Core Extraction and Non-Example Generation: Debugging and Understanding Logical Models

by

Robert Morrison Seater

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

Masters of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[June 2005]
November 2004

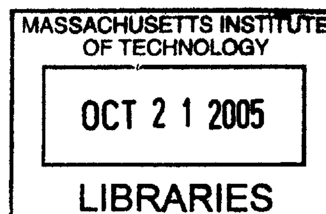
© Massachusetts Institute of Technology 2004. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
November 30, 2004

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

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

BARKER


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 2 1 2005
LIBRARIES

Core Extraction and Non-Example Generation: Debugging and Understanding Logical Models

by

Robert Morrison Seater

Submitted to the Department of Electrical Engineering and Computer Science
on November 30, 2004, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science and Engineering

Abstract

Declarative models, in which conjunction and negation are freely used, are a powerful tool for software specification and verification. Unfortunately, tool support for developing and debugging such models is limited. The challenges to developing such tools are twofold: technical information must be extracted from the model, then that information must be presented to the user in way that is both meaningful and manageable. This document introduces two such techniques to help fill the gap. *Non-example generation* allows the user to ask for the role of a particular subformula in a model. A formula's role is explained in terms of how the set of satisfying solutions to the model would change were that subformula removed or altered. *Core extraction* helps detect and localize unintentional overconstraint, in which real counterexamples are masked by bugs in the model. It leverages recent advances in SAT solvers to identify irrelevant portions of an unsatisfiable model. Experiences are reported from applying these two techniques to a variety of existing models.

Thesis Supervisor: Daniel N. Jackson
Title: Associate Professor

Acknowledgments

This research was supported by grant 0086154 (*Design Conformant Software*) from the ITR program of the National Science Foundation, by grant 6895566 (*Safety Mechanisms for Medical Software*) from the ITR program of the National Science Foundation, and by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

The core extraction portion of this thesis (Chapter 3) is isomorphic to a paper co-authored with Ilya Shlyakhter, along with Manu Sridharan, Daniel Jackson, and Mana Taghdiri [19]. In addition, the author is grateful for the insight and feedback given by his advisor Dr. Daniel Jackson and colleagues Gregory Dennis, Emina Torlak, Derek Rayside, Dr. Robert Miller, and Dr. Michael Ernst.

This work was completed while the author was a member of the Software Design Group (SDG), a research group at the Computer Science and Artificial Intelligence Labratory (CSAIL) at the Massachusetts Institute of Technology (MIT).

Contents

1 Declarative Modeling: Benefits and Drawbacks **21**

1.0.1 Benefits of Declarative Modeling . . . 22

1.0.2 Under- and Over-constraint . . . 24

1.0.3 Writing and Analyzing Declarative Models . . . 26

1.1 The Alloy Language . . . 27

1.2 Errors: Modeling Bugs vs. System Faults . . . 27

2 Non-Example Generation: Explaining Subformulae **29**

2.1 Motivating Non-Examples . . . 30

2.1.1 Windows and Rain . . . 30

2.1.2 Counterfactual Reasoning? . . . 32

2.1.3 Events vs. Policies . . . 36

2.1.4 What is the Role of a Formula? . . . 37

2.2 Formalization and Representation: Conjunction Diagrams . . . 39

2.2.1 Deletion (and Sabotage) Formalized . . . 39

2.2.2 Conjunction Diagrams . . . 40

2.2.3 Representing Sabotage . . . 42

2.2.4 Theorems . . . 43

2.3 Propositional Logic Examples . . . 46

2.3.1 Trivial Examples . . . 46

2.3.2 CNF Example . . . 48

2.3.3 Using *Expansion* to Explore a Model . . . 50

2.3.4 Making Dinner . . . 51

2.4 Handling Rich Logics . . . 52
2.4.1 Non-Boolean Values . . . 53
2.4.2 Defined Names and Other Syntactic Sugars . . . 54
2.5 Alloy Examples . . . 57
2.5.1 Ceilings and Floors . . . 58
2.5.2 Sequence Library . . . 64
2.5.3 The Firewire Network Protocol . . . 66
2.6 Expansion and Refinement . . . 80
2.7 Related Work . . . 80
2.7.1 Understanding Counterexamples with `explain` . . . 80
2.7.2 Mutation Testing . . . 81
2.7.3 Logic Programming: Prolog . . . 82
2.7.4 Near Miss Learning . . . 84

3 Core Extraction: Identifying Overconstraint 87
3.1 Introduction . . . 88
3.2 A Toy Example . . . 90
3.3 The Core Extraction Algorithm . . . 95
3.3.1 Constraint Language . . . 96
3.3.2 Translation . . . 96
3.3.3 Mapping Back . . . 98
3.3.4 Complications . . . 99
3.4 Experience . . . 99
3.4.1 Common Mistakes . . . 100
3.4.2 Locating Known Overconstraints . . . 102
3.4.3 Blunders Discovered . . . 105
3.4.4 Performance . . . 106
3.5 Related work . . . 107
3.6 Conclusions . . . 108

A Firewire Alloy Model 113

B Proofs **121**

B.1 Deletion and Sabotage are Sufficient to Compute Correctness . . 121

B.2 Disjoint Entries 123

B.3 Empty Entries 124

List of Figures

2-1 An Alloy model of the relation between the weather, the status of your bedroom window, and whether or not you get wet. 31

2-2 There are only 6 pairs of entries in a conjunction diagram which are not always disjoint. 45

2-3 An Alloy model written by Daniel Jackson about Paul Simon's song "One Man's Ceiling Is Another Man's Floor". 58

2-4 A counterexample to the claim that one man's floor is another man's ceiling. 59

2-5 The role of the `PaulSimon` constraint 60

2-6 A counterexample to the claim that the `Geometry` constraint is sufficient to force each man's floor to be some man's ceiling 61

2-7 The role of the `Geometry` constraint 62

2-8 The role of the presence of the `NoSharing` constraint 63

2-9 A simplified version of Alloy's sequence module 65

2-10 The role of part of the `add` predicate in the sequence module. An arc labeled `next[seq0]` from node `widget1` to node `widget0` means that, in the sequence `[seq0]`, `widget1` immediately preceeds `widget0`. . . . 67

2-11 In state `S0`, the system is initialized and the nodes are waiting. Each link has a message queue which is initially empty. `Node0` has only one incoming link, and thus it has only one incoming link which has not been classified as a `parentLink`. In state `S1`, `Node0` becomes active and sends a request to `Node1`, declaring its willingness to be a child. . 71

2-12 In state S2, `Node2` acknowledges `Node0`'s request, indicating that `Node2` is willing to be `Node0`'s parent. `Node0` thus declares that its outgoing link is a `parentLink`, making it a leaf in the tree being constructed. In state S3, `Node2` becomes active and notices that it how has only one outgoing link which is not marked as being a `parentLink`. It thus sends a request to `Node1`, indicating its willingness to be a parent. . . 72

2-13 In state S4, `Node1` acknowledges `Node2`'s request, indicating its willingness to be a child. Node2 sees that only one of its incoming links is not a `parentLink`, so it sends a request to `Node1` that the other incoming link be made a `parentLink`. In state S5, `Node1` activates and sees that it can only be a leaf node. 73

2-14 In state S6, all of `Node2`'s incoming links are `parentLinks`, and those choices have been validated by its neighbors. `Node2` thus declares itself to be the root, and the algorithm terminates. 74

2-15 One state from a trace in *PCD* from the entire constraint describing of the `stutter` operation 75

2-16 *PCD* from the conjunction diagram for the role of the latter half of the constraint `s'.op = Stutter => SameState (s, s')` 76

2-17 The formula `n in s.active` in the specification for the `Elect` operator is crucial to disallowing the case where an inactive node is elected as the root. Shown, are states S5 and S6 (the final two states) of one such trace. This trace would appear in the *PCD* entry of the conjunction diagram. 79

3-1 A toy Alloy model describing the behavior of a web cache 91

3-2 The corrected version of the web cache Alloy model, taking into account the information provided by core extraction 94

3-3 *A Roadmap to Core Extraction.* (1) A model is created in any constraint language which is reducible to SAT in a structure preserving fashion. (2) During translation to CNF, each clause generated is annotated with the AST node from which the clause was produced. (3) A SAT solver (used as a black box) determines that the model is unsatisfiable and extracts an unsatisfiable core (a subset of the CNF clauses which is also unsatisfiable). (4) The core is mapped back to the original model by marking (as "relevant") any part of the AST indicated by the annotation of any clause in the CNF core. The analysis is now complete. The remaining steps concern guarantees made to the user about what the markings on the AST mean; they are not actually executed during normal use of the tool. (5) The user is guaranteed that changing the unmarked (non-relevant) portions of the AST will leave the model unsatisfiable. (6) Specifically, the CNF corresponding to the altered AST will be a superset of the unsatisfiable core previously extracted, and thus will itself be unsatisfiable. 109

3-4 Translation of AST to CNF, and mapping back of unsatisfiable core. The AST is for the (trivially unsatisfiable) Alloy formula of the form "(some p) && (no p) && ...". To each node, a sequence of Boolean variables (b1 through b6) is allocated to represent the node's value. From each inner node, translation produces a set of clauses relating the node's Boolean variables to its childrens' Boolean variables. The highlighted clauses form an unsatisfiable core, which is mapped back to the highlighted AST nodes. 110

List of Tables

2.1 A generic conjunction diagram with column formulae Col_1 and Col_2 and row formulae Row_1 and Row_2 41

2.2 The generic conjunction diagram for deletion 41

2.3 The conjunction diagram for sabotage 43

2.4 The conjunction diagram for deletion and sabotage 43

2.5 The role of the final clause of a CNF formula 44

2.6 The role of B in $A \vee B$ 46

2.7 The role of B in $A \wedge B$ 47

2.8 Our intuition for the role of a CNF clause 48

2.9 Our intuition for the role of a term in a CNF clause 49

2.10 The role of a redundant CNF clause 49

2.11 The role of a term in a redundant CNF clause 50

2.12 The role of $\neg WifeLate$ in the dinner example 51

Introduction

Suppose you are a software developer and have formulated an important aspect of your program as a logical model. Now you want to make sure that your model is saying what you think it is saying and that you are correctly interpreting the results it gives you.

Consider this excerpt from an Alloy model which is discussed in depth later on.

```
pred Geometry (){no m: Man | m.floor = m.ceiling}
```

What role does this constraint play in the model? Is it necessary? What would have happened if it had been written it incorrectly? Is it allowing solutions which ought to be eliminated? Is it eliminating solutions that ought to be allowed?

The answers to these questions could probably be deduced through careful reasoning but (a) it would be a lot of work, and (b) there would always remain a lingering doubt that you missed something. Such concerns beg for automated tool support. We present two such tools to help the user answer these questions.

Explaining Roles

When looking at a logical model, it is natural to ask what role some particular constraint plays in that model. But what sort of an animal is the role of a subformula? Logicians and philosophers have addressed this issue in a number of ways that involve extending the logic to include special constructs. In Chapter 2, we propose *non-example generation* as an appealing, lightweight alternative. This technique explains the role of a formula by calculating what would happen differently if that formula

were absent or altered. We introduce *conjunction diagrams* as a notation to present such information.

Determining the effect of deleting a suformula T is not as simple as removing T and re-analyzing the model. Suppose, after removing T, your checker produces a new solution. Was it a solution before you removed T? Are the old solution still solutions? What you really want to know is what new solutions were added and what old solutions have been inhibited. A more sophisticated approach is required, and we introduce *non-example generation* as one such approach.

Overconstraint

Logical models are susceptible to unintentional overconstraint in which real counterexamples are masked by bugs in the model. In Chapter 3, we introduce *Core extraction*, a new analysis that mitigates this problem in the presence of a checker which translated the model to CNF. It exploits a recently developed facility of SAT solvers to deduce an unsatisfiable subset of a CNF which is often much smaller than the clause set as a whole. This unsatisfiable "core" is mapped back into the syntax of the original model, showing the user irrelevant fragments of the model. This information can be a great help in discovering and localizing overconstraint, sometimes pinpointing it immediately. The construction of the mapping between the model and an equivalent CNF is given for a generalized modeling language, along with a proof of the soundness of the claim that the marked portions of the model are irrelevant. Experiences in applying core extraction to a variety of existing models are discussed.

Summary

Chapter 1 provides background and puts the remaining chapters in context. We introduce declarative modeling and discuss its benefits and drawbacks.

Chapter 2 introduces *non-example generation*, a technique which explains the role of a subformula by computing solutions it is responsible for allowing or disallowing. We begin by motivating our approach as a lightweight alternative to counterfactual

reasoning, formalize it, and introduce *conjunction diagrams* as a convenient notation. We expand the technique to account for quantifiers and named predicates, then use it to examine and explain several Alloy models.

Chapter 3 introduces *core extraction*, a technique for detecting unintentional overconstraint by showing the user unused portion of the model. We introduce the technique in the context of a general constraint language, and discuss its implementation. We prove the correctness of the algorithm, and report on our experience in applying to several Alloy models.

Chapter 1

Declarative Modeling: Benefits and Drawbacks

"There is nothing so annoying as a good example!"

Mark Twain

Roughly speaking, there are two ways to model a transition system [1] [2]. In the *operational* idiom, transitions are expressed using assignment statements, either with the control flow of a conventional imperative program (as in Promela, the language of the Spin model checker [16]), or using a variant of Dijkstra's guarded commands (as in Murphi [11] and SMV [7]). In the *declarative* idiom, transitions are expressed with constraints, either on whole executions, or, more often, on individual steps. This idea is rooted in the early work on program verification; the operation specifications of the declarative languages VDM, Larch and Z are essentially the pre- and post-conditions of Hoare triples.

For readers unfamiliar with these idioms, it may help to think of an operational specification as one that gives a recipe for constructing new states from old ones, and a declarative specification as one that gives a fact that can be observed about the

[1] This introduction is largely lifted from Daniel Jackson's description of declarative modeling, as given in [19]. It has been altered to suite this document, but it retains much of his content, organization, and elements of his style.

[2] A *transition* system is a model which specifies states and actions which cause the system to transition between those states. A solution is thus a trace of states.

relationship between old and new states. An operational modeler asks "How would I make X happen?"; a declarative modeler asks "How would I recognize that X has happened?".

The advantage of the operational idiom is its executability. A simulation, either random or guided by inputs from the user, can give useful feedback to a designer. In model checking, the ability to generate a state's successor in a single computational step makes it possible to explore the reachable state space by depth-first search (as in Spin [7]). In contrast, declarative models have been viewed as not executable, and less amenable to automatic analysis in general, since even generating successors requires search. Recently, however, we have developed an analysis based on SAT that allows both simulation and systematic exploration of declarative models [22]. A common form of analysis that we perform is similar to bounded model checking [4]: the SAT solver is used to find traces that violate specified properties. In fact, earlier symbolic methods could also handle models with declarative elements. The earliest versions of SMV, for example, provided a construct for expressing transitions implicitly. Its analysis, being symbolic, was not hindered by the inability to generate successors of a state constructively.

The advantage of the declarative idiom is its expressibility. For some kinds of problem, especially the control-intensive aspects of a system, the operational idiom can be more natural and direct. But in many cases, especially for software systems, the declarative idiom is more flexible, more natural, and sometimes, surprisingly, more amenable to analysis. In many contexts, it is more natural and concise to use a declarative description; often one need only to describe the rules of the game and what it means to have won, and not ever think about what moves will be needed (and which must be avoided) in order to win.

1.0.1 Benefits of Declarative Modeling

Partial Descriptions. The declarative idiom better supports partial descriptions. Sometimes, only one operation is of interest. In a study of a name server [27], for example, only the lookup operation was modeled and analyzed, since the operations

for storing and distributing name records were straightforward. An explicit invariant on the structure of the name database took the place of the operations that in an operational model would define the reachable states implicitly. Even if the lookup operation were not written declaratively, the need to account for the invariant in generating initial states makes the description essentially declarative.

Underspecification. The need to constrain a model's behavior only loosely arises in many ways. It arises when implementation issues are to be postponed or ignored; analysis of a cache protocol, for example, can establish its correctness irrespective of the eviction policy. It arises when analyzing a family of systems: an analysis can check that a collection of design or style rules implies certain desirable properties, and thus that any system built in conformance with the rules will have those properties too. And it arises when accounting for an unpredictable environment: checking a railway signaling protocol, for example, for all possible train motions. In these cases, a declarative description is often succinct and natural, where an operational idiom would, in contrast, require an explicit enumeration of possibilities. Cache eviction, for example, might be specified by saying that the resulting cache, viewed as a set of address/value pairs, is a subset of the original cache. The motion of trains on a network might be specified by saying that the new track segment occupied by a train is either its old one, or one connected to it.

Analyzing Specifications. Specifications can be used not only as yardsticks of analysis, but also as subjects in their own right. It is easy to make mistakes writing specifications, so it helps to analyze their properties directly: to check that one follows from another, for example, or to generate executions over which specifications differ. If the model and specification are written in the same declarative language, 'masking' is possible. If the model `M` fails to have properties `P` and `Q`, we might want to know whether the problems are correlated. By checking the conjunction of `P` and `M` against `Q`, we can find out whether fixing `M` so that it satisfies `P` would also fix `M` with respect to `Q`. A declarative analyzer also helps refactoring; any fragment of a model or specification can be compared to a candidate replacement by conjecturing the equivalence of the two.

Non-Operational Problems. Some problems are simply not operational in nature, and demand a logical rather than a programmatic description. Alloy has been used, for example, to check the soundness of a refinement rule: this involved modeling state machines and their trace semantics, and checking that the rule related only machines with appropriately related semantics. Many subjects are well described in a rule-based manner: ontology models, security policies, and software architectural styles, for example.

Topology Constraints. Sometimes one particular aspect of a system has a declarative flavor. For example, many distributed algorithms are designed to work only if the network's topology takes some form, such as a ring or tree. A declarative model can be constructed that constrains the network appropriately, but does not limit it to a single topology. The analysis will then account for all executions over all acceptable topologies (for a network of some bounded size). The Firewire example described in Sections 2.5.3 and 3.4 exploits this benefit.

Avoiding Initialization. In some systems, normal operation is preceded by an initialization phase in which the system is configured. An operational description of such a system will suffer from traces that are made longer than necessary by their initialization prefixes. A declarative description can bypass the initialization phase with an invariant that captures its possible results, thus shortening the traces. The result is not only simpler description, uncluttered by the details of initialization, but also more efficient analysis, since a bounded model checking analysis can use a lower bound on trace length and still reach all states.

1.0.2 Under- and Over-constraint

"Example is the school of mankind, and [we] will learn at no other"

Edmund Burke

The very mechanisms that give declarative modeling its power – conjunction and negation – also bring a curse. Any modeler faces the dual risks of underconstraint and overconstraint. In a declarative setting, underconstraint turns out to be easy to

notice and correct, while overconstraint is both difficult to recognize and dangerous when it goes undetected.

Underconstraint An *under*constrained model does not eliminate all the situations which the user intended to eliminate. The consequence of underconstraint is the discovery of bogus or absurd counterexamples. When the user encounters such a counterexample, it is obvious that the model is underconstrained and usually straightforward to add a constraint to the model to eliminate the extraneous solution. Furthermore, if the user does not discover the underconstraint there is no harm done: If a valid counterexample is generated despite the underconstraint, the user still has found a fault in the system. If there are no counterexamples despite the underconstraint, then the user has inadvertently proven a stronger claim than intended.

Overconstraint It is unfortunately easy to write a model that has fewer behaviors than intended. An *over*constrained model does not allow some or all behaviors that the user intended to investigate. A check of a safety property may then pass only because the offending behavior has been accidentally ruled out (probably along with many other behaviors). In the extreme, an overconstrained model may have no solutions at all. As a result, there will be no counterexamples to any assertion! Such situations are easily detected simply by analyzing the model – effectively a liveness check[3]. The much more worrisome case is when the model is satisfiable, but bugs in the constraints have accidentally eliminated those cases which violate the assertion. The analyzer will (correctly) report that there are no counterexamples to the model the user actually wrote, and the user will (erroneously) interpret this to mean that there are no counterexamples to the model she intended to write; unlike with underconstraint, undetected overconstrained is a serious problem.

The risk can be mitigated by working carefully. One can exploit the ability to build and analyze a model incrementally, adding as few (and as weak) constraints as possible to establish the required safety properties. One can simulate the model

[3]If your analyzer can only check an asserted property, then one can perform a liveness check by asserting the property "True" and checking it

extensively, adding conditions to force execution of interesting cases. And of course one can formulate and check liveness properties, at least ruling out the most egregious overconstraints, such as those that lead to deadlock.

None of these approaches, however, counter the risk of overconstraint that is relevant to a particular safety property. The worst overconstraints are not the ones that rule out most behaviors, since they are usually easy to detect, but the ones that rule out exactly those behaviors that would violate the safety property. Since the purpose of checking a safety property is precisely to find behaviors that violate it, we are hardly likely to be able to formulate a liveness constraint to ensure that those behaviors are possible! And of course a liveness check can itself be confounded by an overconstraint that rules out those traces that would be counterexamples to the liveness check itself. A property-specific detection of overconstraint is thus required, and *core extraction*, introduced in Chapter 3, is exactly such an analysis.

1.0.3 Writing and Analyzing Declarative Models

Let's consider what the world looks like from the eyes of a modeler developing a declarative description about a subject system. The modeler begins with an empty set of constraints, allowing all possible worlds and behaviors. Each constraint added to the model eliminates some class of situations, and a complete model reduces the set of possibilities to a meaningful subset. Typically, the model eliminates those which *can*not occur in the subject system. That set of solutions is searched for a solution which violates a user-defined property – thus finding solutions which *should* not occur in the subject system. Solutions are reported to the user as counterexamples – situations which are possible but undesirable.

For example, one could write a model M with constraints that restrict the world to those cases where trains obey traffic signals. One might then write a safety property P which states that trains never collide. Solutions to the equation $M \wedge \neg P$ are counterexamples to the claim that traffic signals are sufficient to prevent collision.

In order to enumerate all the worlds satisfying the model, it is necessary to bound the size of the universe. Such a bound is known as the *scope* of the model. The

implication is that counterexamples are sound but not complete (although they are "complete up to scope"). The role of model checking is to find bugs, not to prove properties, and the assumption is that many bugs can be revealed by small examples [2]. However, by giving up the ability to prove a property for any bound (such as a theorem prover does), the search process is made completely automatic.

Each new conjunct serves to eliminate some set of solutions, but a subformula of one of those conjuncts may actually relax the model and increase the number of solutions. For example, a top-level constraint of the form "`A or B`" excludes solutions but the subformula "`B`" within that constraint allows solutions. In Chapter 2, we introduce *non-example generation*, which explains the role played by a subformula of a model by classifying it as either *relaxing* or *restraining*, and by computing solutions it is responsible for including or excluding.

1.1 The Alloy Language

In developing the process of non-example generation, we will start with symbolic logic and build up to more sophisticated logical languages, such as first order logic, relational logic, and eventually the Alloy modeling language [9, 21, 8] . The techniques described in this paper are not particular to the Alloy language. Rather, they are particular to logical, declarative modeling languages of which Alloy is an example.

1.2 Errors: Modeling Bugs vs. System Faults

One of the chief benefits of writing a model about a piece of software or other system is the discovery of faults in that system. This goal is reflected in the division of a model into a description of the system and a property (which typically asserts the absence of a particular class of faults). Of course, the model may iself contain errors. Bugs are a risk of any language, although certain kinds of bugs (such as overconstraint) are especially problematic for declarative models. It is important to distinguish *bugs* in the model from *faults* in the system being modeled. The techniques in this thesis

help the user resolve the tension and interaction between these two types of errors.

Chapter 2: *non-example generation*

Whether or not there are solutions to the model, we wish to understand the role that particular portions of the system play in preventing (or failing to prevent) faults. By understanding the role that a constraint actually plays, and contrasting it to the role we think it should play, we not only discover bugs in the model, but we also gain insight into why there are or no faults in the system.

Chapter 3: *core extraction*

The absence of counterexamples indicates that either there are no faults in the system or that bugs in the model are masking all of the system's faults. Unintentional overconstraint can often be found by identifying which parts of the model played a role in making it unsatisfiable (and which were irrelevant).

Chapter 2

Non-Example Generation: Explaining Subformulae

"Give me a fruitful error anytime,
full of seeds, bursting with its own corrections."
Vilfredo Pareto

When looking at a logical model, it is natural to ask what role some particular constraint plays in that model. But what sort of an animal is the role of a subformula? In this chapter we discuss some of the ways logicians and philosophers have addressed this issue, and we propose *non-example generation* as a lightweight alternative. This technique explains the role of a formula by calculating what would happen differently if that formula were absent or altered.

This chapter begins by motivating *non-example generation* as a lightweight alternative to counterfactual reasoning for explaining models. *Conjunction diagrams* are introduced as an effective notation both for presenting non-example information to the user and for proving handy properties about the technique. Equipped with a good notation, the technique can be extended beyond propositional logic to include first order logics, relational logics, and logics with declarations and named predicates. Non-example generation is applied to several existing models and experiences are reported.

2.1 Motivating Non-Examples

"A life spent making mistakes is not only more honorable,

but more useful than a life spent doing nothing."

George Bernard Shaw

One often wishes to know what would be the case if the world were slightly different than the way it currently is. When asked to explain a phenomenon, a common answer is to describe how that phenomenon would have differed had the world been slightly different. Understanding a situation means understanding the influence that small changes to the initial conditions of that situation have on its the outcome. Let's consider an example.

2.1.1 Windows and Rain

One morning, you notice that you left your bedroom window open overnight. Explaining to your child why this is a problem, you say something along the lines of "Had it rained, then I would have gotten wet.". One can also present the situation the other way around: having woken up, you notice that it rained last night. You explain why it is a good thing your window was closed by saying "Had I left the window open, then I would have gotten wet.". This example is deceptively simple. Figure 2-1 shows a formalization of it in the Alloy language.

The model begins by defining the set `State` and 8 subsets. The following fact ensures that opposite pairs of those subsets partition the set of states. E.g. every state is either in `Rain` or `Sunny` but not both. The `Physics` constraint describes the conditions under which you get wet. If the window is broken, then you get wet whenever it rains. If the window is intact, then you get wet whenever it rains while the window is open. The `Diligent` constraint ensures that you close the window whenever it rains.

At the end of the model, we ask to analyzer to show us an example of the system. We also check the `StayDry` assertion, which states that you're always dry if the window is intact. No such cases are found, and the assertion passes.

```
module rain

sig State {}
sig Rain, Sunny, Wet, Dry, Open, Closed, Broken, Intact in State {}

/* In any given state, the weather is either rainy or sunny, the
 * window is either open or closed, the window is either broken or
 * intact, and you are either wet or dry. */
fact Partitions {
  //the weather outside
    Rain + Sunny = State
    no Rain & Sunny
  //your bedroom window
    Open + Closed = State
    no Open & Closed
    Intact + Broken = State
    no Intact & Broken
  //your status
    Wet + Dry = State
    no Wet & Dry
}

/* If the window is intact, you get wet when the window is open and it
 * rain. If the window is broken, you get wet when it rains. */
fact Physics {
  all S: State | S in Intact =>
    (S in Rain and S in Open <=> S in Wet)
  all S: State | S in Broken =>
    (S in Rain <=> S in Wet)
}

/* You always close the window when it rains. */
fact Diligent {all S: State | S in Raining => S in Closed}

pred example() {}
run example for exactly 1 State expect 1

assert StayDry {all S: State | S in Intact => S in Dry}
check StayDry for exactly 1 State expect 0
```

Figure 2-1: An Alloy model of the relation between the weather, the status of your bedroom window, and whether or not you get wet.

Intuitively, the `Diligence` policy has something to do with why we manage to stay dry as long as the window is intact. How can we ask the model to verify and elaborate on that intuition?

2.1.2 Counterfactual Reasoning?

"If I had only known, I would have been a locksmith."

Albert Einstein

At first, this looks like a task for counterfactual reasoning. *Counterfactual reasoning* asks how the world would be different if some variable had taken on a different value. For example, we might counterfactually inquire "What would have happened had the window been left open last night?" and expect an answer along the lines of "I would have gotten wet.". Philosophers and logicians have worked on formalizing counterfactual reasoning in a number of ways. However, we will see that each solution either produces undesirable results or requires significant extensions to the logic in which the model is written. The task at hand is to answer questions about propositional logic, so requiring the user to rewrite her model in a different logic is unacceptable.

Adding to the World

Consider a model M from which you can draw some conclusion C. If we add a new fact N, stating our counterfactual supposition, we can draw some new conclusion C'[1].

$$M \vdash C$$

$$(M \wedge N) \vdash C'$$

Unfortunately, this approach tends to produce one of two useless results for C':

[1]Astute readers may object to the use of the proof symbol $\vdash$ when we are talking about model checkers whose finite limitations prevent them from producing proofs. For the purposes of this discussion, imagine that our analyzer can examine infinitely many possible worlds. The finite bound put on a model is an issue of analysis not of semantics; the model is written as if the world were arbitrarily large (or even infinite), and the bound is only enforced when we actually analyze the model. In principle, conclusions can be proven from models. In practice our method of analysis does not consitute proof.

1. **Contradiction:** Suppose the new constraint N contradicts the previous model M. For example, N might be "It rained last night and I left the window open."? This constraint contradicts the `Diligent` constraint already in M, so the new model, $M \wedge N$, is inconsistent. We would conclude that there are no possible worlds in which it rains and the window is open.

2. **Odd Causality:** Even in the case where N does not contradict M, C' may be a very odd conclusion. If we instead added the weaker constraint N = "I left the window open last night.", then analysis of the model $M \wedge N$ will tell us that it could not have rained. We have learned that opening our window prevents rain!

Entailment vs. Causality

In the second case, had we instead added the constraint "It rained last night", then the analyzer would tell us that we couldn't have opened the window. While this is not quite what we want (since we were hoping to learn that we might have gotten wet), it is at least consistent with our intuition about causality. Why does asking about changing the state of the window ("What if I had left the window open?") produce an unintuitive result yet asking about changing the state of the weather ("What if it had rained?") produce a sensivle result? Where does the asymmetry come from.

The problem is that `Diligent` is written in propositional logic, and thus uses *logical implication* not *causal implication.* In `Diligent`, we wrote that $(Rain \Rightarrow Closed)$. From that formula, the *law of the contrapositive* lets us conclude that $(\neg Closed \Rightarrow \neg Rain)$. However, the law of the contrapositive does not apply to *causal* implication; the formula (*Rain causes Closed*) does not imply that ($\neg$*Closed causes* $\neg$*Rain*).

Unfortunately, without adding a "causes" operator to the language, we cannot accurately represent causality, as discussed by Avi Sion [40] and John Stuart Mill [34].

Changing the World

To avoid producing the contradictions we saw in Section 2.1.2 when we added constraints to the model, we might consider instead *changing* part of the model. In this approach, an old fact N in replaced with a new fact N', thus producing a new (hopefully meaningful) conclusion C'.

$$M \vdash C$$
$$M[N \leftarrow N'] \vdash C'$$

The complication here is that you cannot just change part of the state without violating constraints in the model. As before, we cannot force the window to be open when it it raining without violating the `Diligent` constraint.

We are faced with a serious dilemma: Changing part of the world requires one of two approaches:

1. *Constraint Violation* One solution is to allow constraints to be violated if necessary. Unfortunately, we will end up with situations which violate fundamental aspects of the model. For example, we might generate the situation in which it is sunny and I close the window, yet I get wet (violates `Physics`). That is hardly an explanation of how the model works!

2. *Variable Alteration* The other solution is to update other variables until all constraints are satisfied. However, there may be several different ways to change the world, all of which satisfy the constraints. Offhand, it is unclear which one to choose.

Fixing Constraint Violation: Laws vs. Policies

One might try to avoid the problems with *constraint violation* by classifying facts into "fundamental laws", such as `Physics`, versus "preferred policies", such as `Diligent`. When making a counterfactual inquiry, one only allows *policies* to be violated. In our simple model, such a distinction would indeed avoid the problem – the diligence constraint may be violated but the physics constraint may not.

Such a distinction would require that the user annotate the model. Besides being extra work for the user, it might not even be feasible – the user is asking about the model in order to understand it better and may not feel qualified to make such distinctions.

Furthermore, imposing such a distinction limits the questions one can ask – why not allow the user to ask why a funamental law is necessary? For example, what roles does the `Physics` constraint play in our toy model?

Even worse, this solution does not always avoid our previous dilemma. If our supposition violates a fundamental law then we will have to either change the value of some of the other variables or accept that one of the fundamental law is violated. We are back where we started.

Fixing Variable Alteration: Closest Possible World

Since *constraint violation* seemes irreparable, let's look at how *variable alteration* can be improved. David Lewis [31] developed an approach to change the world to satisfy all the constraints, but to do so in a deterministic and sensible manner. In order to determine what would have happened if some event E had occurred, consider the closest possible world in which E holds. In this manner, we only consider worlds in which all the constraints are satisfied, but use a distance metric do decide which other variables should be changed. See Section 2.7.4 for his exact words on the matter.

In our simple example, this seems to work. Given that last night we observed the state

$$Rain, Closed, Intact, Dry$$

we ask "What if the window had been open?". By the Closest Possible World definition, we want the state which is closest to the state

$$Rain, Open, Intact, Dry$$

but which satisfies the model. If we use a simple bit-comparison metric (the distance between two states is the number of variables on which those states differ), then

closest such state is

$$Rain, Open, Intact, Wet$$

which is exactly what we want to see; if we had left the window open in the rain, then we would have gotten wet.

The chief objections to this approach are the oddities which develop in very detailed models. The traditional example of an unintuitive result is the so called *Nixon Paradox*, originally pointed by Kit Fine in 1975 [12]: "What would have happened if Nixon had pushed the button to launch nuclear weapons at Russia?". We expect an answer along the lines of "Russia would have retaliated, reducing both nations to radioactive ash.". However, if we look at the closest possible world to our own in which nixon pushed the button, we instead conclude "There would have been an electrical failure so that the button did nothing, after which Nixon would have come to his senses.".

Problems such as the Nixon Paradox arise when the model becomes very detailed and models unlikely events and corner cases. The model analyzer has no notion of probability and so it will happily make rare events occur if doing so will produce a solution which has a shorter distance. To correct this shortcoming, the model needs a notion of probability – when selecting the "closest possible world" the possibilities need to be weighted by likelihood as well as similarity. Extending traditional logic to include such a notion is an interesting avenue, but it is not the task at hand.

2.1.3 Events vs. Policies

So far, the only ways we've seen to do counterfactual reasoning involve making non-trivial additions to our logic. In fact, there is a solution to this problem which does not involve counterfactual reasoning and all the complexities that go along with it. Rather than asking the role of a bit of the state (e.g. whether or not the window was open), consider asking about the role a constraint plays (e.g. the diligence policy).

What you really want to know is not the role of closing the window on that one occasion, but the role of the constraint which makes you always close the window in

such situations. Instead of "If I left the window open, then ..." you would say "If I were not diligent, then ..."; we will examine explaining the *policy* that lead to the action, not the *action* itself. In order do do this, we will need to develop a notion of the *role of a constraint.* In fact, the formalism we develop will not only let us ask for the role of a constraint, but also about the role of an arbitrary subformula of a constraint.

2.1.4 What is the Role of a Formula?

What kind of an animal is the role of a formula? I appeal to the intuition of what a human would say to another human to explain such a thing; its role is what it allows or what is disallows, or how it interacts with another part of the model. For now, we will focus on *non-example generation*, which addresses the first view. *Core extraction*, described in Chapter 3, addresses the second.

Note that this approach lends itself to a model-theoretic view of the world rather than a logical view of the world. That is, it looks at enumerating the solutions (within some bound) and looking at how that set changes, rather than using syntactic manipulations to produce a proof.

Naive Role Computation

The naive approach is to just check the `StayDry` assertion once with `Diligent` present and once with it absent. This might get us what we want, if we get either of the following results:

(a) if neither check finds any counterexamples, then we know that `Diligent` is redundant; we stay dry either way.

(b) if the first check passes but the second finds a counterexample, then we have an example of a circumstance where `Diligent` is necessary to ensure `StayDry`. The role of `Diligent` is to eliminate that case (and others like it).

As the model stands, we expect the second result. However, we might not be so fortunate; consider the situation in which the window is broken. In that case, there's

nothing you can do to stay dry if it happens to rain, be you diligent or not. The two possible results of the naive approach are the following:

(a) if both checks return the same counterexample (e.g. where the window is broken), you now know one case where `Diligent` is irrelevant since you will get wet anyway. However, you don't know if there are cases where `Diligent` *does* matter.

(b) if both checks return different counterexamples, you have learned even less. Perhaps each counterexample is actually a solution to both checks, or perhaps not.

What we really want is to solve for a counterexample which is only valid because `Diligent` was not enforced. We want an assignment which is *not* a solution to the model with `Diligent` enforced, but which *is* a solution when it is removed. This can be expressed as a solution to

$$\neg M \wedge (M - D)$$

where M is the model, and $(M - D)$ is the model with the `Diligent` constraint removed. Solutions to this formula are cases which are disallowed by the presence of `Diligent`. We now get the desired solution `<Intact, Open, Raining, Wet>`. We term this approach *non-example generation*, since we are generating assignments which are almost solutions to the model, but not quite. In particular, we generate assignments which would be solutions except that they fail to satisfy the target subformulae.

In the next section, we will formalize this notion of *role* in terms of non-examples, develop a convenient notation (conjunction diagrams), and prove some handy lemmas. Later, we will extend the technique beyond propositional logic.

2.2 Formalization and Representation: Conjunction Diagrams

"In a few minutes a computer can make a mistake
so great that it would have taken many men many months to equal it."

Unknown

In the last section, we concluded that to determine the role of the presence of T in M, we should examine solutions to the formula $M \wedge \neg(M - T)$; such solutions satisfy the original model but not the altered model, thus indicating what has been *disallowed* by including T in the model. First, we need to formalize what it means to delete a subformula from a model. Then we will look at other formulae which produce useful information for the user.

2.2.1 Deletion (and Sabotage) Formalized

We need to be precise when we say "*delete* the subformula T from the model M", denoted $(M - T)$. Here is what it shall mean:

- M is the *simple* AST of the user's model model. A *simple* AST is one in which all logical operations have been desugared into $\neg$, binary $\wedge$, and binary $\vee$.
- T is a node in M, at the root of the target subformula. We assume that T is not the root of M and that T's parent node is not a negation (although T itself may be a negation). To ask about the role of a subformula which is directly negated, one must instead ask the role of the negation plus that subformulae.

A well formed tree results when the subtree rooted at T is removed from the AST M, except that the old parent of T will now be either a unary $\wedge$ or a unary $\vee$. That node is interpreted using the following semantics, making $(M - T)$ a *simple* AST:

$$\texttt{unary and} \ \wedge(x) = x$$
$$\texttt{unary or} \ \vee(x) = x$$

Constant Replacement

It will be convenient to talk about deletion in terms of replacing T with a Boolean constant. To that end, we prove the following result:

Theorem 1:

$(M - T)$ is either equivalent to $M[T \leftarrow \texttt{True}]$ (the formula obtained by replacing T with the constant True), or it is equivalent to $M[T \leftarrow \texttt{False}]$ (the formula obtained by replacing T with the constant False).

Proof:

Since M is *simple* and we have discounted the case were T's parent is a negation ($\neg$), we know that T's parent is either an $\wedge$ or an $\vee$. If T's parent is an $\wedge$, then $(M - T) \equiv M[T \leftarrow \texttt{True}]$. If T's parent is an $\vee$, then $(M - T) \equiv M[T \leftarrow \texttt{False}]$.

This property of deletion leads us to define the complementary notion of *sabotage*, which will prove useful later on when we define a notion of *correctness* (Setion 2.2.4).

Definition:

Let C be the constant for which $(M - T) \equiv M[T \leftarrow C]$. The result of *sabotaging* T within M, denoted $(M \sim T)$, is $M[T \leftarrow \neg C]$.

One way of thinking about sabotage is as follows: Where *deleting* T from M leaves the rest of M as intact as possible, *sabotaging* T in M simplifies the rest of M as much as possible, to the extend that altering T can. For example,

$$\begin{aligned} (A \wedge B) - B &\equiv A \wedge \texttt{True} &&\equiv A \\ (A \wedge B) \sim B &\equiv A \wedge \texttt{False} &&\equiv \texttt{False} \end{aligned}$$

Sabotaging a subformula produces the maximum possible effect that the subformula could have on the entire model.

2.2.2 Conjunction Diagrams

It is now clear exactly what we mean when we write $M \wedge \neg(M - T)$. It will also prove fruitful to consider the other 3 possible combinations of negating and conjoining those two formulae: $\neg M \wedge (M - T)$, $M \wedge (M - T)$, and $\neg M \wedge \neg(M - T)$. A convenient

form for representing these combinations is a *conjunction diagram.*

Definition:

A *conjunction diagram* is a table constructed from two sets of logical formulae: one set is written as labels on the columns of a table and the other as labels on the rows. An entry in the table contains a formula (the conjunction of the row and column formulae) and/or a set of assignments (solutions to that formula).

Table 2.1: A generic conjunction diagram with column formulae Col_1 and Col_2 and row formulae Row_1 and Row_2

$\bigwedge$	Col_1	Col_2
Row_1	$Col_1 \wedge Row_1$	$Col_2 \wedge Row_1$
Row_2	$Col_1 \wedge Row_2$	$Col_2 \wedge Row_2$

In our case, the row formulae will be always be M and $\neg M$. When we draw the conjunction diagram for deletion, the column formulae are $(M - T)$ and $\neg(M - T)$. Later we will add additional column formulae.

Table 2.2: The generic conjunction diagram for deletion

$\bigwedge$	$(M - T)$	$\neg(M - T)$
M	PIA Presence Irrelevant to Allowing $M \wedge (M - T)$	PCA Presence Crucial to Allowing $M \wedge \neg(M - T)$
$\neg M$	PCD Presence Crucial to Disallowing $\neg M \wedge (M - T)$	PID Presence Irrelevant to Disallowing $\neg M \wedge \neg(M - T)$

Conjunction diagrams are both useful for performing proofs and for presenting the information to the user. However, it is probably unreasonable to expect to teach the user the precise semantics of conjunction diagrams just to be able to answer simple questions. To that end, each entry is also given a name which indicates the meaning to the user of the solutions in that cell.

- $M \wedge (M - T)$ produces solutions to the original model, M, which are still solutions if T is deleted. We abbreviate this category of solutions **PIA**, pronounced "*Presence Irrelevant to Allowing*". (*Presence* means "failure to delete".)

- $M \wedge \neg(M - T)$ produces solutions to the original model, M, which cease to be solutions if T is deleted. We abbreviate this category of solutions **PCA**, pronounced "*Presence Crucial to Allowing*".

- $\neg M \wedge (M - T)$ produces assignments which are not solutions of the original model, M, but which become solutions if T is deleted. We abbreviate this category of solutions **PCD**, pronounced "*Presence Crucial to Disallowing*".

- $\neg M \wedge \neg(M - T)$ produces assignments which are not solutions of the original model, M, and are still not solutions if T is deleted. We abbreviate this category of solutions **PID**, pronounced "*Presence Irrelevant to Disallowing*".

Each cell of the conjunction diagram contails *all* solutions satisfying the equation written there. To avoid an overwhelming amount of data, when displaying a conjunction diagram to the user, only one (arbitrarily chosen) solution is shown in each cell. As we will see in Section 2.5, one solution is often enough to learn interesting things about a model.

Observation: Each possible assignment to variables of M appears somewhere in the conjunction diagram for deletion; for any assignment A,

$$A \in PIA \cup PCA \cup PCD \cup PID$$

2.2.3 Representing Sabotage

We construct an analogous conjunction diagram for sabotage by setting the column formulae to $(M \sim T)$ and $\neg(M \sim T)$. The interpretation of solutions in the conjunction diagram for sabotage are similar to that of deletion. However, instead of referring to the role of T's *presence* (failure to delete), they refer to the role of T's *integrity* [2] (failure to sabotage).

Sabotage represents the maximum extent to which the target formula can influence the structure of the model as a whole. This information is far less valuable to the

[2]Read "integrity" as *structural* integrity not *moral* integrity.

Table 2.3: The conjunction diagram for sabotage

$\bigwedge$	$(M \sim T)$	$\neg(M \sim T)$
M	IIA Integrity Irrelevant to Allowing $M \wedge (M \sim T)$	ICA Integrity Crucial to Allowing $M \wedge \neg(M \sim T)$
$\neg M$	ICD Integrity Crucial to Disallowing $\neg M \wedge (M \sim T)$	IID Integrity Irrelevant to Disallowing $\neg M \wedge \neg(M \sim T)$

user than is deletion; its real value comes from combining it with information about deletion to compute a notion of *correctness* (see Section 2.2.4. To that end, it will prove convenient to display the two tables together.

Table 2.4: The conjunction diagram for deletion and sabotage

$\bigwedge$	$(M - T)$	$\neg(M - T)$	$(M \sim T)$	$\neg(M \sim T)$
M	PIA $M \wedge (M - T)$	PCA $M \wedge \neg(M - T)$	IIA $M \wedge (M \sim T)$	ICA $M \wedge \neg(M \sim T)$
$\neg M$	PCD $\neg M \wedge (M - T)$	PID $\neg M \wedge \neg(M - T)$	ICD $\neg M \wedge (M \sim T)$	IID $\neg M \wedge \neg(M \sim T)$

2.2.4 Theorems

> "An expert is a person who has made all the mistakes
> that can be made in a very narrow field."
>
> *Niels Bohr*

One can consider other mutations of T, besides replacing it with `True` or `False`, and construct comparable conjunction diagrams for them [3]. However, it turns out that the 8 formulae represented in the conjunction diagram for deletion and sabotage are sufficient to compute (or bound) the effect of making an arbitrary change to T.

Definition:

Let CIA, pronounced "*correctness irrelevant to allowing*", denote the set of all assignments which are solutions to M and which remain solutions regardless of how

[3]The column formulae will be $M[T \leftarrow T']$ (M with T mutated) and $\neg(M[T \leftarrow T'])$ (the negation of that formula)

T is altered. CIA represents solutions which are independent of any change to T. CCA, pronounced "*correctness crucial to allowing*", denotes the set of assignments which are solutions to M which may be disallowed by changes to T. CID and CCD are defined accordingly.

Theorem 2: [4]

$$CIA = PIA \cap IIA$$
$$CID = PID \cap IID$$
$$CCA = PCA \cup ICA$$
$$CCD = PCD \cup ICD$$

That is, an assignment is only *irrelevant* to correctness if *both* its presence and its integrity are also irrelevant. An assignment is *crucial* to correctness if *either* its presence or its integrity is crucial.

Technically, these 4 entries do not belong in a conjunction diaram, as they are not computed by conjoining row and column formulae. However, in terms of the information they carry, they belong with the 8 cells concerning correctness and integrity. We will follow the convention of tacking them onto the end of our conjunction diagrams, with the understanding that they are actually computed in a different manner.

Table 2.5: The role of the final clause of a CNF formula

$\bigwedge$	*presence* $(M-T)$	$\neg(M-T)$	*integrity* $(M \sim T)$	$\neg(M \sim T)$	*correctness*	
M	PIA $M \wedge (M-T)$	PCA $M \wedge \neg(M-T)$	IIA $M \wedge (M \sim T)$	ICA $M \wedge \neg(M \sim T)$	CIA PIA $\cap$ IIA	CCA PCA $\cup$ ICA
$\neg M$	PCD $\neg M \wedge (M-T)$	PID $\neg M \wedge \neg(M-T)$	ICD $\neg M \wedge (M \sim T)$	IID $\neg M \wedge \neg(M \sim T)$	CCD PCD $\cup$ ICD	CID PID $\cap$ IID

Sometimes the right notation is everything. In our case, conjunction diagrams makes the following result obvious, and the proof trivial:

Observation: Consider a non-empty model, M, and a proper [5] subformula of that model, T. The intersection of any two entries in the conjunction diagram of M for the deletion of T is empty. The same is true of the conjunction diagram of M for the

[4]The proofs for theorems appearing in this section have been relegated to Appendix B.
[5]i.e. $M \neq T$

$\wedge$	presence (M-T)	presence ¬(M-T)	integrity (M~T)	integrity ¬(M~T)
M	***PIA***	***PCA***	***IIA***	***ICA***
¬M	***PCD***	***PID***	***ICD***	***IID***

Figure 2-2: There are only 6 pairs of entries in a conjunction diagram which are not always disjoint.

sabotage of T. These results are intuitive given the labels chosen for each entry, and the proof follows trivially from the definition of a conjunction diagram.

We can also make some guarantees about disjointness *between* the conjunction diagram for deletion and that of sabotage.

Theorem 3:

$$PCA \cap ICA = \emptyset$$

$$PCD \cap ICD = \emptyset$$

The interpretation of this result is that it is impossible for a target formula's presence and its integrity to both be crucial to allowing solutions. If replacing T with `True` would disallow some solution, then replacing T with `False` cannot disallow some other solution. Intuitively, the relaxing nature of T is only undone by one of those constants.

So far, we have proven that almost any pair of entries from the conjunction diagrams for deletion and sabotage are disjoint. In fact, there are only 6 pairs which are not always disjoint: PIA/IIA, PIA/ICA, PCA/IIA, PCD/ICD, PCD/IID, and PID/ICD. These pairs are indicated in Figure 2-2, and are proven by the first two examples in Section 2.3.

We can make a stronger statement about certain pairs of disjoint entries; not only are they disjoint but one of them is always empty.

Theorem 4:

$$(PCA = \emptyset) \vee (PCD = \emptyset)$$
$$(ICA = \emptyset) \vee (ICD = \emptyset)$$

The significance of this result is that any given formula in a propsiional logic model either strictly relaxes the model or strictly restricts it. The set of solutions to $M - T$ is either a subset of a superset of the solutions to M. [6]

2.3 Propositional Logic Examples

2.3.1 Trivial Examples

Consider the role of B in the model $A \vee B$. In this case, B is deleted by setting it to `False` and sabotaged by setting it to `True`.

Table 2.6: The role of B in $A \vee B$

$\bigwedge$	*presence* $(M - T) = A$	 $\neg(M - T) = \neg A$	*integrity* $(M \sim T) =$ True	 $\neg(M \sim T)=$False	*correctness*	
M $A \vee B$	PIA A $(A, \neg B)$ (A, B)	PCA $\neg A \wedge B$ $(\neg A, B)$	IIA $A \vee B$ (A, B) $(\neg A, B)$ $(A, \neg B)$	ICA False $(A, \neg B)$ (A,B)	CIA A $(\neg A, B)$	CCA $\neg A \wedge B$
$\neg M$ $\neg A \wedge \neg B$	PCD False $\emptyset$	PID $\neg A \wedge \neg B$ $(\neg A, \neg B)$	ICD $\neg A \wedge \neg B$ $(\neg A, \neg B)$	IID False $\emptyset$	CCD $\neg A \wedge \neg B$ $(\neg A, \neg B)$	CID $\neg A \wedge \neg B$ $\emptyset$

Let's check these results with our intuition: the role of the presence of the target (B) is to allow the solution $(\neg A, B)$; $(\neg A, B)$ is no longer a satisfying assignment if B is deleted from the model. The target has no effect on (A, B) and $(A, \neg B)$, as they would be allowed anyway. It also has no effect on $(\neg A, \neg B)$, as that assignment would be disallowed anyway. Its presence isn't crucial to disallowing any solutions.

[6] In logic, there is a standard notion of the *polarity* of a term (a variable or constant) in a Boolen formula [13]. A logical formula is converted to NNF (Negation Normal Form) by repeatedly applying DeMorgan's Laws to push all negations down to the terms. Once in NNF, a term's *polarity* is whether or not is it negated. The polarity of a term plus the Boolean operator acting on it determine if the term plays a relaxing or restricting role. Only terms have polarity – arbitrary subformula do not.

Since B is a top level disjunct, sabotaging it makes the model trivially True. The integrity of B is not crucial to allowing any solutions, as any solution to M is trivially a solution to $M \sim T \equiv$ True. Conversely, the integrity of B is crucial to *dis*allowing *every* assignment which is not a solution to M.

Now let's change that formula slightly, and consider the role of B in $A \wedge B$. In this formula, B is deleted by setting it to True and sabotaged by setting it to False.

Table 2.7: The role of B in $A \wedge B$

$\wedge$	*presence*		*integrity*		*correctness*	
	$(M - T) = A$	$\neg(M - T) = \neg A$	$(M \sim T) =$ False	$\neg(M \sim T)=$True		
M $A \wedge B$	PIA $A \wedge B$ (A, B)	PCA False $\emptyset$	IIA False $\emptyset$	ICA A $\wedge$B (A, B)	CIA False $\emptyset$	CCA $A \wedge B$ (A, B)
$\neg M$ $\neg A \vee \neg B$	PCD $A \wedge \neg B$ $(A, \neg B)$	PID $\neg A$ $(\neg A, \neg B)$ $(\neg A, B)$	ICD False $\emptyset$	IID $\neg A \wedge \neg B$ $(\neg A, \neg B)$ $(\neg A, B)$ $(A, \neg B)$	CCD $A \wedge \neg B$ $(A, \neg B)$	CID $\neg A$ $(\neg A, \neg B)$ $(\neg A, B)$

Let's check these results with our intuition: the role of the presence of the target (B) is to disallow the solution $(A, \neg B)$. It has no effect on $(\neg A, \neg B)$ and $(\neg A, B)$, as they would be allowed anyway. It also has no effect on (A, B), as that assignment would be allowed anyway. Its presence isn't crucial to allowing any solutions.

Since B is a top level conjunct, sabotaging it makes the model trivially False. The integrity of B is not crucial to disallowing any solutions, as any assignment not satisfying M trivially does not satisfy $M \sim T$ (since it simplifies to False). Conversely, the integrity of B is crucial to allowing *every* solution of M.

Number of Solutions per Cell

In later examples, we will only list *one* solution in each cell of the conjunction diagram, rather than all of them, as the number of variables (and thus total number of assignments) will be much larger. In general, when a conjunction diagram is shown to a human it is necessary to suppress all but one solution in each entry.

2.3.2 CNF Example

Now let's take a look at a model in CNF form. C_2 denotes the second clause, and C_24 denotes the fourth term of the second clause.

$$\begin{aligned} M &= C_1 \wedge C_2 \wedge C_3 \wedge \cdots \wedge C_{length(M)} \\ C_1 &= T_{11} \vee T_{12} \vee T_{13} \vee \cdots \vee T_{1\ length(C_1)} \\ C_2 &= T_{21} \vee T_{22} \vee T_{23} \vee \cdots \vee T_{2\ length(C_2)} \\ \ldots \end{aligned}$$

From our intuitive understanding of CNF, we expect to see that the role of a clause is to restrain the model, and that the role of a term is to relax the model. That is, the role of a clause C_K is given by a conjunction diagram of the form depicted in Table 2.8. In that table, "..." represents a cell we expect to containin one or more solutions.

Table 2.8: Our intuition for the role of a CNF clause

$\wedge$	*presence* $(M - T)$	$\neg(M - T)$	*integrity* $(M \sim T)$	$\neg(M \sim T)$	*correctness*	
M	PIA ...	PCA ...	IIA $\emptyset$	ICA ...	CIA $\emptyset$	CCA ...
$\neg M$	PCD ...	PID ...	ICD $\emptyset$	IID ...	CCD ...	CID ...

The reasoning behind this expectation is as follows: As we saw in Theorem 4 (Section 2.2.4), each constraint plays either a relaxing or a restraining role in a model. Since T is in a top-level conjunct, it is eliminating solutions to the model, and thus PCD is non-empty and PCA is empty. In general, $PCD \neq \emptyset$ indicates that the target formula plays a restricting role in the model. Sabotaging T is equivalent to setting it to `False`, and thus $(M \sim T) =$ `False`. Thus IIA must be empty, since T's integrity is crucial to allowing every solution to M. It follows from Theorem 2 that CIA is empty.

We would expect the role of a term in that clause, T_{KL}, to be relaxing and thus correspond to a conjunction diagram of the form depicted in Table 2.9.

Table 2.9: Our intuition for the role of a term in a CNF clause

$\wedge$	*presence* $(M-T)$	$\neg(M-T)$	*integrity* $(M \sim T)$	$\neg(M \sim T)$	*correctness*	
M	PIA ...	PCA ...	IIA ...	ICA $\emptyset$	CIA ...	CCA ...
$\neg M$	PCD $\emptyset$	PID ...	ICD ...	IID ...	CCD ...	CID ...

In a CNF, we expect a term to play a relaxing role. Therefore PCA ought to contain solutions while PCD is empty (Theorem 4 from Section 2.2.4). In general, $PCA \neq \emptyset$ indicates that the target formula plays a relaxing role in the model. Sabotaging a CNF term is equivalent to setting it to True, effectively deleting that clause containing that term. Thus the integrity portion of this diagram will look like the presence portion of the previous one. We do not have enough information to draw any conclusions about the emptiness of the correctness cells.

Indeed our intuition for those two diagrams may well be correct. However, it is also possible to get a very different result. What would it mean for the role of the CNF clause C_K to be given by Table 2.10?

Table 2.10: The role of a redundant CNF clause

$\wedge$	*presence* $(M-T)$	$\neg(M-T)$	*integrity* $(M \sim T)$	$\neg(M \sim T)$	*correctness*	
M	PIA ...	PCA $\emptyset$	IIA $\emptyset$	ICA ...	CIA $\emptyset$	CCA ...
$\neg M$	PCD $\emptyset$	PID ...	ICD $\emptyset$	IID ...	CCD $\emptyset$	CID ...

Since both PCA and PCD are empty, we know that the target clause is redundant! The target clause is implied by the other clauses, and can be deleted without effect; $(M - T) \Rightarrow M$.

However, assuming M is satisfiable, sabotaging the target clause still has an effect, and ICA will be non-empty. Theorems 2 and 4 tells us that $ICD = \emptyset$ and therefore $CCD = \emptyset$. However, by the same theorems, CCA is non-empty; there exist clauses we could write in place of C_K which would *not* be redundant. While the target clause is redundant, the context in which is appears is not.

Let's look at what we would have seen if we had instead asked for the role of a *term* inside of that redundant clause. Table 2.11 shows the conjunction diagram we would compute.

Table 2.11: The role of a term in a redundant CNF clause

$\bigwedge$	*presence* $(M - T)$	$\neg(M - T)$	*integrity* $(M \sim T)$	$\neg(M \sim T)$	*correctness*	
M	PIA ...	PCA $\emptyset$	IIA ...	ICA $\emptyset$	CIA ...	CCA $\emptyset$
$\neg M$	PCD $\emptyset$	PID ...	ICD $\emptyset$	IID ...	CCD $\emptyset$	CID ...

As before, both PCA and PCD are empty indicating that the target term's presence is irrelevant. Recall our previous observation that sabotaging a term effectively deletes the clause containing it. In this case, the containing clause's presence is assumed to be irrelevant, thus a term in it will have empty ICA and ICD entries. By Theorem 2, CCD and CCA are also empty. In general, $CCD \cup CCA = \emptyset$ means that the context in which the target occurs is totally harmless; nothing we write in its place will have any effect on M.

2.3.3 Using *Expansion* to Explore a Model

Suppose, in the course of exploring our model, we come across a subformula for which $CCD \cup CCA = \emptyset$. As with the CNF, we would expand the target subformula to contain the previous target and some additional constraints. By incrementally expanding the target subformula, we will eventually settle on a target for which $CCD \cup CCA \neq \emptyset$ but either $PCD \cup PCA \neq \emptyset$ or $ICD \cup ICA \neq \emptyset$ [7]. We can then check that result against our intuition for the model and correct it if necessary. We discuss a complementary approach, target *refinement*, in Section 2.6.

[7]Which pair is non-empty depends on whether sabotaging the original target serves to delete or sabotage the enclosing target.

2.3.4 Making Dinner

I am considering making dinner for my wife, who is coming home from work. If it's dinner time, and she's not running late, then I'll make dinner (unless I'm hungry). However, if I'm hungry, I make dinner anyway. I only get hungry at dinner time.

The standard logical encoding of that situation is as follows:

$$(((DinnerTime \land \neg WifeLate) \Rightarrow MakeDinner) \lor Hungry)$$
$$\land(Hungry \Rightarrow MakeDinner)$$
$$\land(\neg DinnerTime \Rightarrow \neg Hungry)$$

Let's ask for the role of $\neg WifeLate$ in the first conjunct. That is, what if I don't pay attention to whether or not she is running late? Solutions in the table below are given in the form $(WifeLate, DinnerTime, MakeDinner, Hungry)$; the entry $(1, 1, 0, 0)$ means that $WifeLate$ and $DinnerTime$ are `True`, while $MakeDinner$ and $Hungry$ are `False`.

Table 2.12: The role of $\neg WifeLate$ in the dinner example

$\bigwedge$	*presence* $(M - T)$	$\neg(M - T)$	*integrity* $(M \sim T)$	$\neg(M \sim T)$	*correctness*	
M	PIA $(1,1,1,1)$	PCA $(1,1,0,0)$	IIA $(1,1,1,1)$	ICA $\emptyset$	CIA $(1,1,1,1)$	CCA $(1,1,0,0)$
$\neg M$	PCD $\emptyset$	PID $\emptyset$	ICD $(0,1,0,0)$	IID $(1,0,1,1)$	CCD $(0,1,0,0)$	CID $\emptyset$

From Table 2.12 we can see that $\neg WifeLate$ plays a relaxing role in the model. In particular, its presence makes it possible for me to not make dinner even though it is dinner time, in the case where she is running late and I am not hungry. Hence PCA contains the solution

$$(WifeLate, DinnerTime, \neg MakeDinner, \neg Hungry)$$

representing the case where she is running late at dinner tine and I'm not hungry, so I don't make dinner

The solution ICD contains is

$$(\neg WifeLate, DinnerTime, \neg MakeDinner, \neg Hungry)$$

and represents the situation in which she is coming home on time for dinner, I am not hungry, and yet for some reason I do not make dinner. Since I always make dinner when I am hungry, this situation cannot occur, even if we delete the target constraint.

We have also learned something about the context in which $\neg WifeLate$ appears. Since ICD is non-empty, we know that it is possible to write a constraint in place of $\neg WifeLate$ which would disallow the case in which she is running late, it's dinner time, and I'm hungry and yet I don't make dinner. In particular, the constraint `False` achieves that end.

By looking at IID, we see that nothing we wrote in that context would ever disallow the solution

$$(\neg WifeLate, DinnerTime, \neg MakeDinner, Hungry)$$

If it's dinner time and I'm hungry, no constraint in that context will stop me from making dinner.

In our earlier examples, the role of a subformula could be determined by what operator was acting on it; conjunctions restricted the model and disjunctions relaxed it. However, in this case our target is in a *conjunction*, but its role is to *relax* the model. The structure of the surrounding model matters!

2.4 Handling Rich Logics

"If I had my life to live again,
I'd make the same mistakes, only sooner."

Tallulah Bankhead

2.4.1 Non-Boolean Values

Many logics support non-Boolean values in addition to Boolean values. In such a case, we can still ask about the role of a *Boolean* formula but not about a *non-Boolean* one. This means that we can support relational logics (such as the one Alloy used) as long as we do not ask for the role of a relational expression.

Handling First Order Logic

What if our logic is *first-order*? Will this complicate non-example generation? It turns out not to, although coming to that conclusion requires understanding how quantifiers are handled by declarative modeling languages.

In order to analyze a first-order model for counterexamples, it is first necessary to eliminate the quantifiers. Since we are operating over a finite universe, quantifiers can be rewritten by enumerating all possible values of the quantified variable. [8]

Once the formula has been grounded out, the body has been duplicated many times (one per element of the quantifier domain). At first glance, that duplication makes us worry that we will need to develop some sort of notion of the 'joint role' of several target subformulae. We might worry that each call site will need to be treated differently: What if deletion is equivalent to replacement by `True` in one context but equivalent to replacement by `False` in another? Furthermore, in order to compute the role of the correctness of all those subformula, we would need to consider the effect of deleting one copy while sabotaging another and leaving a third intact – considering all such combinations would be an exponential computation!

Fortunately, we can avoid the introduction of such a notion. Each grounded out copy of the quantifier body is in exactly the same context. If the target T constitutes the entire quantifier body, grounding out will produce a bunch of conjoined or disjoined copies of T (depending on the quantifier being grounded out) – all those copies will be in the same context and will thus be deleted and sabotaged in identical

[8] For example, in the finite universe where $X = x_1, x_2, x_3$, the quantified formula $\forall x \in X | f(x)$ is equivalent to the ground formula $f(x_1) \wedge f(x_2) \wedge f(x_3)$. Similarly, we can rewrite $\exists x \in X | f(x)$ as the ground formula $f(x_1) \vee f(x_2) \vee f(x_3)$.

manners. Similarly, the structure of each ground copy is the same – only variable values change not the logical structure. Thus, if T is a proper (i.e. it does not include the entire body) subformula of the quantifier body, each grounded copy will appear in the same context. The need to consider all combinations of deleting and sabotaging the different copies T is also averted. Since the local contexts of the copies of T are identical, it would be impossible to change T in a way that would delete one copy but sabotage another.

The consequence of all this is that a target subformula appearing in a quantifier body can be handled just like an ordinary propositional formula. Non-example generation can be applied as usual.

2.4.2 Defined Names and Other Syntactic Sugars

A definition is syntactic sugar which allows a token (the name) to be bound to a formula (the body). Any instance of the name (a call site) is shorthand for the corresponding body. The definition can be parameterized, in which case the body that replaces the name depends on the parameter. For example, the following model contains one definition with two uses:

$$N(x) := (B \Rightarrow x)$$
$$N(A) \wedge (N(C) \vee D)$$

It is desugared into the following model:

$$(B \Rightarrow A) \wedge ((B \Rightarrow C) \vee D)$$

First of all, observe that if the target contains a function *call site*, then it can be handled as usual. The call site can be deleted and sabotaged by replacing it with the appropriate Boolean constants.

After desugaring, there is no trace left of the definitions, so we can proceed with non-example generation as usual. However, there is a catch: if the target subformula T is in the *body* of a definition, then there are now multiple copies of it.

This is the same problem we faced with first-order logic, but it will not be so easy to avoid this time. In the case of grounded out quantifiers, each copy of T appeared in an identical context. Two copies of an in-lined definitions may end up in very different contexts! It turns out that we can avoid the introduction of a 'joint role' computation in the cases that we care about, and simply forbid the uninteresting (and difficult) cases. A proper formalization of joint role is possible, but fortunately it is unnecessary.

The body of each in-lined copy of the definition is the same except for the values of variables passed in. If T is proper (i.e. it does not contitute the entire body), the situation is no different than that of a proper subformula of a quantifier body. Each copy of T will appear in the same context, and non-example generation can proceed as usual.

On the other hand, if T constitutes the entire definition body, then the different copies of T may appear in different contexts. Before leaping into a formalism to address that concern, let us consider what the user is asking by marking an entire definition body as a target. The user is saying "what would happen if *every* call to this definition were deleted?". This is a bizarre question to ask. The user would more likely ask "what would happen if *this* call site were deleted?", which can be handled without complication.[9] For simplicity, we forbid the user from selecting an entire definition body as a target. This restriction slightly reduces the expressiveness of the questions users can ask, but it dramatically reduces the complexity (both conceptual and computation) of the algorithm.

Special Case: Top Level Conjunct

There is a fairly common special case in which we can operate on the original model without desugaring declarations and other special constructs. Consider the case where the user has marked an entire top-level fact; that is, a formula which is a conjunct in

[9]The user might also ask "What if this particular part of the definition body were deleted, but only from this particular call site?". Since we only allow users to mark text on the original (non-desugared) model, we do not support such questions, although the technique could be extended to support them.

the top-level conjunction of the model [10]. In such a case, let R denote the rest of the model, so that $M = R \wedge T$ and $(M - T) = R$. We solve for PIA, PCA, and PCD by using the following derivations, without the need to negate M (or R).

PIA can be computed by any of the following expressions:

$$\begin{aligned}
&M \wedge (M - T) \\
&= (R \wedge T) \wedge R \\
&= R \wedge T \\
&= M
\end{aligned}$$

PCA can be computed by any of the following expressions:

$$\begin{aligned}
&M \wedge \neg(M - T) \\
&= (R \wedge T) \wedge \neg R \\
&= \emptyset
\end{aligned}$$

PCD can be computed by any of the following expressions:

$$\begin{aligned}
&= \neg M \wedge (M - T) \\
&= \neg(R \wedge T) \wedge R \\
&= (\neg R \vee \neg T) \wedge R \\
&= (\neg R \wedge R) \vee (\neg T \wedge R) \\
&= \texttt{FALSE} \vee (\neg T \wedge R) \\
&= \neg T \wedge R
\end{aligned}$$

In summary, if T is a top level conjunct of M, then

$$\begin{aligned}
&PIA = M \\
&PCA = \emptyset \\
&PCD = M[T \leftarrow \neg T]
\end{aligned}$$

[10] It is safe to assume that the top node of any model's AST is a conjunction, since at the very least one must conjoin the model description with the negated property: $M \wedge \neg p$

There is no comparable derivation for PID, but fortunately it is the least useful entry in a conjunction diagram.

What About Sabotage? In the case of a top level conjunction, sabotage (and therefore correctness) are not interesting, since sabotaging T makes M trivially false:

$$(M \sim T) = (M - T) \wedge \texttt{False} = \texttt{False}$$

2.5 Alloy Examples

"Mistakes are the usual bridge
between inexperience and wisdom."
Phyllis Therous

In this section, three Alloy models are discussed in depth. The first model (Section 2.5.1) is a toy example about Paul Simon's song "One Man's Ceiling is Another Man's Floor". It serves to show how conjunction diagrams can explain parts of an Alloy model, although since the model is so simple non-examples will not produce any explanations which could not have easily been generated by hand.

The second model (Section 2.5.2) is distributed with the Alloy language as a library module; it supplies a notion of a sequence (plus helper functions) which may be imported into other models. The author of the sequence module reported difficulties in understanding parts of the sequence module. The process he used to alleviate that confusion is essentially a non-automatic and less precise version of non-example generation.

The third model (Section 2.5.3) describes part of the *Firewire* network protocol. It is a large model, but non-example generation can still help to understand the role of small parts of it. This model demonstrates the value of understanding the role of *correctness* and well as the role of *presence* of the target constraint.

Figure 2-3: An Alloy model written by Daniel Jackson about Paul Simon's song "One Man's Ceiling Is Another Man's Floor".

```
module models/examples/toys/CeilingsAndFloors

sig Platform {}
sig Man {ceiling, floor: Platform}
fact PaulSimon {all m: Man | some n: Man | Above (n,m)}
pred Above(m, n: Man) {m.floor = n.ceiling}
assert BelowToo {all m: Man | some n: Man | Above (m,n)}
check BelowToo for 2 expect 1

pred Geometry (){no m: Man | m.floor = m.ceiling}
assert BelowToo' {
  Geometry() => all m: Man | some n: Man | Above (m,n)}
check BelowToo' for 2 expect 0
check BelowToo' for 3 expect 1

pred NoSharing() {
  no disj m,n: Man | m.floor = n.floor || m.ceiling = n.ceiling}
assert BelowToo'' {
  NoSharing() => all m: Man | some n: Man | Above (m,n)}
check BelowToo'' for 6 expect 0
```

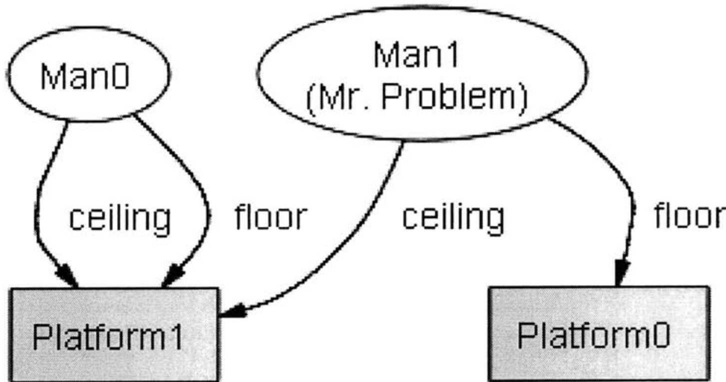
2.5.1 Ceilings and Floors

We will first examine a toy model to illustrate the basic structure of an Alloy model. This model is perhaps the smallest example that uses all kinds of paragraph, and will serve as a segue from toy logic examples to Alloy models written about real applications.

In his 1973 song, Paul Simon said "One Man's Ceiling Is Another Man's Floor.". Under what conditions does it follow that "One Man's Floor Is Another Man's Ceiling."? An Alloy model asking this question is shown in Figure 2-3.

The first two lines create two signatures (sets): `Platform` and `Man`. They also specify two relations, relating each `Man` to a "`ceiling`" `Platform` and a "`floor`" `Platform`. The `fact` enforces Paul Simon's statement that each man's ceiling is some man's floor. Next, a predicate is declared to define one man to be `Above` another

Figure 2-4: A counterexample to the claim that one man's floor is another man's ceiling.



man when the first man's ceiling is the second man's floor. We then `assert` that every man's floor is some man's ceiling and `check` that claim (bounding the size of the universe to include up to 2 `Man` and up to 2 `Platform` objects).

Alloy finds the counterexample shown in Figure 2-4. `Man0`'s ceiling is `man1`'s floor and `man0`'s ceiling is `man1`'s floor, so the `PaulSimon` constraint is satisfied. However, `Man1`'s floor is nobody's ceiling, so the assertion fails.

Let's check our intuition about the role of the `PaulSimon` constraint:

```
fact PaulSimon {all m: Man | some n: Man | Above (n,m)}
```

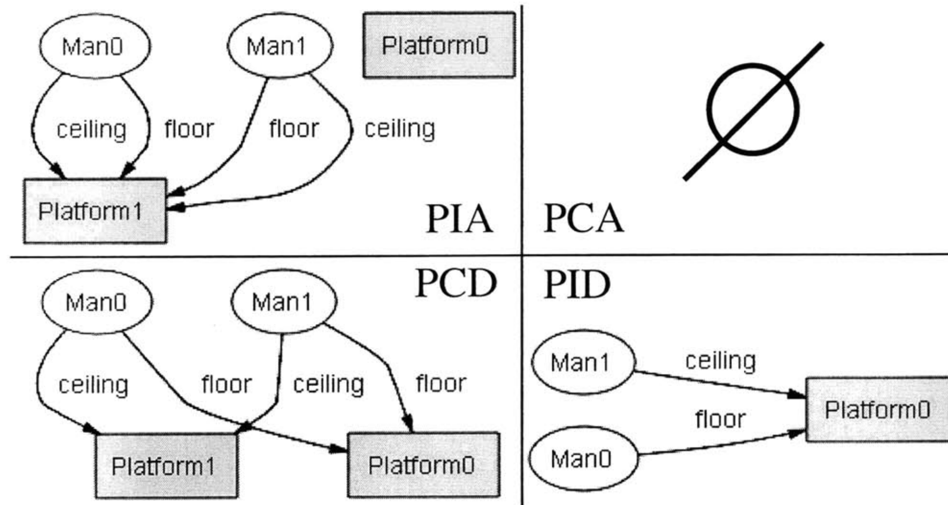
The conjunction diagram for this constraint's presence is shown in Figure 2-5.

We see right off the bat that `PaulSimon` plays a restraining role, since PCA is empty.

Next we look at PCD and see a situation which would be allowed only if `PaulSimon` were omitted: Both men share the same ceiling and the same floor, but floors and ceilings are disjoint – this case violates `PaulSimon` but satisfies the rest of the model. PCD is exactly the sort of situation that `PaulSimon` is suppose to eliminate, so our previous understanding of the model is re-enforced.

PIA shows a situation which is a solution whether or not `PaulSimon` is present: a case in which all ceilings and floors are the same platform. This case is a bit odd, so maybe we want to add another constraint to eliminate it. Indeed, the `Geometry`

Figure 2-5: The role of the `PaulSimon` constraint



constraint discussed below prevents just this sort of situation.

PID shows us a constraint which is so bizarre that it will be excluded whether or not `PaulSimon` is included. This entry is not very informative, but it does re-enforce the belief that we correctly understand the model.

Geometry

Let's consider adding the `Geometry` constraint to remove some of the weirdness we saw previously in the *PIA* entry. `Geometry` ensures that no man's floor is his own ceiling. Alloy shows us that this requirement is still not enough to force every man's floor to be some man's ceiling (although the smallest counterexample now has a scope of 3 instead of 2).

In particular, analysis produces the counterexample shown in Figure 2-6: `Man0` and `Man1` for, a cycle of ceilings and floors. `Man2`'s ceiling is shared with `Man0`, but his floor is nobody's `ceiling`. This counterexample also appears in the *PIA* entry of the conjunction diagram for deletion.

While the `Geometry` constraint is not sufficient to prove our assertion, it still has some effect on the model. We can pick apart the nature of that effect by examining

Figure 2-6: A counterexample to the claim that the `Geometry` constraint is sufficient to force each man's floor to be some man's ceiling

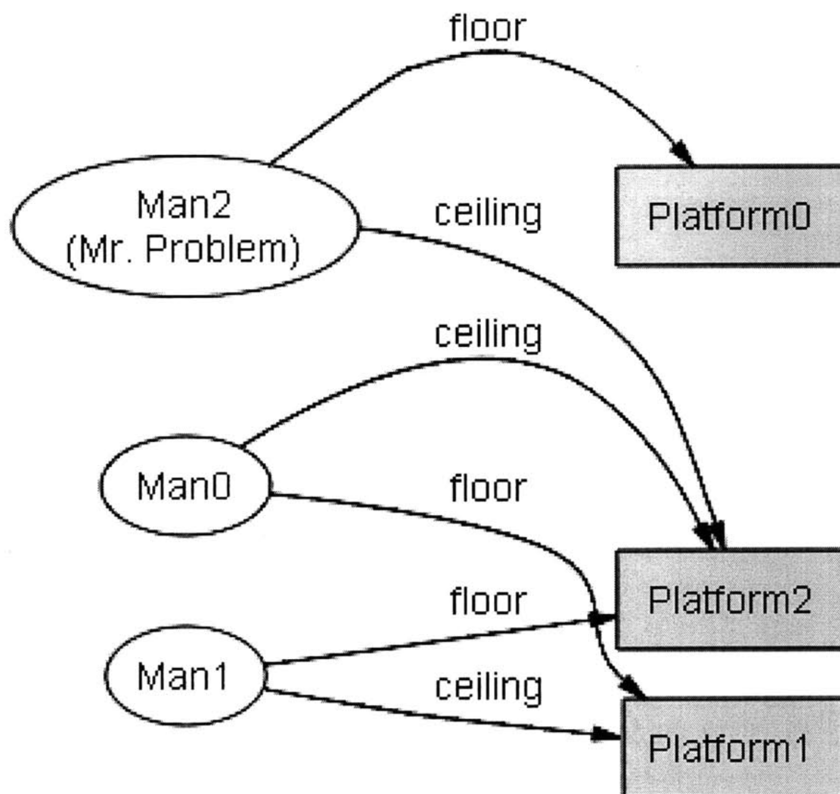
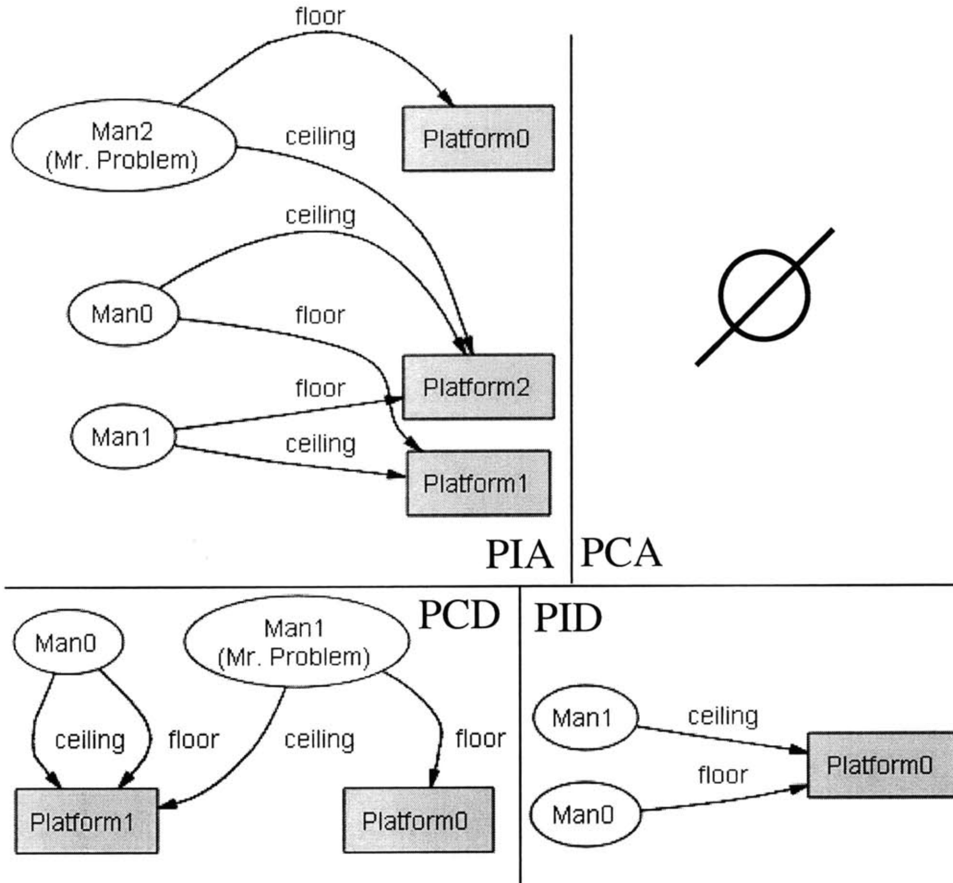


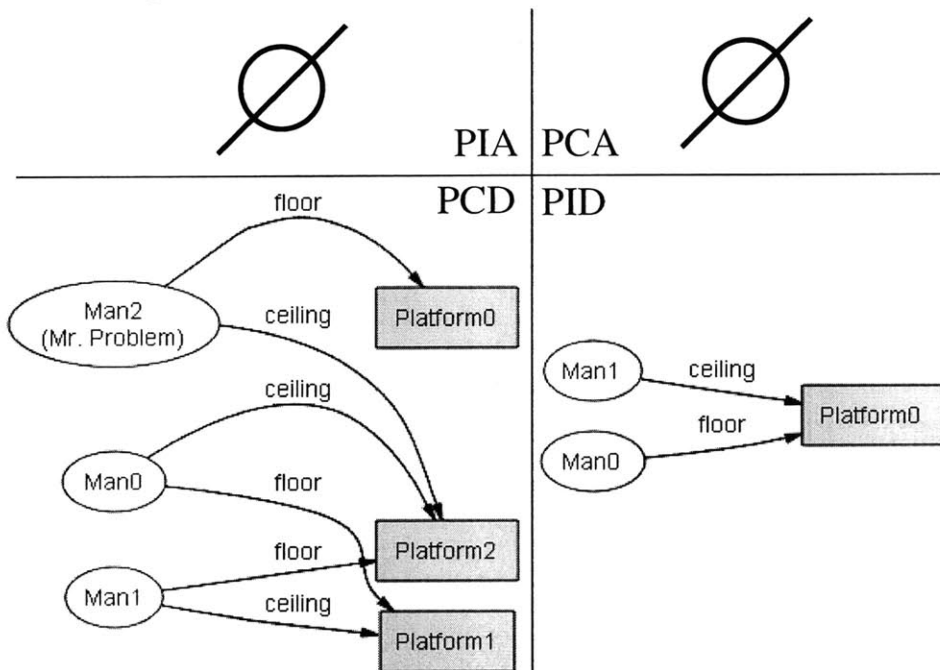
Figure 2-7: The role of the Geometry constraint



the conjunction diagram shown in Figure 2-7.

Since PCA is empty, we confirm our belief that Geometry restrains the model, rather than relaxing it. In particular, PCD shows a solution it is responsible for disallowing – the case where some man's ceiling is his own floor. PIA shows one of the solutions which is allowed whether or not we enforce Geometry, and we can look at it and decide it it ought to be allowed. PID is the least interesting entry, as it contains a bizarre situation which can't occur for many reasons, and would be disallowed regardless of the presence of Geometry.

Figure 2-8: The role of the presence of the NoSharing constraint



NoSharing

The NoSharing constraint further constrains the model so that no two men can share both the same ceiling and the same floor. We assert that NoSharing implies that every man's ceiling is some man's floor, check it for a scope of 6, and find that there are no counterexamples. The role of its presence is given in Figure 2-8.

Both PIA and PCA are empty PCA is empty, indicating that NoSharing does not play a relaxing role. In fact, since PIA is also empty, we know that the unmodified model is unsatisfiable – i.e. the assertion passed. PCD gives us an example of a solution eliminated by NoSharing. As it turns out, the solution shown for PCD is the very counterexample we saw earlier, when NoSharing was not enforced. PID is the same solution we keep seeing there – a solution that violates so many constraints that no one fact is responsible for eliminating it.

2.5.2 Sequence Library

Let's look at an example where the modeler actually got confused about the role of a particular constraint. The developer of this model (Gregory Dennis) reports that every time he looks at it, there is a particular constraint which he has to re-examine and convince himself is necessary. He does so by running the model many times with that constraint removed, and reasoning through each case (by hand) to determine if it is only a solution because the constraint was removed. Non-example generation automates that process.

The Model

Figure 2-9 shows a simplified version of the full model. Like a well written but complex program, this model is easy to use but difficult to understand or modify except perhaps by the original author.

The model encodes a sequence as a mapping from indices to elements. Thus the sequence (A, B, C) would be represented as the relation $(0 \rightarrow A), (1 \rightarrow B), (2 \rightarrow C)$. For simplicity, we have disallowed duplicate elements. For the purposes of easy visualization, we have added a `next` relation in the visualizer that points from each element in a sequence to the next element, if any.

Constraints are added to ensure that the indices begin at 0 and are consecutive thereafter.

The Author's Confusion

The author's confusion arose around the need for the line

```
#added::inds() = #this::inds() + 1
```

in the definition of the add predicate. It states that the size of the last index value of the list after adding an element is one larger than it was before adding the new element. The confusion here is not about whether the constraint should *hold*, but rather it is about whether the constraint should be *enforced*. Thus it is a question

Figure 2-9: A simplified version of Alloy's sequence module

```
module util/sequence2[elem]
open util/ordering[SeqIdx] as ord

sig SeqIdx {}
sig Seq {seqElems: SeqIdx -> lone elem}{
  // Ensure that elems covers a prefix of SeqIdx,
  // equal to the length of the signature
  all i: SeqIdx - ord/first() | some i.seqElems => some ord/prev(i).seqElems
}

// no two sequences are identical
fact canonicalizeSeqs {no disj s1, s2: Seq | s1.seqElems = s2.seqElems}

// helper functions
pred Seq::startsWith (prefix: Seq) {all i: prefix::inds() | this::at(i) = prefix::at
fun next (i: SeqIdx): lone SeqIdx { ord/next(i) }
fun Seq::at (i: SeqIdx): lone elem { i.(this.seqElems) }
fun Seq::elems (): set elem { SeqIdx.(this.seqElems) }
fun Seq::inds (): set SeqIdx { elem.~(this.seqElems) }
fun Seq::lastIdx (): lone SeqIdx { ord/max(this::inds()) }
// returns the index after the last index
// if this sequence is empty, returns the first index,
// if this sequence is full, returns empty set
fun Seq::afterLastIdx () : lone SeqIdx {ord/min(SeqIdx - this::inds())}

// adds an element to the end of a sequence
pred Seq::add (e: elem, added: Seq) {
  //added is the result of appending e to the end of s
  added::startsWith(this)
  added.seqElems[this::afterLastIdx()] = e
  #added::inds() = #this::inds() + 1            //TARGET
}
```

about the role of the constraint's presence, and we construct the conjunction diagram shown in Figure 2-10.

To put it another way, is the target implied by the rest of the model?

The other constraints in the `add` function ensure that

1. the old list is a prefix of the new list, and

2. the next element is in the new list in a position past that prefix.

However, from the *PCD* entry of the conjunction diagram (Figure 2-10), we see the following situation: Sequence `seq1` contains `widget3` followed by `widget1`. `widget0` is added to that list, creating the new sequence `seq0`. However, `widget2` has also been (extraneously) added to the end of that sequence. Apparently the target constraint is necessary to prevent additional stray elements from being added to the end of the new list.

Observations If we had thought of this particular problem beforehand, we could have written an assertion that it never happens. Checking it with Alloy would return us the same counterexample as we saw in the conjunction diagram. However, that requires the user to have already thought of that potential problem, rather than allowing the user to explore the meaning of the constraint.

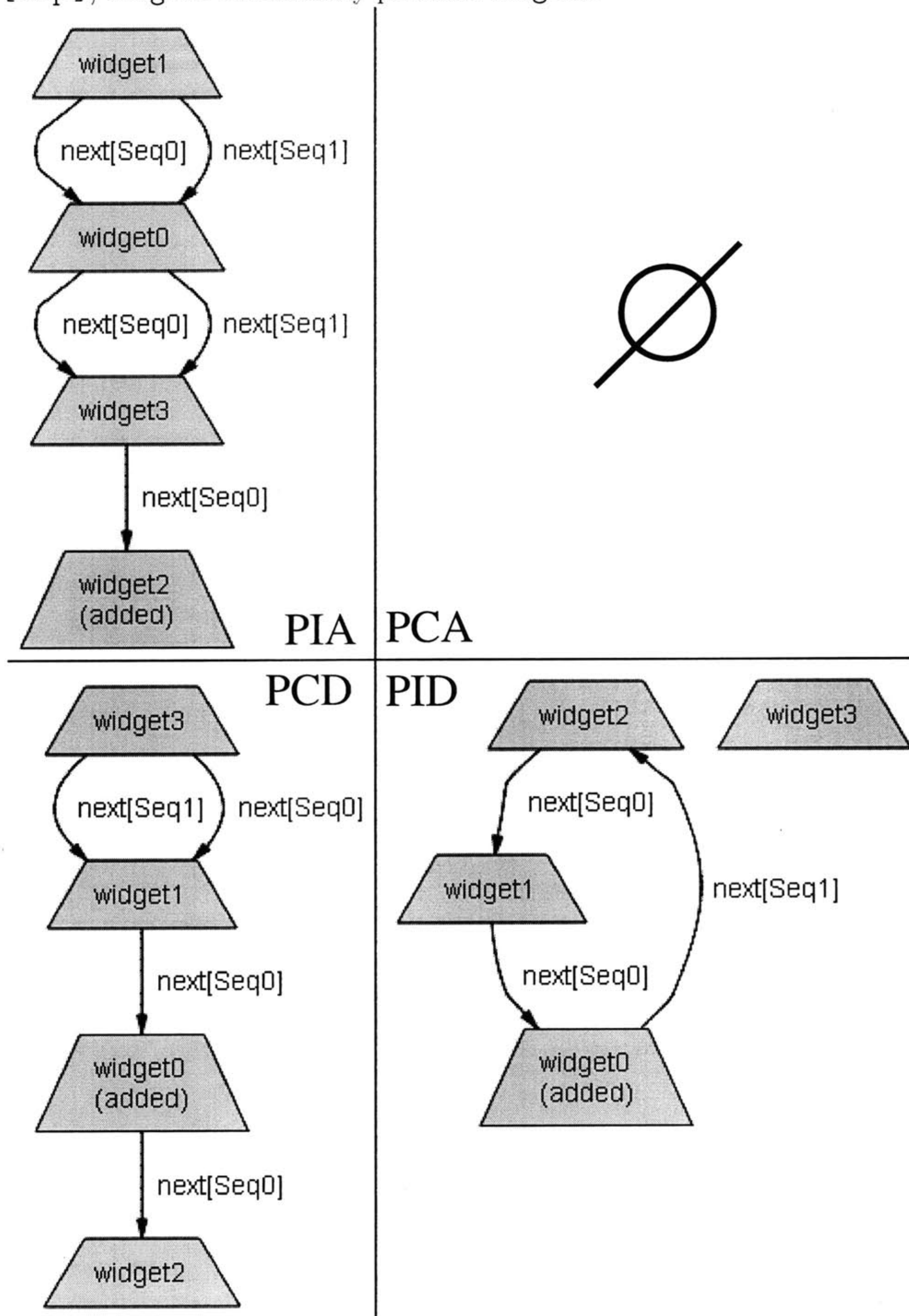
2.5.3 The Firewire Network Protocol

"You must learn from the mistakes of others.
You can't possibly live long enough to make them all yourself."

Sam Levenson

A user can gain wisdom about a model by viewing automatically generated mistakes. You might suppose that you are quite capable of generating your own mistakes and are in no need of more, automatically generated or otherwise. We will argue that understanding the effect of mistakes you *could have* made writing a constraint provides valuable insight into the role of that constraint. In the Firewire model presented in

Figure 2-10: The role of part of the add predicate in the sequence module. An arc labeled next[seq0] from node widget1 to node widget0 means that, in the sequence [seq0], widget1 immediately preceeds widget0.



this section, we will see how non-examples help us understand not only why certain parts of the model were written, but why they weren't written *differently.*

An Overview of the Firewire Model

The Firewire protocol is employed as an initial phase for distributed algorithms. It describes a method for a network of nodes to agree upon a tree-representation of their network. The network is assumed to consist of a collection of nodes connected by a pair of directed links, one in each direction. Viewing a link and its dual as a single, undirected edge, the network as a whole is assumed to form a tree. At the conclusion of the algorithm, the nodes should agree upon which node is the root, and which links point down and which point up. In non-trivial networks, there will be many valid tree representations, and the algorithm is only guaranteed to find one of them.

An Alloy model of this algorithm is given in Appendix A, key sections of which are explained below. The details of the Alloy language are not necessary to understand the model, and all relevant syntax will be explained as we go[11].

The network is modeled as a set of nodes connected by directed links.

```
sig Node {to, from: set Link} {
  to = {x: Link | x.target = this}
  from = {x: Link | x.source = this}
  }

sig Link {target, source: Node, reverse: Link} {
  reverse.@source = target
  reverse.@target = source
  }
```

Each link has `source` and `target` nodes. Each node knows the set of links pointing `to` and `from` it. The second set of curly braces after the `Node` signature define a fact

[11]Interested readers can learn the details of the Alloy language (and download it) from `http://alloy.mit.edu`.

which makes sure that the to and from fields of each node are consistent with the source and target fields of each link.

Each State records facts about the world which change as the algorithm proceeds.

```
sig State {
  part waiting, active, contending, elected: set Node,
  parentLinks: set Link,
  queue: Link -> one Queue,
  op: Op, -- the operation that produced the state
  }
```

At any given point in the algorithm, each node has exactly one status (waiting, active, contending, or elected), each link has a queue of messages (queue, and some operation has just occurred (op).

We define a set of operations which can occur as the algorithm proceeds. We then state that one of these operations occurs at each state transition, and describe the effects of that operation. Remember that since this is a declarative model, we describe how to recognize that a given operation has occurred, rather than describing how that operation actually behaved. Only two of the operation descriptiosn are shown here – the rest can be found in the full model, given in Appendix A.

```
sig Op {}
one sig Init, AssignParent, ReadReqOrAck, Elect, WriteReqOrAck,
    ResolveContention, Stutter extends Op {}

pred Trans (s, s': State) {
  s'.op = Stutter => SameState (s, s')
  s'.op = Elect => {
    s'.parentLinks = s.parentLinks
    some n: Node {
      n in s.active
      n in s'.elected
```

```
      NoChangeExceptAt (s, s', n)
      n.to in s.parentLinks
      QueuesUnchanged (s, s', Link)
      }}
  ... other operations ...
}
```

Understanding the details of each operation would be quite overwhelming, so instead we generate a sample run of the algorithm using the Alloy **run** command. Since Alloy operates over a finite domain, we provide it with a set of bounds.

```
run ElectionHappens for 1 Int, 7 Op, 2 Msg,
  exactly 3 Node,  6 Link, 3 Queue, 7 State
```

The ElectionHappens predicate mentioned in the command just states that initially no node is elected as the root by eventually one is selected. Executing this statement tells Alloy to analyzes the model and generate a solution – in our case solutions take the form of traces of the algorithm over several states. One such trace is shown in Figures 2-11 through 2-14.

While a sample run gives us some intuition for how the algorithm proceeds, it does not offer much insight into the *role* each operation plays in the correctness of the algorithm as a whole. We can see an operation's effect on one particular trace, but not why it is necessary in general or why it should be written the way it is. To answer those more general questions, we turn to non-example generation.

Applying Non-Example Generation

Let's examine what happens if we delete the entire constraint concerning one of the operations. Let the target subformula be

```
  s'.op = Stutter => SameState (s, s')
```

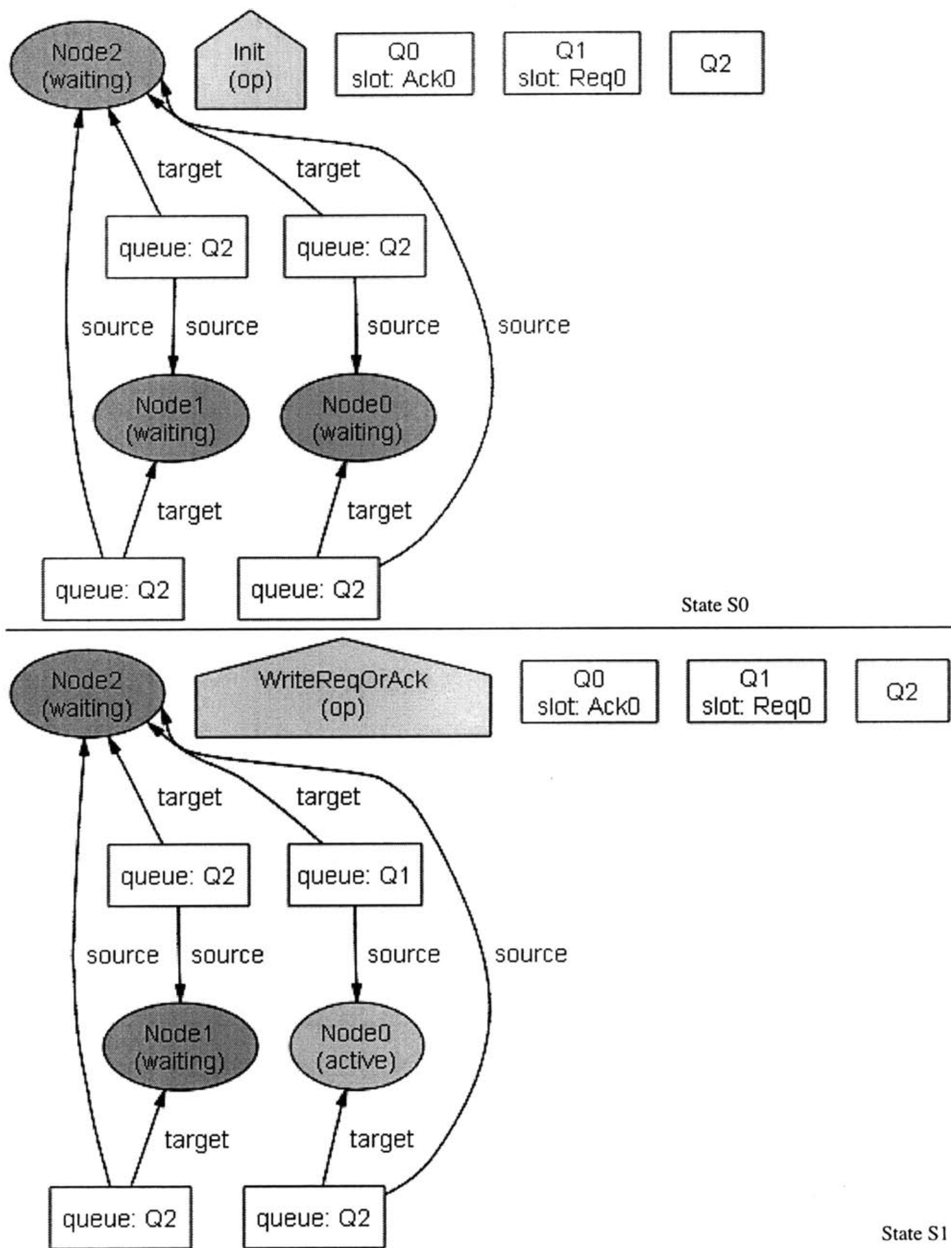


Figure 2-11: In state S0, the system is initialized and the nodes are waiting. Each link has a message queue which is initially empty. `Node0` has only one incoming link, and thus it has only one incoming link which has not been classified as a `parentLink`. In state S1, `Node0` becomes active and sends a request to `Node1`, declaring its willingness to be a child.

Figure 2-12: In state S2, Node2 acknowledges Node0's request, indicating that Node2 is willing to be Node0's parent. Node0 thus declares that its outgoing link is a parentLink, making it a leaf in the tree being constructed. In state S3, Node2 becomes active and notices that it how has only one outgoing link which is not marked as being a parentLink. It thus sends a request to Node1, indicating its willingness to be a parent.

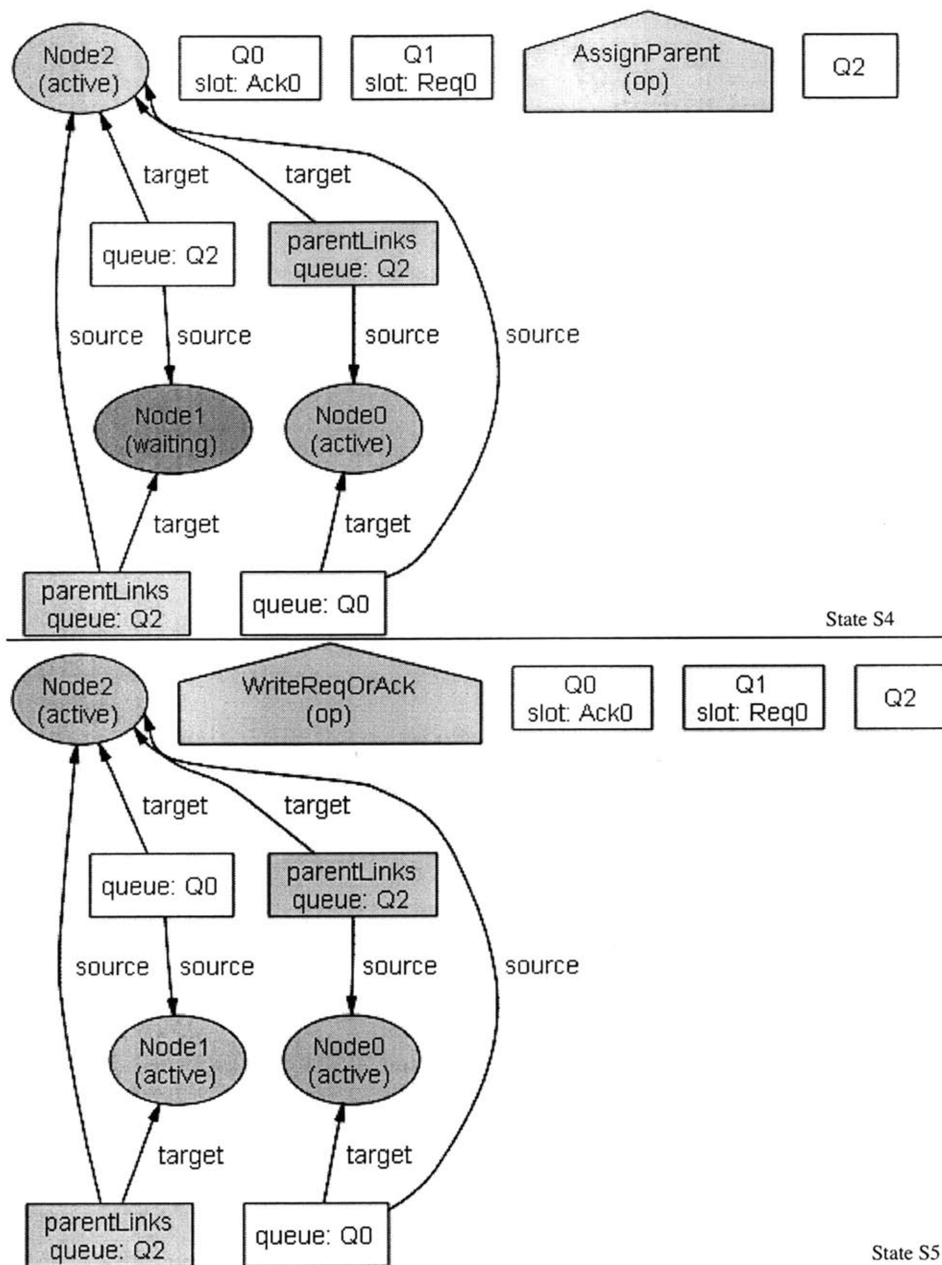


Figure 2-13: In state S4, Node1 acknowledges Node2's request, indicating its willingness to be a child. Node2 sees that only one of its incoming links is not a parentLink, so it sends a request to Node1 that the other incoming link be made a parentLink. In state S5, Node1 activates and sees that it can only be a leaf node.

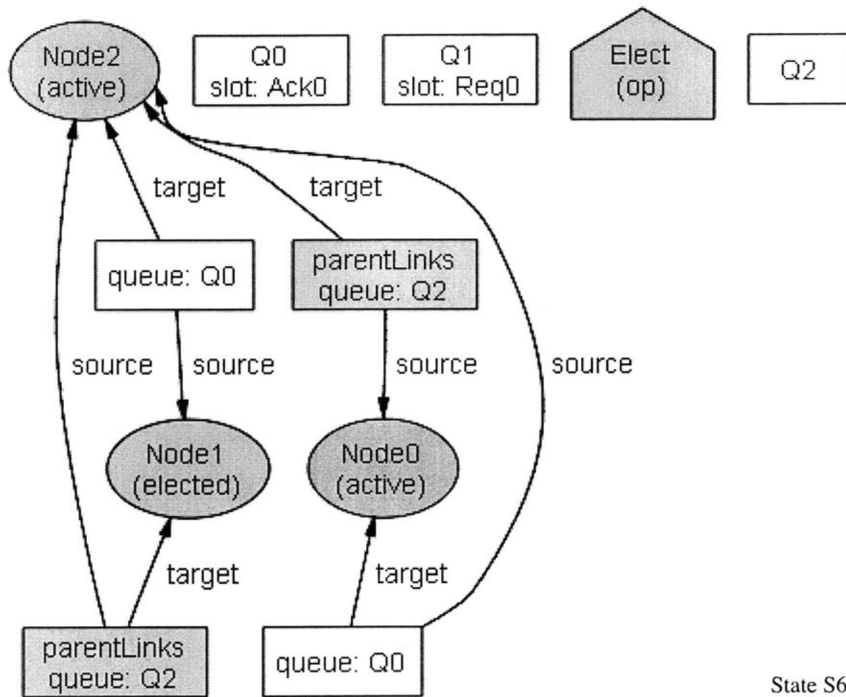


Figure 2-14: In state S6, all of Node2's incoming links are parentLinks, and those choices have been validated by its neighbors. Node2 thus declares itself to be the root, and the algorithm terminates.

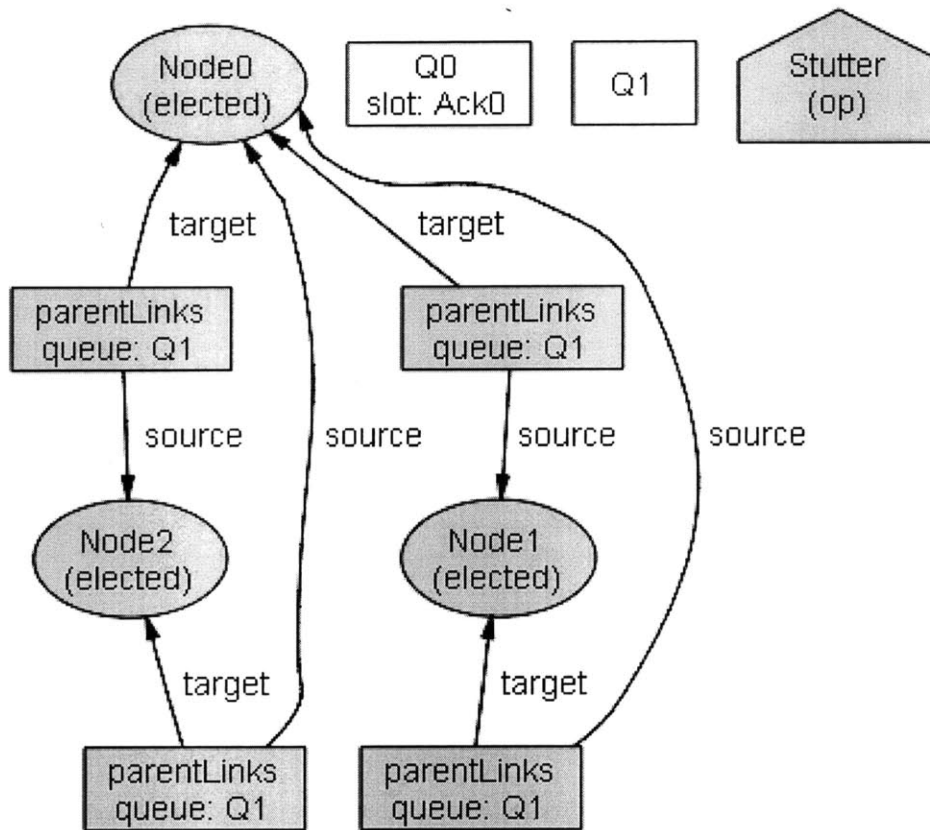


Figure 2-15: One state from a trace in *PCD* from the entire constraint describing of the `stutter` operation

from the predicate `Trans` [12]. The constraint says that a state following the `Stutter` operation must be equivalent to the state preceding the operation. Stuttering just allows the system to make a no-op transition.

One might suspect that deleting this subformula would effectively disallow that operation from occurring. However, the *PCD* entry of the conjunction diagram (Figure 2-15) tells a different story.

Failing to describe `Stutter` does not disallow it, but rather it allows it to take on any behavior! What's going on is that elsewhere in the model, we list the possible operations and state that exactly one of them occurs between each pair of states. If

[12] Since this formula is part of a predicate, generating the conjunction diagrams for integrity and correctness requires computing the joint role of all its call sites. However, it is only called in one location, so we need not bother. The notion of a joint role is introduced in Section ??.

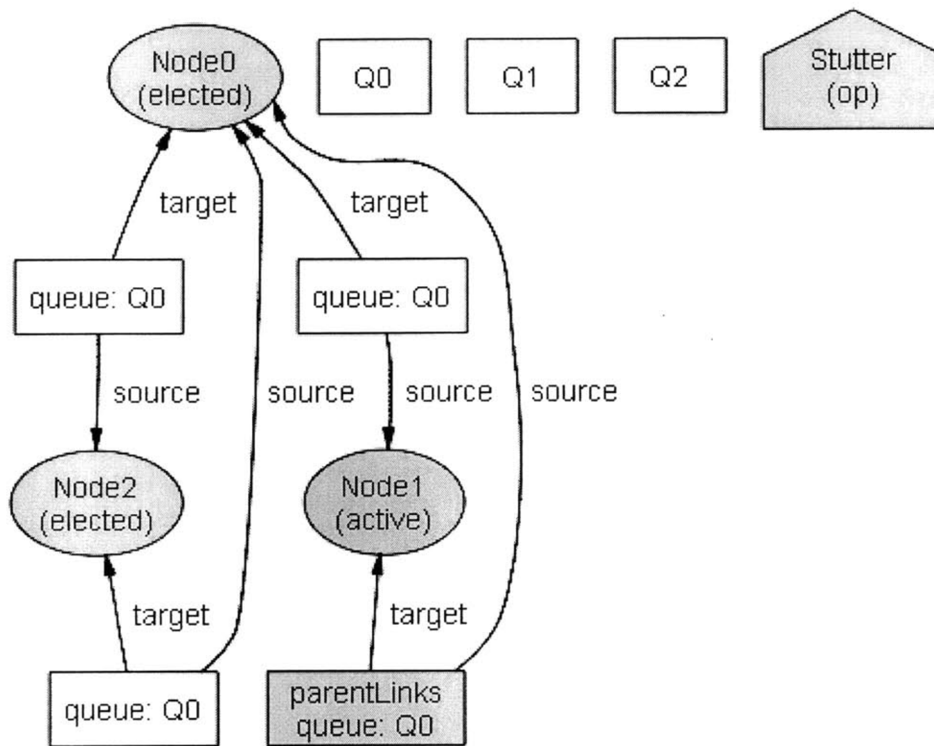


Figure 2-16: *PCD* from the conjunction diagram for the role of the latter half of the constraint `s'.op = Stutter => SameState (s, s')`

the `Stutter` operation is chosen and there is no constraint describing its behavior, then any behavior is allowed.

Refined Target Since deleting that formula had such an extreme effect on the model, its conjunction diagram was not very informative. When a target formula has too much influence on the model, we need to choose a smaller one. So we refine the target to include only the latter half of the previous target:

```
SameState (s, s')
```

PCD (Figure 2-16) contains a trace in which the system never stutters. If we look back at the model, we can confirm that observation. Deleting the new target is equivalent to setting it to the constant `False`. The target appears in the context

```
s'.op = Stutter => SameState (s, s')
```

Deleting the target and desugaring the conditional gives us the following constraint

```
!(s'.op = Stutter)
```

The `Stutter` operation cannot occur.

Looking at ICD, we see that sabotaging the current target is equivalent to deleting the entire line containing it – the situation we examined before and which is shown in Figure 2-15.

Since the `stutter` operation is so simple, let's look at a more complicated one.

```
s'.op = Elect => {
  s'.parentLinks = s.parentLinks
  some n: Node {
    n in s.active
    n in s'.elected
    NoChangeExceptAt (s, s', n)
    n.to in s.parentLinks
    QueuesUnchanged (s, s', Link)
    }}
```

The `Elect` operation occurs just before the algorithm terminates, and involves the nodes agreeing upon a node to be the root. If we choose the entire constraint, we will see the same behavior as when we deleted the entire constraint describing `Stutter`; we would be allowing the `Elect` operation to have an arbitrary effect. Similarly, if we choose the target to be the entire formula following the conditional, we would be preventing `Elect` from ever occurring.

Further Refined Target In order to understand the details of `Elect`, we need to further refine the target by making it smaller. The role that most of the lines play is obvious. This one is perhaps the most cryptic:

```
n in s.active
```

The node being elected (n) is required to have been active in the previous state. While the *meaning* of this line is not difficult to understand, it is not clear why it is *necessary*.

Looking at *PCD*, shown in Figure 2-17, we see a solution which appears to be ok. It does indeed involve an inactive node being elected as the root, but that node is a valid root, as are the `parentLink` labels.

So, is this constraint really necessary? *PCD* showed us a situation which it eliminates which need not be eliminated. However, there might be a solutions it eliminates which *should* be eliminated. Since we only show the user one arbitrarily chosen element of *PCD*, such a solution might still exist. What the conjunction diagram *does* do for us is (a) make us suspicious about this constraint and (b) suggest a property to check. We write the following property and check it against the model with the target formula deleted:

```
assert OK {
Execution() =>
  (all s: State | (some s.elected)
    => all n: Node | s.elected in n.*(~source.((s.parentLinks) <: target))
  )
}
check OK for 1 Int, 7 Op, 2 Msg, exactly 3 Node,
             6 Link, 3 Queue, 7 State
             expect 0
```

If the rules of execution are obeyed (`Execution()`), then the node elected (`s.elected`) must be a valid root. To be a valid root, a node must be reachable from any other node by following `parentLink` links some number of times. [13]

Analyzing that assertion produces no counterexamples. We now know that (in a network of up to 3 nodes) permitting inactive nodes to be elected does not produce

[13] The expression (`source.((s.parentLinks) <: target)`) is Alloy's way of expressing the relation from nodes to nodes indicated by the `parentLinks`. This roundabout way of expressing "up edges" is necessary because the model reifies links. Perhaps there is a clearer way to write the model, but our task is to understand it as it stands.

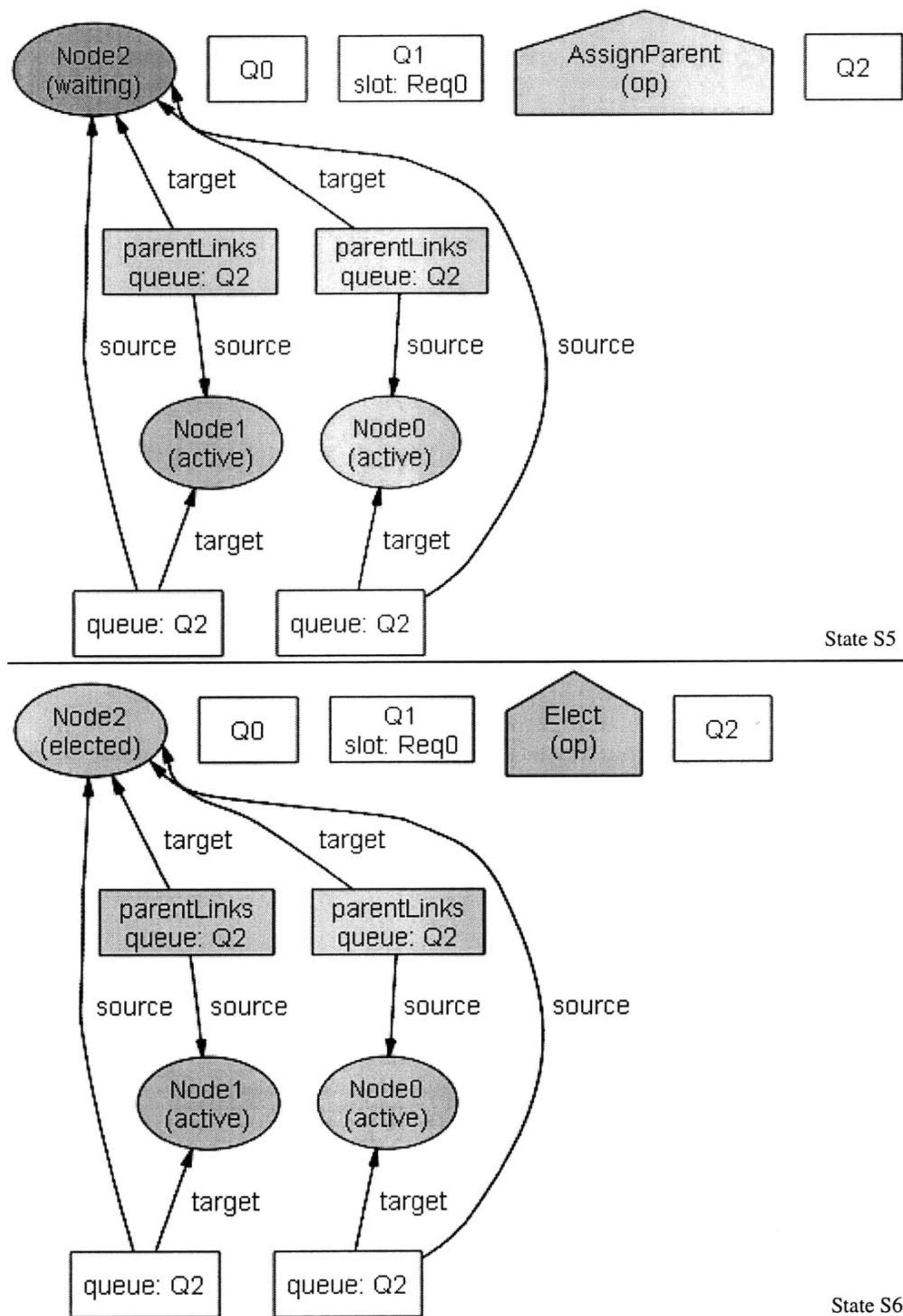


Figure 2-17: The formula `n in s.active` in the specification for the `Elect` operator is crucial to disallowing the case where an inactive node is elected as the root. Shown, are states `S5` and `S6` (the final two states) of one such trace. This trace would appear in the *PCD* entry of the conjunction diagram.

ill-formed trees. We only know to check this property because of our exploration of the role of various subformulae.

2.6 Expansion and Refinement

"The road to wisdom? Well, it's plain and simple to express:

Err and err again but less and less and less."

Piet Hein

Often, the first target formula one chooses will produce a trivial or unhelpful conjunction diagram. In such a case, *target expansion* or *target refinement* is in order.

If altering the target has little or no effect, then we probably want to expand the target to include more of the model. In Section 2.3.2, we saw an example of using *target expansion* to investigate a CNF formula.

On the other hand, if altering the target has a tremendous effect on the model, then we should reduce the target to contain an even smaller subformula. In Section 2.5.3, we saw an example of performing several *target refinements* in a row while exploring operations in the `Firewire` network protocol.

2.7 Related Work

2.7.1 Understanding Counterexamples with `explain`

Groce, Kroening, and Lerda [1, 10] developed the tool `explain`, which address the difficulty users have with understanding lengthy solution to a model. Where our goal is to explain a particular subformula of the model, their goal is to explain a particular solution of the model. Both approaches use the insight that slightly illegal behavior is useful to explain regular behavior, and so both approaches produce explanations in the form of non-satisfying assignments to the model. `explain` is given a particular solution, so it uses a distance metric to find the most similar non-solution. Non-

example generation is given a particular subformula, so it uses mutations to that subformula to produce non-solutions that violate only that subformula.

It takes a great deal of time and effort to work through a long solution to make sure that it is not a bug in the model and to understand it sufficiently to correct the system being modeled. `explain` uses a metric to measure the distance between two assignments or solutions, allowing it to identify the "most similar solution". Given a counterexample (an assignment satisfying the model but violating the property), `explain` generates an execution that is as similar as possible to the failing run but which does not violate the property. It can also generate a new counterexample which is as different as possible from the previous run.

2.7.2 Mutation Testing

Both non-example generation and mutation testing deal with how small, artificial changes to the source code (or, in our case, the model) affect the set of valid executions (or solutions). In the case of mutation testing, the goal is to make sure that the test suite can detect *when* the set of valid runs changes. In the case of non-example generation, the goal is to explain the effect of that change by showing *how* the solution set changes.

Mutation testing has been well-known to computer scientists for decades [28, 43, 17, 25, 32, 37]. It is based on applying mutation operators to the source code of a program to seed that program with faults. A test case which is able to distinguish the mutant from the original program is said to "kill" that mutant. If a test suite kills all the mutants generated, then the test suite is complete with respect to that class of mutations; it will detect future bugs of that type. If the mutant program does not violate the test suite, then that suite is insufficient, and that mutant suggests a test case to include in the suite. An infinite test suite of all valid and invalid runs would kill all non-equivalent mutants. The goal is to select a small, finite subset of those runs which still kills non-equivalent mutants. Ensuring that the mutations are semantically different from the original program is a major topic of research.

One could apply mutation testing to a model instead of a program. One would

provide a class of mutations and a set of test cases (assignments). Such an approach would be very different than non-example generation, since it would test how the chosen assignments were affected by that class of mutations. It would not explain how arbitrary changes to a particular formula affect the set of all solutions.

2.7.3 Logic Programming: Prolog

Declarative modeling is similar in form and philosophy to *logic programming*, for which *Prolog* is perhaps the best known language [14]. As with a declarative model, a logic program is a set of declarative statements which describe restrictions on results rather than dictating valid paths. There are many debugging tools for Prolog, but the expressiveness limitations of the language prevent applying non-example generation to that domain.

Logic Programs vs. Logical Models

The execution of a model checker is essentially a search of the finitized state space, using a SAT solver to analyze a boolean encoding of the model. In contrast, the execution of a Prolog program is effectively an application of theorem proving by first-order resolution – a process using unification, tail recursion, and backtracking. For efficiency reasons, many implementations estimate unification in an unsound manner.

Prolog cannot express arbitrary first-order predicate logic; a Prolog program is a set of *Horn clause*,[15] which takes the form of a conditional with a conjunctive normal form (CNF) antecedent (with all positive terms) and single term consequence: $CNF => P$.

[14]The name *Prolog* is taken from *programmation en logique*, and the language was developed by Alain Colmerauer in the early 1970's.

[15]The term *Horn clause* is derived from the logician Alfred Horn who first pointed out the significance of such clauses in his 1951 article "*On sentences which are true of direct unions of algebras*", Journal of Symbolic Logic, 16, 14-21.

Prolog Debuggers

In 1983, Shapiro [38] introduced a debugging system for Prolog, and most modern debuggers still take the same form [20]. Debugging is a process of writing meta-programs, which are themselves written in Prolog [29, 36]. These meta-programs provide information about the intended semantics of the program. They are provided by an oracle (i.e. the user) and typically take the form of a partial specification. Debugging a Prolog program in this style is analogous to debugging an Alloy model by writing assertions about it. The assertions help to link the user's intention with the implementation, and both Prolog debuggers and Alloy assertions can be automatically checked. The user can provide assertions about the behavior of the model as a whole, but cannot inquire about how some particular portion of the model contributes to that behavior.

There are more sophisticated tools for understanding and debugging Prolog, even some which generate visualizations. Hopfner and Seipel [33] have developed a method which answers the question "What parts (files, modules) of a Prolog-project are needed in order to make a specific predicate work correctly?"; "[I]f one only wants to export [or reorganize] parts of a project...then it is sufficient to install only these parts in the new environment.". The notion of "works correctly" is one of explicit dependence, not of implicit interaction; it is about what parts of the program are needed in order to make the target well formed, not which parts interact with the target, or what would happen if the target were absent.

Non-Examples in Prolog?

Due to the limited expressive power of Horn clauses, generating non-examples for Prolog is not as straight forward as it is in a language such as Alloy which supports arbitrary first order logic, and it may not even be possible. One cannot arbitrarily negate a subformula of the model without it ceasing to be a Horn clause.

2.7.4 Near Miss Learning

Patrick Winston's classic work on *near miss learning* [46] has the same psychological foundations as non-example generation. Both techniques are based on the notion that, in order to understand a category, one must see near-misses – examples which are almost (but not quite) elements of that category.

Non-example generation is for understanding models, which we view as a description of a set of (satisfying) assignments. Near misses are assignments which violate the model but would satisfy it if the chosen subformula were altered (PCD, ICD, and CCD). We also use a notion of a near-hit – assignments which satisfy the model but wouldn't if the chosen subformula were altered (PCA, ICA, and CCA).

Winston's work is in artificial intelligence, and is concerned with training a robot to correctly categorize objects it sees. His approach involves presenting the machine with positive examples, negative examples, and near misses. In this manner, Winston trains the machine in the same way we train the user.

Winston's Approach Consider the task of training a machine to learn which inputs satisfy some function, f. There may be a partial description of f, or none at all. You can train the machine by giving it positive examples, which are known to satisfy f, and negative examples, which are known to not satisfy f. There is also a notion of a *near miss*, introduced by Winston in 1970 [45], which is a negative example which only fails to satisfy f for "a small number of reasons". Winston's classic example is that of an arch.

- **positive:** two non-touching blocks supporting a third
- **negative:** fifty blocks line up in a row
- **near miss:** two touching blocks supporting a third, or two non-touching blocks with the third on the ground nearby.

A set of positive examples and near miss negative examples (a *near miss group*) is an effective training method when combined with half a dozen heuristics for how

to restrain or relax the machine's model of f based on those example. Interested readers will find clear descriptions of these heuristics, and this approach to learning in general, in Patrick Winston's text book [46].

Winston gives two guiding principles to help understand why it is that near misses are so helpful to learning, be it by a machine or a human:

1. *You cannot learn if you cannot know.* Good teachers help their students acquire the necessary representation [thus allowing them to behave correctly in new or extreme situations].

2. *You cannot learn if you cannot isolate what is important.* Good teachers help their students by providing not only positive examples, but also negative examples and near misses [to identify what is important about the positive examples].

Counterfactual and Causal Reasoning

In Section 2.1.2, we motivated non-example generation as a lightweight alternative to counterfactual reasoning. The brief summary given in that section does not do justice to David Lewis's work on formalizing counterfactual logics.

David Lewis [31] suggested the defining counterfactuals in terms of "closest possible worlds". A question "What would have happened if event X had occurred?" is answered by examining all possible worlds in which X occurred. Among those worlds, the one which is closest to the current world is selected. His exact words on the matter are as follows:

"A [counterfactually implies] C is true in the actual world if and only if (i) there are no possible A-worlds; or (ii) some A-world where C holds is closer to to the actual world than is any A-world where C does not hold.

We shall ignore the first case in which the counterfactual is vacuously true. The fundamental idea of this analysis is that the counterfactual A [counterfactually implies] C is true just in case it takes less of a departure from actuality to make the antecedent true along with the consequent than to make the antecedent true without the consequent."

Chapter 3

Core Extraction: Identifying Overconstraint

"Logic takes care of itself;
all we have to do is to look and see how it does it."
Ludwig Wittgenstein

"Logic, like whiskey, loses its beneficial effect
when taken in too large quantities."
Lord Dunsany

Along with its many benefits, declarative modelling brings the risk of overconstraint, in which real counterexamples are masked by bugs in the model. In the extreme, the model has no bad transitions because it has no transitions at all! *Core extraction* is a new analysis that mitigates this problem in the presence of a checker which translated the model to a CNF formula based on reduction to SAT. It exploits a recently developed facility of SAT solvers to deduce an unsatisfiable subset of a CNF formula which is often much smaller than the clause set as a whole. This unsatisfiable "core" is mapped back to the syntax of the original model, showing the user the fragments of the model which caused it to be unsatisfiable (and thus revealing unused or irrelevant portions of the model). Relevance information can be a great help in discovering and localizing overconstraint, sometimes pinpointing it immediately. The

construction of the mapping between the model and an equivalent CNF formula is given for a generalized modelling language, along with a proof of the soundness of the claim that the marked portions of the model are irrelevant. Experiences in applying core extraction to a variety of existing models are discussed. [1]

3.1 Introduction

The risk of overconstraint in declarative specification languages such as Z [41] and VDM [24] was recognized long ago, but only very limited automatic tool support exists to mitigate it. In Z, preconditions are implicit; it is regarded as good style for an operation's precondition to appear explicitly in the text of the operation's schema. This discipline results in proof obligations (that the explicit conditions imply any implicit preconditions). Checking these obligations is no easier than checking any other property of a Z specification[2]. Similarly, in VDM, the 'implementability' criterion leads to a similar obligation. Because of the difficulty of discharging proof obligations automatically, most tools for Z and VDM simply extract the proof obligations but leave the user to determine their validity.

Analysis tools that support simulation or the checking of liveness properties can mitigate the problem of overconstraint, but the risk remains: a safety property may hold because of a subtle overconstraint that may not be noticed even if a host of liveness checks have passed. Moreover, counterexamples to liveness may themselves be ruled out because of overconstraint.

Even when a modeler suspects an overconstraint, identifying the conflicting constraints is often frustrating. Currently, the only systematic technique for finding causes of conflict in a declarative model is to manualy disable individual constraints until the culprits are identified. This task can be lengthy and runs the risk of introducing new errors into the model. The model checker provides no help to the user in

[1]Chapter 3 is isomorphic to a paper co-authored with Ilya Shlyakhter, along with Manu Sridharan, Daniel Jackson, and Mana Taghdiri [19]. For example, the introduction was originally written by Daniel Jackson and only minor changes have been made.

[2]And may indeed be harder, since the precondition of an operation involves a higher-order quantification over its state components.

finding the overconstraint, other than to report whether a given version of the model is still overconstrained. As discussed in Section 5, 'vacuity testing' addresses this problem [30, 3, 6, 44], but does not apply to declarative models and helps debug only overconstrained properties, not overconstrained models.

This chapter presents *core extraction*, a new approach to addressing the problem of overconstraints in declarative models. Satisfiability solvers have recently developed a facility for extracting the *unsatisfiable core* of a CNF formula: that is, a subset of the clause set sufficient to cause a contradiction [47, 14]. For declarative model analyses that can be cast as satisfiability instances, the unsatisfiable core can be mapped back onto the model. In other words, we can identify the parts of the model responsible for producing the unsatisfiable CNF core. Those parts, by themselves, suffice to produce an overconstraint, and their identification can help the user find the overconstraint.

Showing an unsatisfiable core may also alert the user to the unexpected presence of an overconstraint. If almost the entire formula is relevant (i.e. necessary to preventing counterexamples to the property being checked), it will raise confidence that the system description is not overconstrained, that the safety property is not vacuous, and that it holds (at least for the bounded domain). But if the analysis highlights only a small part of the system description (or does not highlight the property being checked), it indicates as strong possibility that the model is unintentionally overconstrained. If it highlights only the safety property, it suggests that the property is a tautology, and thus vacuously satisfied.

Our presentation is set in the context of an analysis for first-order relational logic that has the flavor of model checking. In short, a system is specified as a formula, whose models (which correspond to the behaviours of the system) assign values to relations of various arities. A safety property is checked by conjoining its negation to this formula; solutions to this new formula are counterexamples. If there are no solutions, and the model is correct, then the property being checked holds (up to the specified scope). The formula is translated to a propositional formula by bounding the carrier sets from which the relations draw the values of their atoms. Models of the propositional formula are found by a SAT solver, and translated back into the

relational domain for display to the user. This analysis scheme has been described previously [22]; until now, if no counterexample were found, no further information would be given. This lack of information has been the a complaint from users of our tool.

Although we developed these techniques in the context of the declarative modeling language Alloy, which we will use to present case studies and examples. However, both the technique and its implementation were intentionally kept much more general; they are sufficiently modular to apply to any language which is reducable to SAT in a structure-preserving fashion [3]. Concequently, our techniques should also apply in related settings such as BMC [4] and planning [26]. We provide a simple semantic guarantee of correctness, assuring the user that deleting constraints identified as irrelevant will preserve unsatisfiability of the model.

3.2 A Toy Example

To illustrate the use of core extraction on a toy example, consider the problem of checking the design of a web caching scheme. The task is to design the `Get` operation that obtains a page from its owner, or from a cache. The correctness of `Get` will be contingent on some assumptions about the freshness of pages delivered by the owner, which are recorded as an invariant. A complete (albeit simplified) Alloy model for this problem is shown in Figurefig:webcache.

A detailed knowledge of Alloy is not needed to grasp the point of the example, but some explanation is in order. A fuller explanation of Alloy may be found elsewhere [23].

Each signature (labeled `sig`) introduces a set of atoms: `Time` for the atoms representing moments in time, `URL` for the URL's of documents, `Page` for the contents of documents, and `Server` for the caches and owners. A field declared within a signature is simply a relation of some arity, whose columns are the sets given, and implicitly, in

[3]Our techniques and implementation still apply to non-structure-preserving translations, but the more structure is lost, the less useful (larger and less likely to pinpoint the source of an overconstraint) the extracted core will be.

Figure 3-1: A toy Alloy model describing the behavior of a web cache

```
module webcache

sig Time {}
sig URL {}
 //A Server records (at most) one Page
 // per URL at any given time.
sig Server {page: Time -> URL ->? Page}
 //Page expiration is modeled by a set
 // of times at which the page is fresh.
sig Page {life: set Time}

//Page recorded by any Server is fresh
fact ServerFresh {
  all s:Server, t:Time,
     u:URL, p:Page |
    (t -> u -> p) in s.page =>
       t in p.life  }

//The cache may drop & add entries from the owner,
//but no stale pages may remain afterwards
fun Get (t,t':Time, cache,owner:Server,
         u:URL, p:Page) {
  cache.page[t'] in cache.page[t]
    + owner.page[t] -
    {u:URL, p:Page | t' not in p.life}
    => p in (cache+owner).page[t'][u] }

//result of Get is always a fresh page
assert Freshness {
  all t,t':Time, cache,owner:Server,
    u:URL, p:Page |
    Get (t, t', cache, owner, u, p)
      => t' in p.life  }

check Freshness
```

the leftmost position, the signature itself. Thus `life` is a binary relation from Page to `Time`, associating with each page the set of times for which it is current, and page is a relation of arity 4, containing a tuple `(s,t,u,p)` when server `s` maps URL `u` to page `p` at time `t`. The question mark in the declaration of the field `page` indicates that at most one page is associated with a given server, time and URL.

The remaining paragraphs of the model are formulas that play different roles. An assertion is a formula that is conjectured to be valid; here `Freshness` asserts that the result of a `Get` is always a fresh page. The assertion makes use of a function, `Get`, which is a parameterized formula that is simply inlined, and (implicitly) any global facts. It's arguments comprise two times (`v` and `v'`), two servers (`cache` and `owner`), a URL (`u`), and a page (`p`). In this case, the fact `ServerFresh` is implicitly applied, which states that servers always yield fresh pages. The command `check Freshness` instructs the tool to search for counterexamples to the assertion; by default, the search is conducted in a 'scope' that limits each basic set to 3 atoms.

The formulas within the fact, function and assertion are written in an ASCII form of first-order logic, enriched with relational operators. The keyword `in` is the subset operator, `+` is set union, and `-` is set subtraction. The dot and square brackets are two variants of a single relational image operator with different precedence and argument order. For example, the expression

```
(cache+owner).page[t'][u]
```

does several things: first it takes the union of the two sets `cache` and `owner`; then it takes their image under the relation `page`; then takes the image of first the time `t'` under this relation, yielding a relation from URL's to pages representing the aggregate contents of cache and server at time `t'`; then takes the image of the URL `u` under this relation; and finally gives the set of all pages associated with the URL `u` in cache or owner at time `t'`.

The times `t` and `t'` represent the moment just before and just after the `Get` occurs. The function as a whole can be read as follows: the mapping from URL's to pages in the cache after (`cache.page[t']`) is a subset of the union of the mapping before

(`cache.page[t]`) and the mapping in the owner (`owner.page[t]`), minus all entries whose pages are no longer fresh ({`u:URL, p:Page | t' not in p.life`}). In other words, the cache is at liberty to drop any entries, and to add any entries from the owner, so long as no stale pages remain afterwards. The fact is the crucial invariant recording the assumption that pages in a server are always fresh.

In this case, there is no counterexample (and there would be no counterexample even in a larger scope). We run the core extraction analysis; its output is an abstract syntax tree, annotated (roughly speaking) with information about which nodes are relevant, and, for those representing formulas with free variables, for which values of the variables. We examine the tree top-down, looking for formulas that are deemed irrelevant. Surprisingly, the entire first line of `Get` is irrelevant: even though the page may be taken from the cache (as specified in the second line), how the cache is updated is irrelevant.

What's wrong? First we check that the page is taken from the new and not the old value of the cache. No problem here; the second line of `Get` correctly refers to `t'` and not `t`. Then we see the blunder: the fact states that *all* servers yield fresh pages, including the cache, so the assertion follows trivially from it. This might seem like an absurd error to make, but in fact, an author of this paper made this error unwittingly during the development of this example because he started with a more elaborate model that distinguished caches from servers, then simplified it erroneously.

To fix the model, we partition the set of servers into caches and owners, modify the fact to constrain only owners, and declare the arguments of `Get` to belong to the appropriate subsets. This correction is shown in Figure 3.2.

Running core extraction again shows that almost all formulas in the function and fact are relevant. The expression

```
c.page[t] + o.page[t]
```

in the first line of `Get`, however, is marked as irrelevant. This makes sense though; since all stale pages are removed from the cache (by the set subtraction that follows), the source of additional pages is irrelevant. That the remaining formulas are deemed

Figure 3-2: The corrected version of the web cache Alloy model, taking into account the information provided by core extraction

```
module webcache

sig Time {}
sig URL {}
sig Server {page: Time -> URL ->? Page}
part sig Cache, Owner extends Server {}
sig Page {life: set Time}

fact  OwnerFreshness {
  all s: Owner, t: Time,
    u: URL, p: Page |
    (t -> u -> p) in s.page =>
       t in p.life  }

fun Get (t,t':Time, c:Cache,
         o:Owner, u:URL, p:Page) {
  c.page[t'] in c.page[t]
  + o.page[t] -
    {u:URL, p:Page |
     t' not in p.life} p in
      (c+o).page[t'][u]  }

assert Freshness {
  all t,t':Time, c:Cache,
   o:Owner, u:URL, p:Page |
    Get (t, t', c, o, u, p)
    => t' in p.life  }
check Freshness
```

relevant gives us some confidence that our model is no longer vacuous; the irrelevance of this particular expression suggests that an assertion about the authenticity of a page is needed.

As another example, suppose we erroneously wrote

```
p in c.page[t'][u]
```

for the second line of `Get`, so that the page is always taken from the cache, and never from the owner. In this case, the fact `OwnerFreshness` is found in its entirety to be irrelevant: not surprisingly, since the cache from which the page is drawn has been purged of stale entries.

3.3 The Core Extraction Algorithm

Core extraction is run only on formulas already found to be unsatisfiable: either simulations that have no behaviours, or checks that have no counterexamples. It runs fully automatically without user intervention. Its result is an identification of fragments of the original model that were irrelevant to demonstrating unsatisfiability. In our implementation, these fragments are shown as a colored abstract syntax tree that is synchronized with the text in an editor. An irrelevant fragment is colored red, and may be a subformula or an expression. For quantified formulas, the tree indicates for which values of bound variables the body formula is irrelevant.

This section presents the algorithm on which our implementation is based, in a generalized form. It starts with an abstract syntax tree (AST) in which there are no quantifiers; our implementation involves an additional step of mapping back from this tree, resulting from grounding out quantifiers, to the original syntax tree.

The correctness criterion for the algorithm is that the claim of irrelevance is sound. More precisely, the function computed at any AST node marked as irrelevant can be replaced by any function that leaves the AST well formed (in particular a constant function of the appropriate type), without making the resulting AST satisfiable. An irrelevant child of an `AND` node, for example, can be read as the constant `true`. In

some cases, as here, this means that the node can be removed entirely. A roadmap of how the algorithm fits into the overall use of the tool algorithm is shown in Figure 3-3.

3.3.1 Constraint Language

We define an abstract constraint language for expressing formulas on a collection of variables $v_i \in V$. A formula is expressed as an abstract syntax tree, in which each node computes a predefined function of its children. The root node computes a Boolean function, which becomes the value of the formula as a whole. The leaf nodes are variables. We denote the universe of computable values by U. The set F of node functions has elements of the form $f_i : U^* \rightarrow U$ Trees are thus defined by $Tree = F \times Tree^* + V$. [4]. An assignment from U to each $v_i \in V$ induces an assignment from U to each AST node; the value of a leaf node is the value of the AST variable at that node, while the value of a node $n = Tree(f, ch)$ is computed by applying the node function f to the sequence of values assigned to the children ch. Testing satisfiability of the AST involves finding an assignment to the variables v_i which induces the value *true* in the root node, or determining that none exists.

3.3.2 Translation

Satisfiability of the formula can be tested by converting it to a Boolean formula in conjunctive normal form (CNF). The translation framework is illustrated in Figure 3-4. To convert an AST to CNF, we allocate to each AST node $n \in Tree$ a sequence of Boolean variables $bv(n) \in BV^*$ representing the node's value. The sequences of Boolean variables allocated to two nodes are identical if these are leaf nodes with the same AST variable, otherwise the sequences are disjoint. We define functions $enc : U \rightarrow Bool^*$ and $dec : Bool^* \rightarrow U$ for encoding and decoding values in U as binary strings. An assignment of Boolean values to all the Boolean variables allocated for AST nodes thus corresponds to assigning a value from U to each AST node. An

[4]The sets U and F are determined by the semantics of the particular constraint language. For example, for Alloy [23], U contains relational and Boolean values, and F includes relational and Boolean operators. The particular semantics are unimportant for this paper.

assignment of U values to AST nodes is consistent if the value at each non-leaf node equals the result of applying the node's node function to the sequence of U values assigned to the node's children. We translate an AST to CNF by generating CNF clauses on the Boolean variables allocated to AST nodes, so that the conjunction of these clauses is true of a given assignment to Boolean variables iff the Boolean assignment corresponds to a consistent assignment of U values to AST nodes.

The translation is done separately for each AST node. For each node, we produce a set of CNF clauses relating the Boolean variables allocated to that node, to the Boolean variables allocated to the node's children. Intuitively, the clauses are true iff the U value represented by the nodes's Boolean variables equals the result of applying the node's node function to the sequence of U values represented by the Boolean variables allocated to the node's children. The clauses output from translating an AST node depend only on the node function which the node computes of its children, and on the Boolean variables allocated to the node and the children.

For each node function f_i, we define a corresponding "CNF translation" function

$$\hat{f}_i : BV^*, BV^{**} \rightarrow \mathbb{P}\ Clause$$

$\hat{f}_i$ takes a sequence of boolean variables from the domain BV, corresponding to the result of the function, and a sequence of sequences of boolean variables corresponding to the arguments, and returns a set of clauses that encode the function in CNF. The correctness of this function is justified with respect to the encoding function and the semantics of f_i itself; its result evaluates to true iff the Boolean variables allocated to the result of f_i encode the value computed by applying f_i to the argument values encoded by the Boolean variables allocated to the arguments.

Using these individual translation functions, we can now translate the tree. The function $transl : T \rightarrow \mathbb{P}\ Clause$ translates one AST node to CNF, and is defined as

$$transl(t) \equiv \text{let } t = Tree(f, ch) \mid \hat{f}(bv(t), map(bv, ch))$$

The CNF translation of an entire AST is then just the union of translations of its

nodes:

$$translTree(t) = \cup_{n \in nodes(t)} transl(n)$$

Correct translation to CNF requires that for each node t, for any Boolean assignment $ba : BV \rightarrow Bool$ satisfying $transl(t)$, we have

$$\begin{aligned} &f(map(dec, map(\lambda\ cv\ .\ map(ba, cv), map(bv, ch)))) \\ &= dec(map(ba, bv(t))) \end{aligned}$$

where the node t computes the node function f of its children ch. To test satisfiability, we constrain the Boolean variable(s) allocated to the root to represent the value $true$ from U, by adding the appropriate unit clauses.

3.3.3 Mapping Back

Suppose now that the CNF C translated from our AST is unsatisfiable, and the SAT solver identifies an unsatisfiable core $C' \subseteq C$. We define a predicate $irrel : T \rightarrow Bool$ on AST nodes, which is true for nodes whose translations contributed no clauses to the unsatisfiable core:

$$irrel(t) \equiv \{t \mid transl(t) \cap C' = \emptyset\}$$

Claim: For any node n for which $irrel(n)$ holds, we can replace the node function f_i with an *arbitrary* node function f_j without making the AST satisfiable. To show this, we argue that the CNF translation of the mutated AST will still include the unsatisfiable core.

Proof: The function bv, which allocates Boolean variables to AST nodes, does not depend on node functions; the sequence of Boolean variables allocated to a given AST node depends only on the overall structure of the AST and the position of the node within the AST. Therefore, the same sequences of Boolean variables are allocated to all AST nodes in the mutated AST as in the original AST.

For any node whose node function has not changed, $transl$ will thus output the

same clause set. Any node n whose clause set contributed to the core will node function, and *transl* will output the same clause set for that node. Each clause of the unsatisfiable core is thus present in the translation of the mutated AST, meaning that the mutated AST is still unsatisfiable.

3.3.4 Complications

The description of the AST above was made rather abstract to make it clear that although we have implemented the scheme for Alloy, it could be applied straightforwardly to any constraint language that can be reduced to SAT. The description of the translation is likewise more abstract, because this allows it to accommodate more advanced translations. The Alloy Analyzer, for example, identifies opportunities for sharing among subformulas [39], so that the AST is in fact not a tree but a DAG. This can be modelled by having the *bv* function allocate the same Boolean variables to different AST nodes. Similarly, like most tools that generate CNF, the analyzer uses auxiliary Boolean variables to prevent CNF explosion; this can be modeled by having *bv* allocate additional Boolean variables to each AST node.

3.4 Experience

To evaluate the technique, we performed a variety of experiments and small studies. First, we examined our logs of common mistakes made in modelling, and identified those that would likely be detected by core extraction. Second, we revisited some models that had suffered from overconstraint during their construction, reinstated the overconstraints, and ran the core extractor to see how it fared. We describe a case in which it worked well, and some in which it did not. Third, we ran the extractor on several models for which we had no expectation of overconstraint; to our surprise, we found some serious flaws.

3.4.1 Common Mistakes

In this section, we will describe some pitfalls that we have observed (and had reported to us by users) to be common in Alloy modelling, and show how core extraction helps to highlight them. Mistakes in the use of a formalism are of course less interesting than true conceptual mistakes, but their consequences can be just as painful (and just as much of a deterrent for potential modelers). Arguably they reflect flaws in the language design, but no language is perfect, and all include similar potentials for error.

Pitfall #1: assuming variables with different names have distinct values

A quantifier may not bound its variables as intended. In a model with signatures `Person` and `Name`, and a relation `name` from `Person` to `Name`, one might write

```
all p, q : Person | p.name != q.name
```

to say that each person has a unique name. But this formula is always false unless the set `Person` is empty, since it cannot be satisfied when `p = q`. The extracted core would likely include the property being checked[5], a constraint requiring non-emptiness of `Person`, and this quantified formula. Writing instead

```
all p : Person, q : Person - p |
   p.name != q.name
```

would result in a much larger core (suggesting that it is correct, or at least more sensible).

Pitfall #2: omitting a special case for the final state of a trace

A similar blunder is sometimes made in trace-based models in which the modeler constraints transitions between states. We often lift a constraint on state pairs to a

[5]If the model is overconstrained and no instances are possible, then even the property being checked may be omitted from the extracted core. In such a case, there exists a core which does not contain the property being checked, although there is no guarantee that that particular core will be extracted.

constraint on traces so that we can analyze traces using bounded model checking. An example of such a lifting formula is the following:

```
all s : State |
  LegalTransition (s, s.next)
```

where `LegalTransition` is the formula for the transition relation, and `next` is a relation that maps a state to its successor. This particular attempt is flawed, and will yield an empty trace set. The set of states is finite; usually, we bound it by the machine diameter, which we have computed with the analyzer's help. There is therefore a last state, for which `s.next` will be the empty set, thus making any fact about next states vaccuously false. This is a classic "fencepost error", and modelers are just as prone to making such a mistake as are other programmers. The extracted core will show that the body of the quantified formula is only relevant for the last state, since the presence of that state alone will make facts about `s.next` fail. This is a big red flag for the user, who presumably expects most or all of the states to be relevant. The correct formula is

```
all s : State - LastState |
   LegalTransition(s, s.next)
```

Pitfall #3: Confusing the given constraints and their intended concequences

Novice modellers often make the mistake of writing a constraint explicitly that should instead be implied by the other constraints of the system. Consider a leader-election algorithm that should allow for at most one leader at any given time. Using our paradigm for modelling traces of algorithms, a beginner may write the following erroneous declaration:

```
sig State {
   leader : option Participant, ...
}
```

The `option` keyword by itself constrains `leader` to map each `State` atom to at most one `Participant`. A correct declaration would use the `set` keyword instead, allowing for any number of leader participants in a state and forcing other constraints to enforce the property of at most one leader. When checking the property, the core will make the error in the declaration obvious; it will contain only the constraint generated because of the use of `option`.

3.4.2 Locating Known Overconstraints

We took flawed versions of two models known to suffer from overconstraint an extracted their cores. These are exacting tests, since they represent the hardest cases we know of. There have been many simpler cases of overconstraint that we did not record which core extraction would likely isolate immediately, but still took hours to track down manually.

Iolus

The more successful case involved an analysis we performed [42] of Iolus, a scheme for secure multicasting [35]. In Iolus, nodes multicast messages to other nodes within a group whose membership changes dynamically. Scalability is achieved by partitioning groups into subgroups, arranged in a tree, each with its own Key Distribution Server (KDS) maintaining a local encryption key shared with members of the subgroup. When a member joins or leaves a subgroup, its KDS generates a new local key and distributes it to the updated list of subgroup members. This was modelled by specifying that after a member joins or leaves, there is a key shared by the new members, and no others. By mistake, the model said the key was shared by the members of the entire *group* – thus including all nodes in contained subgroups. This severely restricted the trace set, potentially masking errors.

We attempted to detect this overconstraint using our constraint core functionality. We first checked an assertion stating that no node can read messages sent to the group when that node was not a group member, one of the correctness properties of the

system. There was no counterexample, and unfortunately, the extracted core included most of the constraints in the model. This result can be explained as follows. The error in the model is only a *partial* overconstraint; while the error excludes some legal traces of the system, it still allows many traces violating the correctness property. Therefore, it is not surprising that most of the other constraints in the system are still required to establish correctness. Just because the core contains most of a model does not, unfortunately, imply that the model is free of overconstraint.

One method of finding overconstraints in this situation is to check correctness properties on a restricted set of traces, where it is still expected that most constraints of the model must be in the core. For the Iolus model, we attempted to check the aformentioned correctness property on traces that had at least three key distribution servers (constraining the size of relations is a typical way to restrict the search space). With this additional restriction, the core no longer included the constraints defining the transitions of the system or the formula stating the property, a clear indication of overconstraint.

Two observations should be noted. First, when an overconstraint is more partial and subtle (as in this case), some thinking by the user will be necessary to find its source, even after the constraint core identifies its existence. This issue is fundamental; when several formulas in a model together overconstrain the system, the core can help to identify them by eliminating irrelevant formulas from consideration, but the reason why the remaining formulas contradict each other may still not be obvious. Second, while this process of checking assertions in restricted spaces to find overconstraints lacks automation, it still has important advantages over the process of finding these overconstraints manually (without core extraction). Previously, a user who suspected an overconstraint in a model would search for it by explicitly checking that classes of legal traces were not ruled out by the system. Our new method of inspecting cores over restricted sets of traces gives more useful information; even if a class of traces is not entirely ruled out by a model, the core may show that important constraints are irrelevant for that class, showing where the overconstraint lies.

Firewire

A model of the widely studied Firewire 'tree identify' protocol [18] suffered from a modelling blunder that produced a nastily subtle overconstraint. The declarative form of the model allowed it to include a topological constraint (that the links between nodes form a connected, acyclic graph), so that analysis would cover all possible topologies involving a given number of nodes. Most model checking analyses, in contrast, hardwire a particular topology.

The model reified operations as entities, with the following declarations:

```
sig Op {}
disj sig NodeOp
   extends Op {node: Node}
disj sig LinkOp
   extends Op {link: Link}
static part sig SendRequests,
   Elect extends NodeOp {}
static part sig AddChild,
   GetResponse, Resolve
   extends LinkOp {}
```

Operations are classified into node and link operations, each associated with a particular node or link respectively. A fragment of the transition relation specification shows how this is used:

```
fun Trans (s, s': State, op: Op) {
  ...
  op in Elect => {
      s.mode[op.node] in Waiting
      ...
      s'.mode[op.node] in Elected
      }
```

Analysis of this model produced bewildering results. For 6 nodes, no trace without repeated states was found longer than 4 states, suggesting a machine diameter of 4. But an assertion that at least one node is always elected within that bound was violated. Some subset of the traces was ruled out by an overconstraint.

In retrospect, as always, the flaw was easy to see. The modeller got confused about whether the atoms of the signature `Op` represented operation types or operation instances. Thinking of them as types, he added the keyword `static` in their declarations, limiting a set such as `AddChild` to a single element. The confusing was exacerbated by the presence of a message type partitioned into requests and acknowledgments, for which it was sufficient to have exactly one message of each type (since it contained no other information). But the operation carries with it its node or link. The consequence therefore, was that each operation could only be performed on a single node or link, and for most topologies, this ruled out all but the shortest traces.

Core extraction did not give much useful information. We trimmed the model down to smaller and smaller fragments (not actually being aware of the location of the overconstraint ourselves). Along the way, core extraction helped with the pruning, but it did not pinpoint the problem. Finding overconstraints of this sort is a challenge for future work.

3.4.3 Blunders Discovered

Running our core extractor revealed flaws in two models that we had believed not to be overconstrained. We explain one of them here. In a different version of the Firewire tree identify protocol, we had added a stuttering operation at the last minute, but failed to adjust the scopes of the analyses. The declaration of operations read:

```
sig Op {}
static part sig Init, AssignParent,
    ReadReqOrAck, Elect, WriteReqOrAck,
    ResolveContention, Stutter
    extends Op {}
```

listing the 7 operation types. A command was specified as:

```
check AtMostOneElected for 6 Op, 2 Msg,
      3 Node, 6 Link, 3 Queue, 9 State
```

incorrectly bounding the number of operation types by 6. Since the declaration of operations can only be satisfied with 7, there is a glaring overconstraint. Core extraction pinpointed it immediately, showing all to be irrelevant by the declaration. While one could imagine language improvements (eg. a built-in enumerated type construct) to eliminate this specific example of overconstraint, it would be impossible for the analyzer to detect constraints on scopes in general, so this type of overconstraint will always exist.

Careless errors, like those described in this section, will occur in programs as readily as in models. Eliminating such errors from models will not eliminate them from programs. However, eliminating careless errors from models may enable discovery of subtler errors in models that would otherwise have been missed. Finding the sublter errors in models can help prevent such errors from occurring in programs.

3.4.4 Performance

For core extraction we have used a recent modification of the Zchaff satisfiability solver that added core extraction functionality [47]. We found that Zchaff's performance supports interactive identification of overconstraints. The modified solver's performance on unsatisfiable instances was comparable to the performance of the original solver. We have also done some experiments with the BerkMin solver [14, 15]; preliminary experiments indicate that BerkMin's performance is similar to Zchaff's.

An unsatisfiable core can be refined by iterating the solver on the core, pruning away additional clauses irrelevant to unsatisfiability. Running 10-20 such iterations can often reduce the core by about 30%. Since subsequent iterations run on smaller CNF files, the overhead of iteration is often insignificant, especially for severely overconstrained models. However, in our preliminary experiments we have found no

significant benefit in additional iterations in terms of what portion of the model was identified as relevant.

3.5 Related work

The problem of detecting when a property is vacuously satisfied by a model has been addressed in the context of temporal model checking [30, 3, 6, 44]. Given a temporal logic formula, these methods produce a "witness" formula that is satisfied if and only if the original formula is vacuously satisfied. Thus, vacuous satisfaction can be detected with an additional model checking run. Several characteristics of these methods prevent them from solving the problem of overconstraint in declarative models. First, overconstraint occurs most often in the definition of the model-checked algorithm rather than in the specification of correctness properties. Published vacuity detection methods may alert the user to the presense of an overconstraint (by showing that the entire correctness property is irrelevant), but cannot pinpoint the location of overconstraint within the model. Second, these methods were described for temporal logic formulas, and either assume a particular form of the formula [3] or require a separate model-checking run to test for irrelevance of each subformula [44]. These limitations preclude published vacuity detection methods from being effective on our problem.

Debugging Equation-Based Models

Peter Bunus [5] has developed a system for debugging numeric systems of equations describing circuits. As with logical models, numeric equations face the dual risks of under- and over-constraint. A crucial difference is that the systems of circuit equations Bunus considers ought to have have exactly one solution. It is therefore easy to determine *if* the system is under- or over-constrained, but it is difficult to narrow down the *source* of the problem. In contrast to logical models, underconstraint turns out to be much harder to debug than overconstraint.

Overconstraint If there are no solutions, the tool determines an unsatisfiable subset – in spirit, this approach is identical to core extraction. In order to detect such a subset, the algorithm operates over a bipartite graph which relates each equation to the variables it mentions. An overconstrained system has no subset constituting a bijection between the variables and equations. Bunus uses combinatorial properties of the graph to prove the absence of such a subset.

Underconstraint If there are multiple solutions, it must be the case that there are fewer non-redundant equations than variables. Redundant equations are fairly easy to detect, so the challenge is to determine what variables can be removed or what equations should be added in order to produce exactly one solution. Bunus employs a technique similar to the one used for overconstraint, but this time the algorithm is searching for (a) extraneous variables which can be removed, and (b) new equations which could be added. The system cannot deduce the new equations directly, but it can provide hints about what they should look like (such as what variables the equation may refer to).

3.6 Conclusions

We have presented *core extraction*, a new analysis that helps discover overconstraint in declarative models. Utilizing the "unsatisfied core" functionality of recent SAT solvers, our tool identifies the set of constraints in a model relevant to preserving a given safety property; the exclusion of seemingly relevant constraints from this set indicates an overconstraint. Our experience has shown that core extraction quickly identifies simple overconstraints that have taken hours to identify previously or that lingered unnoticed for months. Furthermore, we have had some success in applying core extraction to more subtle overconstraints, although work remains to further simplify the debugging process in this case. Core extraction addresses a key deficiency in automatic analysis of declarative models, and may have useful application to other analyses that rely on SAT, such as planning and bounded model checking.

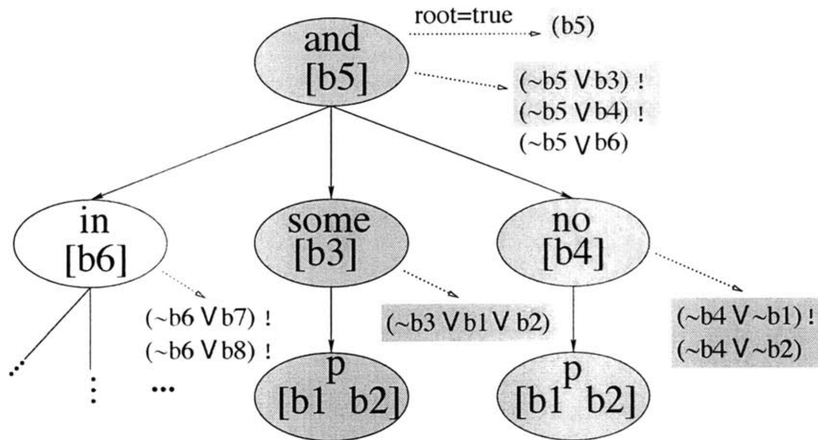


Figure 3-3: *A Roadmap to Core Extraction.* (1) A model is created in any constraint language which is reducible to SAT in a structure preserving fashion. (2) During translation to CNF, each clause generated is annotated with the AST node from which the clause was produced. (3) A SAT solver (used as a black box) determines that the model is unsatisfiable and extracts an unsatisfiable core (a subset of the CNF clauses which is also unsatisfiable). (4) The core is mapped back to the original model by marking (as "relevant") any part of the AST indicated by the annotation of any clause in the CNF core. The analysis is now complete. The remaining steps concern guarantees made to the user about what the markings on the AST mean; they are not actually executed during normal use of the tool. (5) The user is guaranteed that changing the unmarked (non-relevant) portions of the AST will leave the model unsatisfiable. (6) Specifically, the CNF corresponding to the altered AST will be a superset of the unsatisfiable core previously extracted, and thus will itself be unsatisfiable.

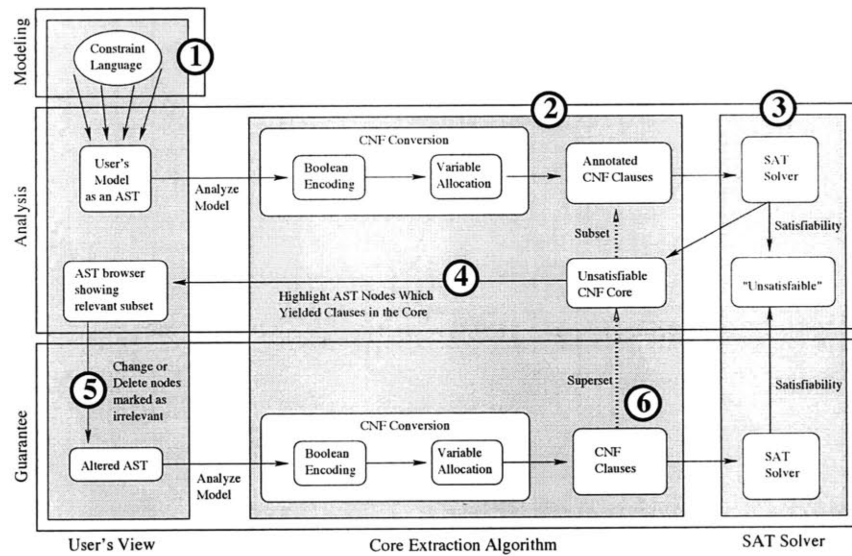


Figure 3-4: Translation of AST to CNF, and mapping back of unsatisfiable core. The AST is for the (trivially unsatisfiable) Alloy formula of the form "(some p) && (no p) && ...". To each node, a sequence of Boolean variables (b1 through b6) is allocated to represent the node's value. From each inner node, translation produces a set of clauses relating the node's Boolean variables to its childrens' Boolean variables. The highlighted clauses form an unsatisfiable core, which is mapped back to the highlighted AST nodes.

Conclusions

We have introduced two algorithms that provide tool support for understanding logical models. Both techniques address problems that Alloy users have brought to our attention – understanding the role of a subformula and detecting unintentional overconstraint. Our initial experience applying the techniques on our own Alloy models suggests that they can be helpful but that they are far from complete solutions to either problem.

Non-example generation helps users understand the role of particular parts of their models. When someone looks at a model for the first time, they often ask questions about the role of (or need for) certain constraints. We formalize the notion and introduce conjunction diagrams as a convenient notation. Our experience from applying the technique on our own models is that it generates useful explanations about why the constraint should be *present*; PCD and PCA were the most helpful entries of the conjunction diagram, PIA was somewhat helpful for double checking one's intuition, and PID was never helpful. However, information about *integrity* was never helpful, and information about *correctness* was only rarely helpful. If these conclusions are backed up by future experience, it may be appropriate to only show the user PCA, PCD, and PIA.

Non-example generation is not currently implemented, although we hope to produce a prototype to explore its role in the modeling process. An implementation would allow us to develop the techniques of target expansion and target refinement, and allow us to rapidly apply the technique to new examples. We have already shown how to avoid several complications of such an implementation. For example, there is no need for a notion of the 'joint role' of several different subformula, even when

dealing with function bodies. We have also shown that the target subformula marked by the user needs to be tracked to the desugared AST, but it does not need to be tracked all the way to the ground DAG or the CNF representation.

Core extraction helps users identify unintentional overconstraint. The danger of overconstraint in logical modeling is well known, and Alloy users have often requested tool support to help overcome it. Our own experience applying core extraction suggests that core extraction is very helpful in the common case of 'dumb' overconstraint bugs, but only mildly helpful in more subtle cases.

Core extraction is currently implemented in the context of Alloy 2.0 and has many rough edges. Edmond Lau, a colleague of the author, is currently adapting that implementation to integrate smoothly with Alloy 3.0 (the most recent version of Alloy) and to have a smoother user interface. The current interface displays a model's core as colored markings on a browse-able AST of the model in which quantifiers have been grounded out. Users would rather see the annotations directly on the original model text, averting the need to expose them to the AST and the notion of grounding out.

There is a significant and growing Alloy user community, giving us the opportunity to get feedback from real modelers about the effectiveness of these tools. Alloy is being taught in about a dozen software courses around the country, providing a us with a wealth of users upon which to test new tools. As the implementations are completed and refined, we will be able to better tailor these tools to users' needs, and document how one can make the best use of these techniques.

Appendix A

Firewire Alloy Model

A complete Alloy model describing part of the Firewire network protocol is given below. This model is described and investigated in Sections 2.5.3 and 3.4.

```
module examples/case_studies/firewire
open util/ordering[State] as ord

abstract sig Msg {}
one sig Req, Ack extends Msg {}

sig Node {to, from: set Link} {
  to = {x: Link | x.target = this}
  from = {x: Link | x.source = this}
  }

sig Link {target, source: Node, reverse: Link} {
  reverse.@source = target
  reverse.@target = source
  }

-- at most one link between a pair of nodes in a given direction
```

```
fact {no disj x,y: Link | x.source = y.source && x.target = y.target}

-- topology is tree-like: acyclic when viewed as an undirected graph
fact Topology {
some tree: Node lone -> Node, root: Node {
  Node in root.*tree
  no ^tree & iden & Node->Node
  tree + ~tree = ~source.target
  }
}

sig Op {}
one sig Init, AssignParent, ReadReqOrAck, Elect, WriteReqOrAck,
ResolveContention, Stutter extends Op {}

sig State {
  part waiting, active, contending, elected: set Node,
  parentLinks: set Link,
  queue: Link -> one Queue,
  op: Op, -- the operation that produced the state
  }

pred SameState (s, s': State) {
  s.waiting = s'.waiting
  s.active = s'.active
  s.contending = s'.contending
  s.elected = s'.elected
  s.parentLinks = s'.parentLinks
  all x: Link | SameQueue (s.queue[x], s'.queue[x])
  }
```

```
pred Trans (s, s': State) {
  s'.op != Init
   s'.op = Stutter => SameState (s, s')
  s'.op = AssignParent => {
    some x: Link {
      x.target in s.waiting & s'.waiting
      NoChangeExceptAt (s, s', x.target)
      ! IsEmptyQueue (s, x)
      s'.parentLinks = s.parentLinks + x
      ReadQueue (s, s', x)
      }}
  s'.op = ReadReqOrAck => {
    s'.parentLinks = s.parentLinks
    some x: Link {
      x.target in s.(active + contending)
       & if PeekQueue (s, x, Ack) then s'.contending else s'.active
      NoChangeExceptAt (s, s', x.target)
      ! IsEmptyQueue (s, x)
      ReadQueue (s', s, x)
      }}
  s'.op = Elect => {
    s'.parentLinks = s.parentLinks
    some n: Node {
      n in s.active
      n in s'.elected
      NoChangeExceptAt (s, s', n)
      n.to in s.parentLinks
      QueuesUnchanged (s, s', Link)
      }}
```

```
  s'.op = WriteReqOrAck => {
    -- note how this requires access to child ptr
    s'.parentLinks = s.parentLinks
    some n: Node {
      n in s.waiting & s'.active
      lone n.to - s.parentLinks
      NoChangeExceptAt (s, s', n)
      all x: n.from |
        let msg = if x.reverse in s.parentLinks then Ack else Req |
          WriteQueue (s, s', x, msg)
      QueuesUnchanged (s, s', Link - n.from)
      }}
  s'.op = ResolveContention => {
    some x: Link {
      let contenders = x.(source + target) {
        contenders in s.contending & s'.active
        NoChangeExceptAt (s, s', contenders)
        }
      s'.parentLinks = s.parentLinks + x
      }
    QueuesUnchanged (s, s', Link)
    }
}

pred NoChangeExceptAt (s, s': State, nodes: set Node) {
  let ns = Node - nodes {
  ns & s.waiting = ns & s'.waiting
  ns & s.active = ns & s'.active
  ns & s.contending = ns & s'.contending
  ns & s.elected = ns & s'.elected}}
```

```
sig Queue {slot: lone Msg, overflow: lone Msg}

pred SameQueue (q, q': Queue) {
    q.slot = q'.slot && q.overflow = q'.overflow
  }

pred ReadQueue (s, s': State, x: Link) {
--   let q = s'.queue[x] | no q.(slot + overflow)
  no s'.queue[x].(slot + overflow)
  all x': Link - x | s'.queue[x'] = s.queue[x']
  }

pred PeekQueue (s: State, x: Link, m: Msg) {
  m = s.queue[x].slot
  }

pred WriteQueue (s, s': State, x: Link, m: Msg) {
        let q = s'.queue[x] |
  no s.queue[x].slot =>
    ( q.slot = m && no q.overflow),
    some q.overflow
  }

pred QueuesUnchanged (s, s': State, xs: set Link) {
  all x: xs | s'.queue[x] = s.queue[x]
  }
```

```
pred IsEmptyQueue (s: State, x: Link) {
  no s.queue[x].(slot + overflow)
--  let q = s.queue[x] | no q.(slot + overflow)
  }

pred Initialization (s: State) {
  s.op = Init
  Node in s.waiting
  no s.parentLinks
  all x: Link | IsEmptyQueue (s, x)
  }

pred Execution () {
  Initialization (ord/first())
  all s: State - ord/last() | let s' = ord/next(s) | Trans (s, s')
  }

pred NoRepeats () {
  Execution ()
  no disj s, s': State | SameState (s, s')
  no s: State | s.op = Stutter
  }

pred NoShortCuts () {
  all s: State |
    ! Trans (s, ord/next(ord/next(s)))
  }
```

```
assert AtMostOneElected {
  Execution () => all s: State | lone s.elected
  }
check AtMostOneElected for 1 Int, 7 Op, 2 Msg,
  3 Node, 6 Link, 3 Queue, 9 State expect 0

assert OneEventuallyElected {
  Execution () => some s: State | some s.elected
  }
check OneEventuallyElected for 1 Int, 7 Op, 2 Msg,
  3 Node, 6 Link, 3 Queue, 9 State expect 1

pred ElectionHappens() {
  Execution ()
  some s: State | some s.elected
  some s: State | no s.elected
}
run ElectionHappens for 1 Int, 7 Op, 2 Msg,
  exactly 3 Node,  6 Link, 3 Queue, 7 State
  expect 1
```

Appendix B

Proofs

In Section 2.2.4, a number of theorems were stated without proof. Those proofs have been relegated to this appendix.

B.1 Deletion and Sabotage are Sufficient to Compute Correctness

Let T be a target subformula of some model M. We have examined the effect of deleting or sabotaging T from M, but one could also imagine asking about making other changes to T. One might ask "what is the role of the *correctness* of this subformula?", meaning "What would have happened if I has written the target differently or even incorrectly?".

Definition:

Let CIA, pronounced "*correctness irrelevant to allowing*", denote the set of all assignments which are solutions to M and which remain solutions regardless of how T is altered. CIA represents solutions which are independent of any change to T. CCA, pronounced "*correctness crucial to allowing*", denotes the set of assignments which are solutions to M which may be disallowed by changes to T. CID and CCD are defined accordingly.

It turns out that we do not actually need to examing every possible mutation to

T in order to compute correctness information. In fact, we only need the 8 cells from the conjunction diagrams for deletion and sabotage.

Theorem 2:

$$CIA = PIA \cap IIA$$
$$CID = PID \cap IID$$
$$CCA = PCA \cup ICA$$
$$CCD = PCD \cup ICD$$

Intuitively, an assignment is only *irrelevant* to correctness if *both* its presence and its integrity are also irrelevant. An assignment is *crucial* to correctness if *either* its presence or its integrity is crucial.

Part 1: $CIA = PIA \cap IIA$

Proof: Let A be some assignment in CIA. If no change to T stops A from being a solution to the model, then the particular changes of setting it to `true` or `false` cannot stop it from being a solution. Thus $CIA \subseteq PIA \cap IIA$.

Let B be some assignment in $PIA \cap IIA$. Suppose $B \notin CIA$; there is some T' such that substituting it for T disallows B. Under the assignment B, T' must evaluate to either `True` or `False`. However, by the definitions of PIA and IIA, setting T to either `True` or `False` leaves B as a solution. Thus the supposition is contradictory, and we can conclude that $PIA \cap IIA \subseteq CIA$.

It follows from $CIA \subseteq PIA \cap IIA$ and $PIA \cap IIA \subseteq CIA$ that $CIA = PIA \cap IIA$.
□

The proof of $CID = PID \cap IID$ follows analogously.

Part 2: $CCA = PCA \cup ICA$

Proof:

Let A be some assignment in $PCA \cup ICA$. If the specific change to T of replacing it with `true` (or the specific change of replacing it with `false`) is enough to stop A from being a solution, then allowing an arbitrary change to T will also disallow A. Thus $PCA \cup ICA \subseteq CCA$.

Let B be some assignment in CCA. Suppose $B \notin PCA \cup ICA$; setting T to either `true` or `false` disallows B. By the definition of CCA, there is some T' such that

replacing T with T' disallows B. However, under the assignment B, the subformula T' evaluates to either `true` or `false`. Thus the supposition is contradictory, and we can conclude that $CCA \subseteq PCA \cup ICA$.

It follows from $PCA \cup ICA \subseteq CCA$ and $CCA \subseteq PCA \cup ICA$ that $CCA = PCA \cup ICA$. □

The proof of $CCD = PCD \cup ICD$ follows analogously.

B.2 Disjoint Entries

Within the conjunction diagram for presence (or within the conjunction diaram for integrity), all the entries are disjoint. We can also make some guarantees about disjointness *between* the conjunction diagram for deletion and that of sabotage.

Theorem: $PCA \cap ICA = \emptyset$

Proof:

By construction, the formula generating $PCA \wedge ICA$ is

$$(\neg(M - T) \wedge M) \wedge (\neg(M \sim T) \wedge M)$$

which simplifies to

$$M \wedge \neg(M - T) \wedge \neg(M \sim T)$$

Suppose there were some solution S satisfying that statement. S would necessarily be a solution to M, and under S, T must evaluate to either `True` or `False`. However, M with T set to a constant is (by definition) the same as either $(M - T)$ or $(M \sim T)$. However, that contradicts the fact that S must violate each of those equations, and thus the original supposition must be false. □

The proof that $PCD \cap ICD = \emptyset$ is analogous.

B.3 Empty Entries

We can make a stronger statement about certain pairs of disjoint entries; not only are they disjoint but one of them is always empty.

Theorem:

$$(PCA = \emptyset) \vee (PCD = \emptyset)$$
$$(ICA = \emptyset) \vee (ICD = \emptyset)$$

Proof: Suppose there were some solution S_{PCA} to PCA and some solution S_{PCD} to PCD.

Without loss of generality, assume that there are no double (or triple etc.) negations, $\wedge$ and $\vee$ are binary (i.e. no triple conjunction), and that conditionals and biconditionals have been desugared into disjunctions and negations.

The proof follows by structural induction. The supposition fails in each of the two base cases – $M = T$ and $M = \neg T$. In the inductive case, T is a proper subformula of M and so there is some Boolen connective at the top of M's AST conntecting two subtrees. One subtree contains (or equals) T and the other does not. In order for the supposition to hold in the inductive case, it must hold in both child trees. By structural induction, it fails for at least one child – the one containing T. The details of the proof follow:

Base cases:

1. $M = T$
2. $M = \neg T$

Inductive cases: If neither base case applies, then there is some top level binary connective combining two subformulae; one contains T and one does not. We will term these formulae F_T and F respectively. M can thus be rewritten as follows:

- $M = F \wedge F_T$

Lemma: F is TRUE under both S_{PCA} and S_{PCD}

Proof: This lemma can be formulated logically as two claims. First we prove that the formula for PCA implies F:

$$
\begin{array}{ll}
(M \wedge \neg(M - T)) \Rightarrow F & \textit{claim} \\
((F \wedge F_T) \wedge \neg(F \wedge (F_T - T))) \Rightarrow F & \textit{rewrite } \mathrm{M} \\
(F \wedge F_T \wedge (\neg F \vee \neg(F_T - T))) \Rightarrow F & \textit{DeMorgan's Law} \\
\neg F \vee \neg F_T \vee (F \wedge (F_T - T)) \vee F & \textit{desugar conditional}
\end{array}
$$

The final statement contains $F \vee \neg F$ and is thus tautologically true. It is equivalent to the first statement, proving our first claim.

The second claim we need to prove is that the formula for PCD implies F:

$$
\begin{array}{ll}
(\neg M \wedge (M - T)) \Rightarrow F & \textit{claim} \\
(\neg(F \wedge F_T) \wedge (F \wedge (F_T - T))) \Rightarrow F & \textit{rewrite } \mathrm{M} \\
((\neg F \vee \neg F_T) \wedge F \wedge (F_T - T)) \Rightarrow F & \textit{DeMorgan's Law} \\
(F \wedge F_T) \vee \neg F \vee \neg(F_T - T) \vee F & \textit{desugar conditional}
\end{array}
$$

The final statement contains $F \vee \neg F$ and is thus tautologically true. It is equivalent to the first statement, proving our second claim.
Since F is TRUE in both S_{PCD} and S_{PCA}, and since $M = F \wedge F_T$, we can reduce our problem to examining $M' = F_T$. M' will either match one of the base cases or one of the inductive cases.

(2) $M = F \wedge F_T$

Lemma: F is FALSE in both S_{PCA} and S_{PCD}

Proof: This lemma can be formulated logically as two claims. The first is that the formula for PCA implies $\neg F$.

$$
\begin{array}{ll}
(M \wedge \neg(M - T)) \Rightarrow \neg F & \textit{claim} \\
((F \vee F_T) \wedge \neg(F \vee (F_T - T))) \Rightarrow \neg F & \textit{rewrite } \mathrm{M} \\
((F \vee F_T) \wedge \neg F \wedge \neg(F_T - T)) \Rightarrow \neg F & \textit{DeMorgan's Law} \\
(\neg F \wedge \neg F_T) \vee F \vee (F_T - T) \vee \neg F & \textit{desugar conditional}
\end{array}
$$

The final statement contains $F \vee \neg F$ and is thus tautologically true. It is equivalent to the first statement, proving our first claim.

The second claim we will show is that the formula for PCD implies $\neg F$.

$$
\begin{array}{ll}
(\neg M \wedge (M - T)) \Rightarrow \neg F & \textit{claim} \\
(\neg(F \vee F_T) \wedge (F \vee (F_T - T))) \Rightarrow \neg F & \textit{rewrite}\ \mathrm{M} \\
(\neg F \wedge \neg F_T \wedge (F \vee (F_T - T))) \Rightarrow \neg F & \textit{DeMogan's Law} \\
F \vee F_T \vee (\neg F \wedge \neg(F_T - T)) \vee \neg F & \textit{desugar conditional}
\end{array}
$$

The final statement contains $F \vee \neg F$ and is thus tautologically true. It is equivalent to the first statement, proving our second claim.

Since F is FALSE in both S_{PCD} and S_{PCD}, and since $M = F \vee F_T$, we can reduce our problem to examining $M' = F_T$. M' will either match one of the base cases or one of the inductive cases.

Both inductive cases reduce to strictly simpler cases, and both base cases satisfy the theorem. □

Interpretation The interpretation of this lemma is that each portion of a propositional logic model either strictly relaxes the model or strictly restricts it. The set of solutions to $M - T$ is either a subset of a superset of the solutions to M.

Bibliography

[1] Flavio Lerda Alex Groce, Daniel Kroening. Understanding counterexamples with explain. In Wizard V. Oz and Mihalis Yannakakis, editors, *Proc. 16th International Conference on Computer Aided Verification (CAV)*, number 16, pages 453–456, Boston, July 2004. Springer. LNCS 3114.

[2] Sarfraz Khurshid Darko Marinov Alexandr Andoni, Dumitru Daniliuc. Evaluating the small scope hypothesis. September.

[3] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Design Automation Conference*, 1999.

[5] Peter Bunus. *Debugging and Structural Analsis of Declarative Equation-Based Languages.* Licentiate engineering dissertation, Linkopings Universitet, School of Engineering, 2002. Thesis No. 964.

[6] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for temporal logic model checking. *Lecture Notes in Computer Science*, 2031:528–??, 2001.

[7] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, 1999.

[8] Ilya Shlyakhter Daniel Jackson, Ian Schechter. Alcoa: the alloy constraint analyzer, 2000.

[9] Mandan Vaziri Daniel Jackson. Finding bugs with a constraint solver. *International Symposium on Software Testing and Analysis (ISSTA)*, 2000.

[10] Flavio Lerda Daniel Kroening, E. Clake. A tool for checking ansi-c programs. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.

[11] Alan J. Hu David L. Dill, Andreas J. Drexler and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[12] Kit Fine. Vagueness, truth and logic. Philosophical Writings, 1975. p. 452.

[13] Melvin Fitting. Herbrand's theorem for a modal logic. 1996.

[14] Eugene Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proceedings of Design, Automation and Test in Europe (DATE2003)*, Munich, Germany, March 2003.

[15] Evgueni Goldberg and Yakov Novikov. Berkmin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, March 2002.

[16] G. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[17] William E. Howden. Completeness criteria for testing elementary program functions. In *Proc. International Conference on Software Engineering (ICSE)*, pages 235 – 243, 1981.

[18] IEEE. *IEEE Standard for a High Performance Serial Bus, Standard 1394-1995*. IEEE, Aug 1996.

[19] Daniel Jackson Manu Sridharan Mana Taghdiri Ilya Shlyakhter, Robert Seater. Debugging overconstrained declarative models using unsatisfiable cores. *18th IEEE International Conference on Automated Software Engineering (ASE)*, October.

[20] Dietmar Seipel Jack Minker. Disjunctive logic programming: A survey and assessment, in honor of robert a. kowalski. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, pages 472–511, 2002. ISBN:3-540-43959-5.

[21] Daniel Jackson. Automating first-order relational logic. *Foundations of Software Engineering*, November.

[22] Daniel Jackson. Automating first-order relational logic. In *Proceedings ACM SIGSOFT Conference on Foundations of Software Engineering*, San Diego, November 2000.

[23] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2001.

[24] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[25] R. M. Hierons K. Adamopoulos, M. Harman. Mutation testing using genetic algorithms: A co-evolution approach. *AAAI Genetic and Evolutionary Computation Conference (GECCO)*, 2004.

[26] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, 1992.

[27] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.

[28] Adam Kolawa. Mutation testing: A new approach to automatic error-detection. *STAREAST*, 1999. conference organized by Software Quality Engineering.

[29] M. Kulas. Debugging prolog using annotations. In M. Ducasse, A. Kusalik, and G. Puebla, editors, *Proc. of the 10th Workshop on Logic Programming Environments (WLPE'99), Las Cruces, NM*, volume 30, issue 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. http://www.elsevier.nl/locate/entcs/volume30.html.

[30] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 82–96, 1999.

[31] David Lewis. Causation. *Journal of Philosophy*, 70:556–356, 1973.

[32] S. Danicic M. Harman, R. M. Hierons. The relationship between program dependence and mutation testing. In *Proc. Mutation*, pages 15 – 23, October.

[33] Dietmar Seipel Marbod Hopfner. Reasoning about rules in deductive databases. In *Proc. 17th Workshop on Logic Programming (WLP)*, 2002.

[34] John Stuart Mill. Five cannons of induction. Philosophical Writings, 1843. lived 1806-1873.

[35] Suvo Mittra. Iolus: A framework for scalable secure multicasting. In *Proceedings ACM SIGCOMM'97*, pages 277 – 288, Cannes, September 1997.

[36] Simin Nadjim-Tehrani. Debugging prolog programs declaratively. In M. Bruynooghe, editor, *Proc. 2nd Workshop on Meta-Programming in Logic*, pages 137–155. Springer, 1990. LNCS 3114.

[37] S. Danicic R.M. Hierons, M. Harman. Using program slicing to assist in the detection of equivalent mutants. In *Proc. The Journal of Software Testing, Verification, and Reliability*, pages 233 - 262, 1999.

[38] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[39] Ilya Shlyakhter, Manu Sridharan, Robert Seater, and Daniel Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. http://ilya.cc/sharing.ps, May 2003.

[40] Avi Sion. *Causal Logic: Definition, Deduction and Induction of Causation, Volition and Allied Cause-Effect Relations*, chapter 1.1. Avi Scion, Geneva, Switzerland, second edition, 2003. TheLogician.net.

[41] J. Michael Spivey. *The Z Notation: A Reference Manual, 2nd ed.* Prentice-Hall, 1992.

[42] Mana Taghdiri. Lightweight modelling and automatic analysis of multicast key management schemes. Master's thesis, Massachusetts Institute of Technology, 2002.

[43] Richard J. Lipton Frederick G. Sayward Timothy A. Budd, Richard A. DeMillo. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220 – 233, Las Vegas, Nevada, 1980. ACM Press, NY, NY, USA. ISBN:0-89791-011-7.

[44] M. Vardi, R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, and A. Tiemeyer. Enhanced vacuity detection in linear temporal logic. In *Proceeding of International Conference on Computer-Aided Verification (CAV'03)*, 2003.

[45] Patrick Henry Winston. *Learning Structural Descriptions from Examples.* PhD thesis, Massachusetts Institute of Technology, 1970.

[46] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, Reading, Massachusetts, third edition, 1992. pp. 150-356.

[47] Lintao Zhang and Sharad Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of Design, Automation and Test in Europe (DATE2003)*, Munich, Germany, March 2003.

3506-22