–241, San Francisco, CA, 2013.

[79] C. Nentwich, W. Emmerich, and A. Finkelsteiin. Consistency management with repair actions. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 455–464. IEEE Computer Society, 2003.

[80] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58 (4):66–73, Mar. 2015.

[81] R. Olaechea, S. Stewart, K. Czarnecki, and D. Rayside. Modelling and multi-objective optimization of quality attributes in variability-rich software. In *NFPinDSML Workshop at MODELS Conference*, NFPinDSML '12, pages 2:1–2:6. ACM, 2012.

[82] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209. ACM, 2011.

[83] L. Popelínsky. Efficient relational learning from sparse data. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '02, pages 11–20, London, UK, UK, 2002. Springer-Verlag.

[84] F. Raimondi, J. Skene, and W. Emmerich. Efficient online monitoring of web-service slas. In M. J. Harrold and G. Murphy, editors, *Proc.16th FSE*, pages 170–180, Atlanta, GA, Nov. 2008.

[85] D. Rayside and H.-C. Estler*. A spreadsheet-like user interface for combinatorial multi-objective optimization. In P. Martin and A. W. Kark, editors, *Proc.CASCON*, Toronto, Nov. 2009.

[86] D. Rayside, F. Chang*, G. Dennis*, R. Seater*, and D. Jackson. Automatic visualization of relational logic models. In *First Workshop on the Layout of (Software) Engineering Diagrams (LED'07)*, Sept. 2007.

[87] D. Rayside, H.-C. Estler*, and D. Jackson. A Guided Improvement Algorithm for Exact, General Purpose, Many-Objective Combinatorial Optimization. Technical Report MIT-CSAIL-TR-2009-033, MIT CSAIL, 2009. URL `http://hdl.handle.net/1721.1/46322`.

[88] D. Rayside, V. Montaghami*, F. Leung*, A. Yuen*, K. Xu*, and D. Jackson. Synthesizing iterators from abstraction functions. In W. Binder and K. Ostermann, editors, *Proc.11th GPCE*, Dresden, Germany, Sept. 2012.

[89] A. Reder and A. Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 220–229. ACM, 2012.

[90] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1): 57–95, Apr. 1987.

[91] N. Rosner, J. P. Galeotti, C. G. L. Pombo, and M. F. Frias. ParAlloy: towards a framework for efficient parallel analysis of alloy models. In *International Conference on Abstract State Machines, Alloy, B and Z*, pages 396–397. Springer, 2010.

[92] N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias. Ranger: Parallel analysis of alloy models by range partitioning. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 147–157. IEEE, 2013.

[93] S. Saghafi and D. J. Dougherty. Razor: provenance and exploration in model-finding. In *4th Workshop on Practical Aspects of Automated Reasoning (PAAR)*. Citeseer, 2014.

[94] R. M. Seater. Core extraction and non-example generation: debugging and understanding logical models. Master's thesis, Massachusetts Institute of Technology, 2004.

[95] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983. ISBN 0262192187.

[96] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 94–105. IEEE, 2003.

[97] J. Silva. A comparative study of algorithmic debugging strategies. In *Logic-Based Program Synthesis and Transformation*, pages 143–159. Springer, 2007.

[98] A. Sullivan, R. N. Zaeem, S. Khurshid, and D. Marinov. Towards a test automation framework for alloy. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, pages 113–116. ACM, 2014.

[99] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proc.9th FSE*, pages 99–108, Vienna, Austria, Sept. 2001.

[100] S. Toda. On the computational power of PP and (+)P. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 514–519, Oct. 1989.

[101] E. Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, MIT, Cambridge, MA, USA, 2009. AAI0821754.

[102] E. Torlak and D. Jackson. Kodkod: A relational model finder. In O. Grumberg and M. Huth, editors, *Proc.13th TACAS*, volume 4424 of *LNCS*, pages 632–647, Braga, Portugal, Mar. 2007. Springer-Verlag.

[103] E. Torlak, F. S.-H. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *Proceedings of the 15th international symposium on Formal Methods*, FM '08, pages 326–341. Springer-Verlag, 2008.

[104] E. Torlak, M. Taghdiri, G. Dennis, and J. P. Near. Applications and extensions of Alloy: past, present and future. *Mathematical Structures in Computer Science*, 23 (04):915–933, 2013.

[105] M. Ulbrich, U. Geilmann, A. A. E. Ghazi, and M. Taghdiri. A Proof Assistant for Alloy Specifications. In C. Flanagan and B. König, editors, *Proc.18th TACAS*, volume 7214 of *LNCS*, pages 422–436. Springer-Verlag, 2012. doi: 10.1007/978-3-642-28756-5_29.

[106] A. Vakili and N. A. Day. Temporal logic model checking in alloy. In J. Derrick, J. A. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *LNCS*, pages 150–163. Springer, Springer-Verlag, June 2012.

[107] R. Van Der Straeten, J. P. Puissant, and T. Mens. Assessing the kodkod model finder for resolving model inconsistencies. In *Proceedings of the 7th European conference on Modelling foundations and applications*, ECMFA'11, pages 69–84. Springer-Verlag, 2011.

[108] H. Van Maaren and S. Wieringa. Finding guaranteed muses fast. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 291–304, 2008.

[109] I. Vessey. Expertise in debugging computer programs: an analysis of the content of verbal protocols. *IEEE Trans. Syst. Man Cybern.*, 16(5):621–637, Sept. 1986.

[110] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, third edition, 1992. pp. 150-356.

[111] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 58–68. IEEE Press, 2012.

[112] P. Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, Mar. 2012. ISSN 0146-4833. doi: 10.1145/2185376.2185383. URL `http://doi.acm.org/10.1145/2185376.2185383`.

[113] P. Zave. How to make chord correct (using a stable base). *CoRR*, abs/1502.06461, 2015. URL `http://arxiv.org/abs/1502.06461`.

[114] P. Zave. A practical comparison of alloy and spin. *Formal Aspects of Computing*, 27 (2):239–253, 2015.

[115] D. Zayan, M. Antkiewicz, and K. Czarnecki. Effects of using examples on structural model comprehension: a controlled experiment. In *Proceedings of the 36th International Conference on Software Engineering*, pages 955–966. ACM, 2014.

[116] A. Zeller. Isolating cause-effect chains from computer programs. In J. Daemen and V. Rijmen, editors, *Proc.10th FSE*, pages 1–10, Charleston, SC, Nov. 2002. ISBN 1-58113-514-9.

[117] A. Zeller. *Why programs fail: a guide to systematic debugging.* Morgan Kaufmann, 2009.

Appendices

Appendix A

List of Temporal Patterns' Structures for a Ternary Relation

The complete list of 160 temporal debug patterns' structure for a ternary relation [§5.1.5]. The pattern name is constructued from combination of parameters expressed in Figure 5.3. Debug patterns are design for MARGAUX to analyze a given model and generates discriminating examples [§5.1].

```
pred ExpanddTaillOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                         left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred ExpanddTaillOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| let c = middle.(left'.r)|
  let c' = middle.(left''.r)| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred ExpanddTaillOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                          left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred ExpanddTaillOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
    left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| let c = (left'.r).right|
  let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred ExpanddTaillOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,left_first: univ,
```

```
                                        left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c|(c in c') and
   (c'!inc) and (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}

pred ExpanddTaillOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred ExpanddTaillOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                      left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}

pred ExpanddTaillOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred ExpandTaillOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                        left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred ExpandTaillOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
  (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred ExpandTaillOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                         left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and
   (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred ExpandTaillOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                        left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and
   (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred ExpandTaillOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                     left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and
```

```
    (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred ExpandTaillOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and
    (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred ExpandTaillOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                     left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c|
    (c in c') and (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred ExpandTaillOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c|
    (c in c') and (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred MutateTaillOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| eq[#c' ,# c] and
    (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred MutateTaillOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                               left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| eq[#c' ,# c] and
    (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred MutateTaillOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                         left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c|
    eq[#c' ,# c] and (some delta implies(some c implies lt[max[delta,right_next],min[c,right_next],right_next]))
}
pred MutateTaillOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                         left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c|
    eq[#c' ,# c] and (some delta implies(some c implies lt[max[delta,middle_next],min[c,middle_next],middle_next]))
}
pred ContracttTaillOfRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                             left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}
```

```
pred ContracttTaillOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                           left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
   (c!inc') and (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}
pred ContracttTaillOfMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                          left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and (c!inc') and
   (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ContracttTaillOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                            left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and (c!inc') and
   (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ContracttTaillOfRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                      left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c−c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}
pred ContracttTaillOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                        left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c−c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}
pred ContracttTaillOfMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                     left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c−c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ContracttTaillOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c−c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ContractTaillOfRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}
pred ContractTaillOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
```

```
                                left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}

pred ContractTaillOfMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                         left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ContractTaillOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                           left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ContractTaillOfRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                     left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c−c'| (c' in c) and
    (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}
pred ContractTaillOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'|
   (c' in c) and (some delta implies(some c' implies lt[max[delta,right_next],min[c',right_next],right_next]))
}
pred ContractTaillOfMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                   left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c−c'| (c' in c) and
    (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ContractTaillOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c−c'| (c' in c) and
    (some delta implies(some c' implies lt[max[delta,middle_next],min[c',middle_next],middle_next]))
}
pred ExpanddTailOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                        left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred ExpanddTailOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
```

```
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}

pred ExpanddTailOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                         left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ExpanddTailOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                        left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ExpanddTailOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                     left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred ExpanddTailOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred ExpanddTailOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                     left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ExpanddTailOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ExpandTailOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred ExpandTailOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,left_first: univ,
                                      left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
```

```
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c|
   (c in c') and (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}


pred ExpandTailOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                        left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and
   (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ExpandTailOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and
   (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ExpandTailOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                    left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c'−c| (c in c') and
    (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred ExpandTailOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c'−c| (c in c') and
    (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred ExpandTailOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                    left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c'−c| (c in c') and
    (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ExpandTailOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c'−c| (c in c') and
    (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred MutateTailOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| eq[#c' ,# c] and
  (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred MutateTailOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                        left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
```

```
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| eq[#c' ,# c] and
   (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}


pred MutateTailOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                    left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c'−c| eq[#c' ,# c] and
    (some delta implies(some c implies lt[max[delta,right_next],max[c,right_next],right_next]))
}
pred MutateTailOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                    left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c'−c| eq[#c' ,# c] and
    (some delta implies(some c implies lt[max[delta,middle_next],max[c,middle_next],middle_next]))
}
pred ContracttTailOfRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and (c!inc') and
   (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}
pred ContracttTailOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                          left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and (c!inc') and
   (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}
pred ContracttTailOfMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and (c!inc') and
   (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ContracttTailOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                           left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and (c!inc') and
   (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ContracttTailOfRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c−c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}
pred ContracttTailOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
```

```
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c-c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}


pred ContracttTailOfMiddle_RightStatic_LastLeftEmpty[r: univ->univ->univ, left, middle, right: univ,
                    left_first: univ, left_next: univ->univ, middle_first: univ, middle_next: univ->univ]{
 no last[left,left_next].r
 all left': left-last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c-c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ContracttTailOfMiddle_RightStatic[r: univ->univ->univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ->univ, middle_first: univ, middle_next: univ->univ]{
 all left': left-last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c-c'| (c' in c) and (c!inc') and
    (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ContractTailOfRight_LastLeftEmpty[r: univ->univ->univ, left, middle, right: univ,
   left_first: univ, left_next: univ->univ, right_first: univ, right_next: univ->univ]{
 no last[left,left_next].r
 all left': left-last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c-c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}
pred ContractTailOfRight[r: univ->univ->univ, left, middle, right: univ, left_first: univ,
                         left_next: univ->univ, right_first: univ, right_next: univ->univ]{
 all left': left-last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c-c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}
pred ContractTailOfMiddle_LastLeftEmpty[r: univ->univ->univ, left, middle, right: univ,
  left_first: univ, left_next: univ->univ, middle_first: univ, middle_next: univ->univ]{
 no last[left,left_next].r
 all left': left-last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c-c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ContractTailOfMiddle[r: univ->univ->univ, left, middle, right: univ, left_first: univ,
                          left_next: univ->univ, middle_first: univ, middle_next: univ->univ]{
 all left': left-last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c-c'| (c' in c) and
   (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ContractTailOfRight_MiddleStatic_LastLeftEmpty[r: univ->univ->univ, left, middle, right: univ,
                    left_first: univ, left_next: univ->univ, right_first: univ, right_next: univ->univ]{
 no last[left,left_next].r
 all left': left-last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c-c'| (c' in c) and
    (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}
pred ContractTailOfRight_MiddleStatic[r: univ->univ->univ, left, middle, right: univ, left_first: univ,
                                      left_next: univ->univ, right_first: univ, right_next: univ->univ]{
```

```
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c−c'| (c' in c) and
    (some delta implies(some c' implies lt[max[delta,right_next],max[c',right_next],right_next]))
}

pred ContractTailOfMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c−c'| (c' in c) and
    (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ContractTailOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                      left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c−c'| (c' in c) and
    (some delta implies(some c' implies lt[max[delta,middle_next],max[c',middle_next],middle_next]))
}
pred ExpanddHeaddOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                         left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
}
pred ExpanddHeaddOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
   left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
}
pred ExpanddHeaddOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                          left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpanddHeaddOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
   left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpanddHeaddOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                      left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
}
pred ExpanddHeaddOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
```

```
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
}

pred ExpanddHeaddOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
 left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpanddHeaddOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|
   let delta = c'−c| (c in c') and (c'!inc) and
    (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpandHeaddOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                        left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
}
pred ExpandHeaddOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
}
pred ExpandHeaddOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                         left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and
   (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpandHeaddOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c')
   and (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpandHeaddOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
  left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
}
pred ExpandHeaddOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
```

```
598   let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c|
599     (c in c') and (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
600 }
601
602
603
604 pred ExpandHeaddOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
605  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
606  all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
607   let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c|
608     (c in c') and (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
609 }
610 pred ExpandHeaddOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
611                 left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
612  no left_first.r
613  all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
614   let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c|
615     (c in c') and (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
616 }
617 pred MutateHeaddOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
618                                 left_next: univ→univ, right_first: univ, right_next: univ→univ]{
619  all left': left−last[left,left_next]| let left'' = left'.left_next| let c = middle.(left'.r)|
620   let c' = middle.(left''.r)| let delta = c'−c| eq[#c' ,# c] and
621     (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
622 }
623 pred MutateHeaddOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
624                                 left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
625  all left': left−last[left,left_next]| let left'' = left'.left_next|
626   let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| eq[#c' ,# c] and
627     (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
628 }
629 pred MutateHeaddOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
630   left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
631  all left': left−last[left,left_next]| let left'' = left'.left_next|all middle' : middle|
632   let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c|
633     eq[#c' ,# c] and (some delta implies lt[max[c,right_next],min[delta,right_next],right_next])
634 }
635 pred MutateHeaddOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
636                                           left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
637  all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
638   let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c|
639     eq[#c' ,# c] and (some delta implies lt[max[c,middle_next],min[delta,middle_next],middle_next])
640 }
641 pred ContracttHeaddOfRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
642       left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
643  no last[left,left_next].r
644  all left': left−last[left,left_next]| let left'' = left'.left_next| let c = middle.(left'.r)|
645   let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and (c!inc') and
646     (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
647 }
648 pred ContracttHeaddOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
649                                   left_next: univ→univ, right_first: univ, right_next: univ→univ]{
650  all left': left−last[left,left_next]| let left'' = left'.left_next| let c = middle.(left'.r)|
651   let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and (c!inc') and
652     (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
653 }
654 pred ContracttHeaddOfMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
```

```
655     left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
656  no last[left,left_next].r
657  all left': left−last[left,left_next]| let left'' = left'.left_next| let c = (left'.r).right|
658   let c' = (left''.r).right| let delta = c−c'| (c' in c) and (c!inc') and
659    (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])
660 }
661 pred ContracttHeaddOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
662                              left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
663  all left': left−last[left,left_next]| let left'' = left'.left_next| let c = (left'.r).right|
664   let c' = (left''.r).right| let delta = c−c'| (c' in c) and (c!inc') and
665    (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])
666 }
667 pred ContracttHeaddOfRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle,right: univ,
668                     left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
669  no last[left,left_next].r
670  all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
671   let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c) and
672    (c!inc') and (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
673 }
674 pred ContracttHeaddOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
675     left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
676  all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
677   let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c) and
678    (c!inc') and (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
679 }
680 pred ContracttHeaddOfMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
681                     left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
682  no last[left,left_next].r
683  all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
684   let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and
685    (c!inc') and (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])
686 }
687 pred ContracttHeaddOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
688   left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
689  all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
690   let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and
691    (c!inc') and (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])
692 }
693 pred ContractHeaddOfRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
694     left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
695  no last[left,left_next].r
696  all left': left−last[left,left_next]| let left'' = left'.left_next| let c = middle.(left'.r)|
697   let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
698    (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
699 }
700 pred ContractHeaddOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
701                           left_next: univ→univ, right_first: univ, right_next: univ→univ]{
702  all left': left−last[left,left_next]| let left'' = left'.left_next|
703  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
704   (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
705 }
706 pred ContractHeaddOfMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
707    left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
708  no last[left,left_next].r
709  all left': left−last[left,left_next]| let left'' = left'.left_next| let c = (left'.r).right|
710   let c' = (left''.r).right| let delta = c−c'| (c' in c) and
711    (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])
```

```
}
pred ContractHeaddOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                           left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and
   (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])}
pred ContractHeaddOfRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c)
   and (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
}
pred ContractHeaddOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c) and
   (some delta implies lt[max[c',right_next],min[delta,right_next],right_next])
}
pred ContractHeaddOfMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and
   (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])
}
pred ContractHeaddOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and
   (some delta implies lt[max[c',middle_next],min[delta,middle_next],middle_next])
}
pred ExpanddHeadOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                        left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ExpanddHeadOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ExpanddHeadOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                         left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc) and
   (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpanddHeadOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
   left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and (c'!inc)
   and (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
```

```
}
pred ExpanddHeadOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                     left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])}
pred ExpanddHeadOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and (c'!inc)
   and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ExpanddHeadOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                     left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c| (c in c') and (c'!inc)
   and (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpanddHeadOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c| (c in c') and
   (c'!inc) and (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpandHeadOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                       left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c')
   and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ExpandHeadOfRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                      left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ExpandHeadOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                        left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and
   (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpandHeadOfMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| (c in c') and
   (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpandHeadOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                    left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
```

```
}
pred ExpandHeadOfRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| (c in c') and
   (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ExpandHeadOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                    left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c| (c in c') and
   (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ExpandHeadOfMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c| (c in c') and
   (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred MutateHeadOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                               left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c'−c| eq[#c' ,# c] and
   (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred MutateHeadOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                              left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c'−c| eq[#c' ,# c] and
   (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred MutateHeadOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                        left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c'−c| eq[#c' ,# c] and
   (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred MutateHeadOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                       left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c'−c| eq[#c' ,# c] and
   (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ContracttHeadOfRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                            left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
   (c!inc') and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ContracttHeadOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                                left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
   (c!inc') and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
```

```
}
pred ContracttHeadOfMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
    left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and
   (c!inc') and (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ContracttHeadOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                           left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and
   (c!inc') and (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ContracttHeadOfRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c) and
   (c!inc') and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ContracttHeadOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c) and
   (c!inc') and (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ContracttHeadOfMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                   left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and (c!inc')
   and (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ContracttHeadOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and (c!inc')
   and (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
}
pred ContractHeadOfRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
    left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
   (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ContractHeadOfRight[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
                         left_next: univ→univ, right_first: univ, right_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| let delta = c−c'| (c' in c) and
   (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
}
pred ContractHeadOfMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
```

```
940   let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and
941     (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
942 }
943 pred ContractHeadOfMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
944                                left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
945  all left': left−last[left,left_next]| let left'' = left'.left_next|
946   let c = (left'.r).right| let c' = (left''.r).right| let delta = c−c'| (c' in c) and
947     (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
948 }
949 pred ContractHeadOfRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
950                     left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
951  no last[left,left_next].r
952  all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
953   let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c) and
954     (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
955 }
956 pred ContractHeadOfRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
957     left_first: univ, left_next: univ→univ, right_first: univ, right_next: univ→univ]{
958  all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
959   let c = middle'.(left'.r)| let c' = middle'.(left''.r)| let delta = c−c'| (c' in c) and
960     (some delta implies lte[min[c,right_next],min[delta,right_next],right_next])
961 }
962 pred ContractHeadOfMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
963                 left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
964  no last[left,left_next].r
965  all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
966   let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and
967     (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
968 }
969 pred ContractHeadOfMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
970  left_first: univ, left_next: univ→univ, middle_first: univ, middle_next: univ→univ]{
971  all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
972   let c = (left'.r).right'| let c' = (left''.r).right'| let delta = c−c'| (c' in c) and
973     (some delta implies lte[min[c,middle_next],min[delta,middle_next],middle_next])
974 }
975 pred ExpanddRight[r: univ→univ→univ, left, middle, right: univ,
976                               left_first: univ, left_next: univ→univ]{
977  all left': left−last[left,left_next]| let left'' = left'.left_next|
978  let c = middle.(left'.r)| let c' = middle.(left''.r)| (c in c') and (c'!inc)
979 }
980 pred ExpanddRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
981                                          left_first: univ, left_next: univ→univ]{
982  no left_first.r
983  all left': left−last[left,left_next]| let left'' = left'.left_next|
984  let c = middle.(left'.r)| let c' = middle.(left''.r)| (c in c') and (c'!inc)
985 }
986 pred ExpanddMiddle[r: univ→univ→univ, left, middle, right: univ, left_first: univ,
987                                                    left_next: univ→univ]{
988  all left': left−last[left,left_next]| let left'' = left'.left_next|
989  let c = (left'.r).right| let c' = (left''.r).right| (c in c') and (c'!inc)
990 }
991 pred ExpanddMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
992                                           left_first: univ, left_next: univ→univ]{
993  no left_first.r
994  all left': left−last[left,left_next]| let left'' = left'.left_next|
995  let c = (left'.r).right| let c' = (left''.r).right| (c in c') and (c'!inc)
996 }
```

```
pred ExpanddRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
                               left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| (c in c') and (c'!inc)
}
pred ExpanddRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                              left_first: univ, left_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all middle' : middle|
  let c = middle'.(left'.r)| let c' = middle'.(left''.r)| (c in c') and (c'!inc)
}
pred ExpanddMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
                               left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| (c in c') and (c'!inc)
}
pred ExpanddMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                              left_first: univ, left_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next| all right' : right|
  let c = (left'.r).right'| let c' = (left''.r).right'| (c in c') and (c'!inc)
}
pred ExapndRight[r: univ→univ→univ, left, middle, right: univ,
                 left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  let c = middle.(left'.r)| let c' = middle.(left''.r)| (c in c')
}
pred ExapndRight_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                left_first: univ, left_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)| (c in c')
}
pred ExapndMiddle[r: univ→univ→univ, left, middle, right: univ,
                  left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = (left'.r).right| let c' = (left''.r).right| (c in c')
}
pred ExapndMiddle_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                 left_first: univ, left_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = (left'.r).right| let c' = (left''.r).right| (c in c')
}
pred ExapndRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
                              left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| (c in c')
}
pred ExapndRight_MiddleStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                             left_first: univ, left_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| (c in c')
}
pred ExapndMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
                              left_first: univ, left_next: univ→univ]{
```

```
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| (c in c')
}
pred ExapndMiddle_RightStatic_FirstLeftEmpty[r: univ→univ→univ, left, middle,
                                             right: univ, left_first: univ, left_next: univ→univ]{
 no left_first.r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| (c in c')
}
pred MutateRight[r: univ→univ→univ, left, middle, right: univ,
                                    left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)| eq[#c' ,# c]
}
pred MutateMiddle[r: univ→univ→univ, left, middle, right: univ,
                         left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = (left'.r).right| let c' = (left''.r).right| eq[#c' ,# c]
}
pred MutateRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
                                                 left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)| eq[#c' ,# c]
}
pred MutateMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
                                                  left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'| eq[#c' ,# c]
}
pred ContracttRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                                      left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)|(c' in c) and (c!inc')
}
pred ContracttRight[r: univ→univ→univ, left, middle, right: univ,
                                       left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)|(c' in c) and (c!inc')
}
pred ContracttMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                                       left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = (left'.r).right| let c' = (left''.r).right|(c' in c) and (c!inc')
}
pred ContracttMiddle[r: univ→univ→univ, left, middle, right: univ,
                                        left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = (left'.r).right| let c' = (left''.r).right|(c' in c) and (c!inc')
}
pred ContracttRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle,
                                               right: univ, left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|(c' in c) and (c!inc')
}
```

```
pred ContracttRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
                                 left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|(c' in c) and (c!inc')
}
pred ContracttMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle,
                                 right: univ, left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|(c' in c) and (c!inc')
}
pred ContracttMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
                                 left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|(c' in c) and (c!inc')
}
pred ContractRight_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                 left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)|(c' in c)
}
pred ContractRight[r: univ→univ→univ, left, middle, right: univ,
                   left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = middle.(left'.r)| let c' = middle.(left''.r)|(c' in c)
}
pred ContractMiddle_LastLeftEmpty[r: univ→univ→univ, left, middle, right: univ,
                                 left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = (left'.r).right| let c' = (left''.r).right|(c' in c)
}
pred ContractMiddle[r: univ→univ→univ, left, middle, right: univ,
                    left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
 let c = (left'.r).right| let c' = (left''.r).right|(c' in c)
}
pred ContractRight_MiddleStatic_LastLeftEmpty[r: univ→univ→univ, left, middle,
                                 right: univ, left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|(c' in c)
}
pred ContractRight_MiddleStatic[r: univ→univ→univ, left, middle, right: univ,
                                 left_first: univ, left_next: univ→univ]{
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all middle' : middle| let c = middle'.(left'.r)| let c' = middle'.(left''.r)|(c' in c)
}
pred ContractMiddle_RightStatic_LastLeftEmpty[r: univ→univ→univ, left, middle,
                                 right: univ, left_first: univ, left_next: univ→univ]{
 no last[left,left_next].r
 all left': left−last[left,left_next]| let left'' = left'.left_next|
  all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|(c' in c)
}
pred ContractMiddle_RightStatic[r: univ→univ→univ, left, middle, right: univ,
                                 left_first: univ, left_next: univ→univ]{
```

```
1168   all left': left−last[left,left_next]| let left'' = left'.left_next|
1169    all right' : right| let c = (left'.r).right'| let c' = (left''.r).right'|(c' in c)
1170 }
```

Table A.1 The Ternary Implication Lattice comprises 160 patterns, each represented as a node. An edge of the lattice encodes an implication relation between two patterns. The strongest patterns, i.e. sources in the lattice, are distinguished in bold with overline. Patterns in italic with underline on them are the weakest patterns or sinks of the lattice.

From	To
$\overline{\text{MutateHeaddOfRight\_MiddleStatic}}$	MutateHeadOfRight_MiddleStatic
$\overline{\text{ExpanddHeaddOfMiddle\_RightStatic\_FirstLeftEmpty}}$	ExpanddHeadOfMiddle_RightStatic_FirstLeftEmpty, ExpanddHeaddOfMiddle_RightStatic, ExpandHeaddOfMiddle_RightStatic_FirstLeftEmpty, ExpanddHeaddOfMiddle_FirstLeftEmpty
$\overline{\text{ContracttHeaddOfMiddle\_RightStatic\_LastLeftEmpty}}$	ContracttHeaddOfMiddle_LastLeftEmpty, ContracttMiddle_RightStatic_LastLeftEmpty, ContracttHeaddOfMiddle_RightStatic, ContractHeaddOfMiddle_RightStatic_LastLeftEmpty
$\overline{\text{ContracttTaillOfMiddle\_RightStatic\_LastLeftEmpty}}$	ContracttTaillOfMiddle_LastLeftEmpty, ContracttTaillOfMiddle_RightStatic, ContractTaillOfMiddle_RightStatic_LastLeftEmpty, ContracttTailOfMiddle_RightStatic_LastLeftEmpty
$\overline{\text{MutateTaillOfRight\_MiddleStatic}}$	MutateTailOfRight_MiddleStatic
$\overline{\text{ExpanddTaillOfRight\_MiddleStatic\_FirstLeftEmpty}}$	ExpanddTaillOfRight_FirstLeftEmpty, ExpanddTailOfRight_MiddleStatic_FirstLeftEmpty, ExpanddTaillOfRight_MiddleStatic, ExpandTaillOfRight_MiddleStatic_FirstLeftEmpty
$\overline{\text{MutateTaillOfMiddle\_RightStatic}}$	MutateTailOfMiddle_RightStatic
$\overline{\text{ContracttTaillOfRight\_MiddleStatic\_LastLeftEmpty}}$	ContracttTaillOfRight_MiddleStatic, ContractTaillOfRight_MiddleStatic_LastLeftEmpty, ContracttTaillOfRight_LastLeftEmpty, ContracttTailOfRight_MiddleStatic_LastLeftEmpty
$\overline{\text{MutateHeaddOfMiddle\_RightStatic}}$	MutateHeadOfMiddle_RightStatic
$\overline{\text{ContracttHeaddOfRight\_MiddleStatic\_LastLeftEmpty}}$	ContracttHeaddOfRight_LastLeftEmpty, ContracttRight_MiddleStatic_LastLeftEmpty, ContractHeaddOfRight_MiddleStatic_LastLeftEmpty, ContracttHeaddOfRight_MiddleStatic

Table A.2 Continue Table A.1

From	To
ExpanddHeaddOfRight_MiddleStatic_FirstLeftEmpty	ExpanddHeaddOfRight_FirstLeftEmpty, ExpandHeaddOfRight_MiddleStatic_FirstLeftEmpty, ExpanddHeadOfRight_MiddleStatic_FirstLeftEmpty, ExpanddHeaddOfRight_MiddleStatic
ExpanddTaillOfMiddle_RightStatic_FirstLeftEmpty	ExpandTaillOfMiddle_RightStatic_FirstLeftEmpty, ExpanddTaillOfMiddle_RightStatic, ExpanddTailOfMiddle_RightStatic_FirstLeftEmpty, ExpanddTaillOfMiddle_FirstLeftEmpty
MutateHeadOfRight_MiddleStatic	MutateRight_MiddleStatic
ExpanddHeadOfMiddle_RightStatic_FirstLeftEmpty	ExpanddMiddle_RightStatic_FirstLeftEmpty, ExpandHeadOfMiddle_FirstLeftEmpty, ExpanddHeadOfMiddle_RightStatic, ExpandHeadOfMiddle_RightStatic_FirstLeftEmpty
ExpanddHeaddOfMiddle_RightStatic	ExpandHeaddOfMiddle_RightStatic, ExpanddHeaddOfMiddle, ExpanddHeadOfMiddle_RightStatic
ExpandHeaddOfMiddle_RightStatic_FirstLeftEmpty	ExpandHeaddOfMiddle_RightStatic, ExpandHeadOfMiddle_RightStatic_FirstLeftEmpty
ExpanddHeaddOfMiddle_FirstLeftEmpty	ExpanddHeadOfMiddle_FirstLeftEmpty, ExpanddHeaddOfMiddle, ExpandHeaddOfMiddle_FirstLeftEmpty
ContracttHeaddOfMiddle_LastLeftEmpty	ContractHeaddOfMiddle_LastLeftEmpty, ContracttHeaddOfMiddle, ContracttMiddle_LastLeftEmpty
ContracttMiddle_RightStatic_LastLeftEmpty	ContracttMiddle_RightStatic, ContractMiddle_RightStatic_LastLeftEmpty, ContractTaillOfRight_LastLeftEmpty, ContractHeaddOfRight_LastLeftEmpty
ContracttHeaddOfMiddle_RightStatic	ContracttMiddle_RightStatic, ContracttHeaddOfMiddle, ContractHeaddOfMiddle_RightStatic
ContractHeaddOfMiddle_RightStatic_LastLeftEmpty	ContractMiddle_RightStatic_LastLeftEmpty, ContractHeaddOfMiddle_RightStatic
ContracttTaillOfMiddle_LastLeftEmpty	ContracttTailOfMiddle_LastLeftEmpty, ContractTaillOfMiddle_LastLeftEmpty, ContracttTaillOfMiddle
ContracttTaillOfMiddle_RightStatic	ContracttTaillOfMiddle, ContracttTailOfMiddle_RightStatic, ContractTaillOfMiddle_RightStatic

Table A.3 Continue Table A.1

From	To
ContractTaillOfMiddle_RightStatic_LastLeftEmpty	ContractTailOfMiddle_RightStatic_LastLeftEmpty, ContractTaillOfMiddle_RightStatic
ContracttTailOfMiddle_RightStatic_LastLeftEmpty	ContracttMiddle_RightStatic_LastLeftEmpty, ContractTailOfMiddle_RightStatic_LastLeftEmpty, ContractTailOfMiddle_LastLeftEmpty, ContracttTailOfMiddle_RightStatic
MutateTailOfRight_MiddleStatic	MutateRight_MiddleStatic
ExpanddTaillOfRight_FirstLeftEmpty	ExpanddTaillOfRight, ExpanddTailOfRight_FirstLeftEmpty, ExpandTaillOfRight_FirstLeftEmpty
ExpanddTailOfRight_MiddleStatic_FirstLeftEmpty	ExpandTailOfRight_MiddleStatic_FirstLeftEmpty, ExpanddTailOfRight_MiddleStatic, ExpandTailOfRight_FirstLeftEmpty, ExpanddRight_MiddleStatic_FirstLeftEmpty
ExpanddTaillOfRight_MiddleStatic	ExpanddTaillOfRight, ExpandTaillOfRight_MiddleStatic, ExpanddTailOfRight_MiddleStatic
ExpandTaillOfRight_MiddleStatic_FirstLeftEmpty	ExpandTailOfRight_MiddleStatic_FirstLeftEmpty, ExpandTaillOfRight_MiddleStatic
MutateTailOfMiddle_RightStatic	MutateMiddle_RightStatic
ContracttTaillOfRight_MiddleStatic	ContracttTaillOfRight, ContracttTailOfRight_MiddleStatic, ContractTaillOfRight_MiddleStatic
ContractTaillOfRight_MiddleStatic_LastLeftEmpty	ContractTailOfRight_MiddleStatic_LastLeftEmpty, ContractTaillOfRight_MiddleStatic
ContracttTaillOfRight_LastLeftEmpty	ContracttTailOfRight_LastLeftEmpty, ContracttTaillOfRight, ContractTaillOfRight_LastLeftEmpty
ContracttTailOfRight_MiddleStatic_LastLeftEmpty	ContracttRight_MiddleStatic_LastLeftEmpty, ContractTailOfRight_MiddleStatic_LastLeftEmpty, ContractTailOfRight_LastLeftEmpty, ContracttTailOfRight_MiddleStatic
MutateHeadOfMiddle_RightStatic	MutateMiddle_RightStatic
ContracttHeaddOfRight_LastLeftEmpty	ContracttHeaddOfRight, ContractHeaddOfRight_LastLeftEmpty, ContracttRight_LastLeftEmpty
ContracttRight_MiddleStatic_LastLeftEmpty	ContractHeaddOfMiddle_LastLeftEmpty, ContracttRight_MiddleStatic, ContractMiddle_RightStatic_LastLeftEmpty, ContractTaillOfMiddle_LastLeftEmpty

Table A.4 Continue Table A.1

From	To
ContractHeaddOfRight_MiddleStatic_LastLeftEmpty	ContractHeaddOfRight_MiddleStatic, ContractMiddle_RightStatic_LastLeftEmpty
ContracttHeaddOfRight_MiddleStatic	ContractHeaddOfRight_MiddleStatic, ContracttRight_MiddleStatic, ContracttHeaddOfRight
ExpanddHeaddOfRight_FirstLeftEmpty	ExpandHeaddOfRight_FirstLeftEmpty, ExpanddHeaddOfRight, ExpanddHeadOfRight_FirstLeftEmpty
ExpandHeaddOfRight_MiddleStatic_FirstLeftEmpty	ExpandHeadOfRight_MiddleStatic_FirstLeftEmpty, ExpandHeaddOfRight_MiddleStatic
ExpanddHeadOfRight_MiddleStatic_FirstLeftEmpty	ExpandHeadOfRight_FirstLeftEmpty, ExpandHeadOfRight_MiddleStatic_FirstLeftEmpty, ExpanddRight_MiddleStatic_FirstLeftEmpty, ExpanddHeadOfRight_MiddleStatic
ExpanddHeaddOfRight_MiddleStatic	ExpanddHeaddOfRight, ExpandHeaddOfRight_MiddleStatic, ExpanddHeadOfRight_MiddleStatic
ExpandTaillOfMiddle_RightStatic_FirstLeftEmpty	ExpandTailOfMiddle_RightStatic_FirstLeftEmpty, ExpandTaillOfMiddle_RightStatic
ExpanddTaillOfMiddle_RightStatic	ExpanddTailOfMiddle_RightStatic, ExpandTaillOfMiddle_RightStatic, ExpanddTaillOfMiddle
ExpanddTailOfMiddle_RightStatic_FirstLeftEmpty	ExpandTailOfMiddle_FirstLeftEmpty, ExpandTailOfMiddle_RightStatic_FirstLeftEmpty, ExpanddTailOfMiddle_RightStatic, ExpanddMiddle_RightStatic_FirstLeftEmpty
ExpanddTaillOfMiddle_FirstLeftEmpty	ExpanddTailOfMiddle_FirstLeftEmpty, ExpanddTaillOfMiddle, ExpandTaillOfMiddle_FirstLeftEmpty
MutateRight_MiddleStatic	MutateTaillOfMiddle, MutateHeaddOfMiddle, ContractHeaddOfMiddle, ExpandTaillOfMiddle, ExpandHeaddOfMiddle, ContractTaillOfMiddle
ExpanddMiddle_RightStatic_FirstLeftEmpty	ExpandHeaddOfRight_FirstLeftEmpty, ExpandTaillOfRight_FirstLeftEmpty, ExpanddMiddle_RightStatic, ExapndMiddle_RightStatic_FirstLeftEmpty

Table A.5 Continue Table A.1

From	To
ExpandHeadOfMiddle_FirstLeftEmpty	ExpandHeadOfMiddle, ExapndMiddle_FirstLeftEmpty
ExpanddHeadOfMiddle_RightStatic	ExpandHeadOfMiddle_RightStatic, ExpanddMiddle_RightStatic, ExpandHeadOfMiddle
ExpandHeadOfMiddle_RightStatic_FirstLeftEmpty	ExpandHeadOfMiddle_RightStatic, ExapndMiddle_RightStatic_FirstLeftEmpty
ExpandHeaddOfMiddle_RightStatic	ExpandHeadOfMiddle_RightStatic
ExpanddHeaddOfMiddle	ExpandHeaddOfMiddle, ExpanddHeadOfMiddle
ExpanddHeadOfMiddle_FirstLeftEmpty	ExpanddMiddle_FirstLeftEmpty, ExpandHeadOfMiddle_FirstLeftEmpty, ExpanddHeadOfMiddle
ExpandHeaddOfMiddle_FirstLeftEmpty	ExpandHeadOfMiddle_FirstLeftEmpty, ExpandHeaddOfMiddle
ContractHeaddOfMiddle_LastLeftEmpty	ContractMiddle_LastLeftEmpty, ContractHeaddOfMiddle
ContracttHeaddOfMiddle	ContractHeaddOfMiddle, ContracttMiddle
ContracttMiddle_LastLeftEmpty	ContractMiddle_LastLeftEmpty, ContracttMiddle
ContracttMiddle_RightStatic	ContractMiddle_RightStatic
ContractMiddle_RightStatic_LastLeftEmpty	ContractMiddle_LastLeftEmpty, ContractMiddle_RightStatic, ContractRight_LastLeftEmpty
ContractTaillOfRight_LastLeftEmpty	ContractTailOfRight_LastLeftEmpty, ContractTaillOfRight
ContractHeaddOfRight_LastLeftEmpty	ContractHeaddOfRight, ContractRight_LastLeftEmpty
ContractHeaddOfMiddle_RightStatic	ContractMiddle_RightStatic
ContracttTailOfMiddle_LastLeftEmpty	ContracttTailOfMiddle, ContractTailOfMiddle_LastLeftEmpty, ContracttMiddle_LastLeftEmpty
ContractTaillOfMiddle_LastLeftEmpty	ContractTailOfMiddle_LastLeftEmpty, ContractTaillOfMiddle
ContracttTaillOfMiddle	ContracttTailOfMiddle, ContractTaillOfMiddle
ContracttTailOfMiddle_RightStatic	ContracttMiddle_RightStatic, ContractTailOfMiddle_RightStatic, ContractTailOfMiddle
ContractTaillOfMiddle_RightStatic	ContractTailOfMiddle_RightStatic

Table A.6 Continue Table A.1

From	To
ContractTailOfMiddle_RightStatic_LastLeftEmpty	ContractTailOfMiddle_RightStatic, ContractMiddle_RightStatic_LastLeftEmpty
ContractTailOfMiddle_LastLeftEmpty	ContractMiddle_LastLeftEmpty, ContractTailOfMiddle
ExpanddTaillOfRight	ExpandTaillOfRight, ExpanddTailOfRight
ExpanddTailOfRight_FirstLeftEmpty	ExpandTailOfRight_FirstLeftEmpty, ExpanddTailOfRight, ExpanddRight_FirstLeftEmpty
ExpandTaillOfRight_FirstLeftEmpty	ExpandTaillOfRight, ExpandTailOfRight_FirstLeftEmpty
ExpandTailOfRight_MiddleStatic_FirstLeftEmpty	ExpandTailOfRight_MiddleStatic, ExapndMiddle_RightStatic_FirstLeftEmpty
ExpanddTailOfRight_MiddleStatic	ExpandTailOfRight_MiddleStatic, ExpandTailOfRight, ExpanddRight_MiddleStatic
ExpandTailOfRight_FirstLeftEmpty	ExapndRight_FirstLeftEmpty, ExpandTailOfRight
ExpanddRight_MiddleStatic_FirstLeftEmpty	ExpanddRight_MiddleStatic, ExapndMiddle_RightStatic_FirstLeftEmpty, ExpandHeaddOfMiddle_FirstLeftEmpty, ExpandTaillOfMiddle_FirstLeftEmpty
ExpandTaillOfRight_MiddleStatic	ExpandTailOfRight_MiddleStatic
MutateMiddle_RightStatic	ExpandTaillOfRight, ContractHeaddOfRight, MutateHeaddOfRight, MutateTaillOfRight, ContractTaillOfRight, ExpandHeaddOfRight
ContracttTaillOfRight	ContracttTailOfRight, ContractTaillOfRight
ContracttTailOfRight_MiddleStatic	ContracttRight_MiddleStatic, ContractTailOfRight_MiddleStatic, ContractTailOfRight
ContractTaillOfRight_MiddleStatic	ContractTailOfRight_MiddleStatic
ContractTailOfRight_MiddleStatic_LastLeftEmpty	ContractTailOfRight_MiddleStatic, ContractMiddle_RightStatic_LastLeftEmpty
ContracttTailOfRight_LastLeftEmpty	ContractTailOfRight_LastLeftEmpty, ContracttTailOfRight, ContracttRight_LastLeftEmpty
ContractTailOfRight_LastLeftEmpty	ContractRight_LastLeftEmpty, ContractTailOfRight
ContracttHeaddOfRight	ContracttRight, ContractHeaddOfRight

Table A.7 Continue Table A.1

From	To
ContracttRight_LastLeftEmpty	ContracttRight, ContractRight_LastLeftEmpty
ContracttRight_MiddleStatic	ContractMiddle_RightStatic
ContractHeaddOfRight_MiddleStatic	ContractMiddle_RightStatic
ExpandHeaddOfRight_FirstLeftEmpty	ExpandHeadOfRight_FirstLeftEmpty, ExpandHeaddOfRight
ExpanddHeaddOfRight	ExpandHeaddOfRight, ExpanddHeadOfRight
ExpanddHeadOfRight_FirstLeftEmpty	ExpandHeadOfRight_FirstLeftEmpty, ExpanddRight_FirstLeftEmpty, ExpanddHeadOfRight
ExpandHeadOfRight_MiddleStatic_FirstLeftEmpty	ExpandHeadOfRight_MiddleStatic, ExapndMiddle_RightStatic_FirstLeftEmpty
ExpandHeaddOfRight_MiddleStatic	ExpandHeadOfRight_MiddleStatic
ExpandHeadOfRight_FirstLeftEmpty	ExapndRight_FirstLeftEmpty, ExpandHeadOfRight
ExpanddHeadOfRight_MiddleStatic	ExpandHeadOfRight_MiddleStatic, ExpanddRight_MiddleStatic, ExpandHeadOfRight
ExpandTailOfMiddle_RightStatic_FirstLeftEmpty	ExpandTailOfMiddle_RightStatic, ExapndMiddle_RightStatic_FirstLeftEmpty
ExpandTaillOfMiddle_RightStatic	ExpandTailOfMiddle_RightStatic
ExpanddTailOfMiddle_RightStatic	ExpandTailOfMiddle_RightStatic, ExpandTailOfMiddle, ExpanddMiddle_RightStatic
ExpanddTaillOfMiddle	ExpanddTailOfMiddle, ExpandTaillOfMiddle
ExpandTailOfMiddle_FirstLeftEmpty	ExpandTailOfMiddle, ExapndMiddle_FirstLeftEmpty
ExpanddTailOfMiddle_FirstLeftEmpty	ExpandTailOfMiddle_FirstLeftEmpty, ExpanddMiddle_FirstLeftEmpty, ExpanddTailOfMiddle
ExpandTaillOfMiddle_FirstLeftEmpty	ExpandTailOfMiddle_FirstLeftEmpty, ExpandTaillOfMiddle
MutateTaillOfMiddle	MutateTailOfMiddle
MutateHeaddOfMiddle	MutateHeadOfMiddle
ContractHeaddOfMiddle	*ContractMiddle*
ExpandTaillOfMiddle	ExpandTailOfMiddle
ExpandHeaddOfMiddle	ExpandHeadOfMiddle
ContractTaillOfMiddle	ContractTailOfMiddle

Table A.8 Continue Table A.1

From	To
ExpanddMiddle_RightStatic	ExapndMiddle_RightStatic
ExapndMiddle_RightStatic_FirstLeftEmpty	ExapndRight_FirstLeftEmpty, ExapndMiddle_RightStatic, ExapndMiddle_FirstLeftEmpty
ExpandHeadOfMiddle	*ExapndMiddle*
ExapndMiddle_FirstLeftEmpty	*ExapndMiddle*
ExpandHeadOfMiddle_RightStatic	ExapndMiddle_RightStatic
ExpanddHeadOfMiddle	ExpanddMiddle, ExpandHeadOfMiddle
ExpanddMiddle_FirstLeftEmpty	ExpanddMiddle, ExapndMiddle_FirstLeftEmpty
ContractMiddle_LastLeftEmpty	*ContractMiddle*
ContracttMiddle	*ContractMiddle*
ContractMiddle_RightStatic	*ContractRight*, *ContractMiddle*
ContractRight_LastLeftEmpty	*ContractRight*
ContractTaillOfRight	ContractTailOfRight
ContractHeaddOfRight	*ContractRight*
ContracttTailOfMiddle	ContracttMiddle, ContractTailOfMiddle
ContractTailOfMiddle_RightStatic	ContractMiddle_RightStatic
ContractTailOfMiddle	*ContractMiddle*
ExpandTaillOfRight	ExpandTailOfRight
ExpanddTailOfRight	ExpandTailOfRight, ExpanddRight
ExpanddRight_FirstLeftEmpty	ExapndRight_FirstLeftEmpty, ExpanddRight
ExpandTailOfRight_MiddleStatic	ExapndMiddle_RightStatic
ExpandTailOfRight	*ExapndRight*
ExpanddRight_MiddleStatic	ExapndMiddle_RightStatic
ExapndRight_FirstLeftEmpty	*ExapndRight*
MutateHeaddOfRight	MutateHeadOfRight
MutateTaillOfRight	MutateTailOfRight
ExpandHeaddOfRight	ExpandHeadOfRight
ContracttTailOfRight	ContracttRight, ContractTailOfRight
ContractTailOfRight_MiddleStatic	ContractMiddle_RightStatic
ContractTailOfRight	*ContractRight*
ContracttRight	*ContractRight*
ExpanddHeadOfRight	ExpanddRight, ExpandHeadOfRight

Glossary