

Modeling by Example

by

Lucy Mendel

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

August 11, 2007

Certified by

Daniel N. Jackson

Professor

Thesis Supervisor

Accepted by

Arthur C. Smith

Chairman, Department Committee on Graduate Students

Modeling by Example

by

Lucy Mendel

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Software developers use modeling to explore design alternatives before investing in the higher costs of building the full system. Unlike constructing specific examples, constructing general models is challenging and error-prone. Modeling By Example (MBE) is a new tool designed to help programmers construct general models faster and without errors. Given an object model and an acceptable, or included, example, MBE generates near-hit and near-miss examples for the user to mark as included or not by their mental goal model. The marked examples form a training data-set from which MBE constructs the user's general model. By generating examples dynamically to direct its own learning, MBE learns the concrete goal model with a significantly smaller training data set size than conventional instance-based learning techniques. Empirical experiments show that MBE is a practical solution for constructing simple structural models, but even with a number of optimizations to improve performance does not scale to learning complex models.

Thesis Supervisor: Daniel N. Jackson

Title: Professor

Acknowledgments

I thank my advisor, Prof. Daniel Jackson, for supervising my research at CSAIL in the past two years, and for introducing me to the fascinating area of formal methods. I thank Derek Rayside for being an excellent mentor in this and previous research.

Contents

1	Introduction	13
1.1	Pictorial vs Textual	14
1.2	Specific vs General	15
1.3	Contribution	17
1.4	Overview	18
1.5	Technical Challenges	18
1.6	Inspiration	20
2	Overview	21
2.1	Direct File System Model Construction	23
2.1.1	Object Model	23
2.1.2	Constraints	23
2.2	MBE File System Model Construction	26
2.2.1	Interactive Generalization	27
2.3	Equivalence of User–Constructed and MBE–Generated Textual Models	33
3	Learning	35
3.1	Definitions	35
3.2	Learning Algorithm	39
3.2.1	Model Extraction	40
3.2.2	Model Generalization	44
3.2.3	Termination	51
3.3	Rationale	53

3.3.1	Prototypical Example	54
3.3.2	Grammar	55
3.4	Refinements	56
3.4.1	Generating a Prototypical Example	57
3.4.2	Multiple Prototypes	58
3.4.3	Learning Implications	59
4	Evaluation	65
4.1	Singly Linked List	65
4.2	Doubly Linked List (good prototypical example)	68
4.3	Trivial File System	72
4.4	Medium Complexity File System	78
4.5	Full Complexity File System	79
5	Related Work	89
5.1	Examples and learning	89
5.2	Prototypical concept description	90
5.3	Dynamic invariant detection	91
5.4	Mutation testing	92
6	Conclusion and Future Work	95
6.1	Future Work	95
A	Predicate Library	99

List of Figures

1-1	Gustave Eiffel's blueprint [14] for the Eiffel Tower	16
2-1	Information flow between MBE and the user in order for MBE to learn the user's goal model	22
2-2	File system object model	24
2-3	File system constraints (incorrect model)	24
2-4	File system constraints (correct)	26
2-5	Prototypical example for file system model	26
2-6	MBE's generated textual model for file system. Alloy declarations for the mathematical predicates are given in table A.1	34
3-1	(a) Object model and (b) prototypical example for a simple tree structure . .	36
3-2	Object model for modeling a file system	37
3-3	Metaphorical example space for tree model	38
3-4	Phase 1 of the learning algorithm	39
3-5	Phase 2 of the learning algorithm	40
3-6	The grammar of type and relation expressions, where <code>type_name</code> and <code>relation_name</code> are types and relations defined by some object model . .	41
3-7	Predefined predicates for restricted file system example	42
3-8	Generated constrains for file system example with three predefined predicates	43
3-9	Mutated models from the initial file system model extracted in section 3.2.1	46
3-10	Object model for file system example containing two kinds of file system objects, <code>File</code> and <code>Dir</code>	58

4-1	Singly linked list object model	66
4-2	Singly linked list prototypical example	67
4-3	Initial model extracted from singly linked list prototype (figure 4-2	68
4-4	Singly Linked List goal model constructed by MBE	69
4-5	Doubly linked list object model	70
4-6	Good doubly linked list prototypical example	71
4-7	Initial model extracted from ceiling and floor prototype (figure 4-6	73
4-8	Ceiling and floor goal model constructed by MBE	73
4-9	Two reduced ceiling and floor goal models	74
4-10	Trivial file system object model	74
4-11	Trivial file system prototypical example	75
4-12	Better trivial file system prototypical example	75
4-13	Initial model extracted from trivial file system prototype (figure 4-11)	77
4-14	Trivial file system goal model constructed by MBE	77
4-15	Medium complexity file system goal model constructed by MBE	78
4-16	Complex file system object model	81
4-17	Complex file system prototypical example	81
4-18	Fully complex file system initial model constructed by MBE	86
4-19	Possible goal model for fully complex file system	87
A-1	Alloy code to check whether acyclic implies reflexive or irreflexive	103
A-2	Graph of implications between predicates	103

List of Tables

1.1	Comparison of architectural modeling with software modeling and visualization	17
1.2	Comparison of models constructed by MBE. MBE interacts with users via specific examples in order to construct the general model it outputs.	17
2.1	Examples generated by MBE in the first round of interaction	28
2.2	Examples generated by MBE in the second round of interaction	31
2.3	Examples generated by MBE in the third round of interaction	33
3.1	If M_{c_jUG} generates an example, E , then E is included or excluded depending on whether G_i and c_j contain essential constraints or overconstraints	50
3.2	Comparison of the number of constraints extracted into the initial model for different prototypical examples. 197 constraints were generated overall by applying 16 predicates to the file system object model	60
3.3	Constraints and the marked examples they generated	61
4.1	Summary of MBE's learning time on different models	65
4.2	Summary of MBE's learning time on different models	66
4.3	Instances generated by MBE in the first round of interaction	67
4.4	Instances generated by MBE in the first round of interaction	71
4.5	Instances generated by MBE in the first round of interaction	75
4.6	Instances generated by MBE in the second round of interaction	77
4.7	Instances generated by MBE in the first round of interaction	82
4.8	Instances generated by MBE in the second round of interaction	83

4.9	Instances generated by MBE in the third round of interaction	85
A.1	Predicates from Alloy's graph and relation utilities modules	100

Chapter 1

Introduction

Modeling, or using representations of a system's essential features to communicate and investigate properties of the full system, is a useful and widespread practice in science and engineering. Architects use models to communicate building designs to clients; particle physicists and geologists use models to examine physical reality without expensive or dangerous real-world experimentation; and molecular chemists use models to mimic the behavior of molecules in a controlled environment.

Software developers also use modeling to explore design alternatives before investing in the higher costs of building the full system. Software developers, architects, scientists and engineers all require that models be precise, or accessible to deep and meaningful analysis. A model that is ambiguous or unclear on any essential component of the real-world system it represents causes confusion and error. Equally important is reducing the cost of model construction, measured in terms of development time and model correctness.

A model's preciseness and cost are largely influenced by its representation. Improving a model's precision may mean using a modeling language that is easily but precisely parsed by mechanical analysis. Improving a model's development time and correctness is a more open-ended challenge. One primary goal is to help the programmer not make errors while constructing the model. In this case, the most influential feature of a model's representation is whether the model is specific or general, and to a lesser extent whether the language is pictorial or textual.

1.1 Pictorial vs Textual

The models of scientists and engineers are often pictorial; e.g., schematics, blueprints, maps and graphical simulations. In the book, “Engineering in the Mind’s Eye,” Eugene Ferguson argues that visual thinking not only enriches engineering, but is crucial to its success [18]. Well known tools such as Matlab and SolidWorks provide textual interfaces that aid in constructing pictorial simulations, graphs and drafts. Active Statics is one example of a tool that provides an interactive pictorial interface to structural designs [1]. It is based on graphic statics, a body of precise, pictorial techniques used by such masters as Antonio Gaudi, Gustave Eiffel, and Robert Maillart to create their structural masterpieces. Pictorial models capture the essential properties of systems compactly and precisely, making them well suited for analysis, as well as straightforward to construct and read. Similarly, scientists use pictures, graphs and small examples in research papers so that readers can quickly identify the context of technical discussion.

Existing software modeling tools primarily use specialized textual notations (e.g., Alloy [23], SMV [7], Spin [22], VDM, Larch and Z). A common problem with textual interfaces is that the text must be translated into the developer’s mental model. The textual difference between what the developer meant to write and what they actually wrote may be small, which exacerbates they provide insufficient feedback are inconvenient for detecting and debugging errors. A particularly well known and significant problem is checking that nothing bad happens in an overconstrained model. For example, we might be relieved to know that a proton therapy machine model never overdoses patients, only to find out later that an error in the model prevented the modeled machine from giving doses of any amount.

Overconstrained models mask errors, but detecting and debugging overconstraints is difficult because subtle differences in text can have surprising effects, and people commonly forget to consider tricky corner cases. For example, what happens when two universally quantified variables, a and b , refer to the same element? Corner cases such as these are difficult even for expert software modelers. Forgetting to handle the case where two universally quantified elements were equivalent resulted in a bug in Alloy’s [23] graph

utility module that prevented models of singleton connected graphs.¹

Attempts to create diagrammatic programming tools have resulted in pictures that are either complex, and thus difficult to construct, or simplistic, and thus uninteresting to analyze (e.g., UML). According to Fred Brooks in *No Silver Bullet*, “Whether we diagram control flow, variable scope nesting, variable cross-references, data flow, hierarchical data structures, or whatever, we feel only one dimension of the intricately interlocked software elephant” [19].

1.2 Specific vs General

The difficulty in modeling software systems results from a fundamental difference between the models of software developers and architects. Software developers build systems that specify many possible executions, and thus software models themselves must specify many possible *examples*, or particular configurations of system states. A *general model* declares which examples are included by the model and which are excluded. A general model for star network topology is “all nodes, called spokes, are connected to a central node, called a hub, and all communication between spokes goes through the hub.” Three computers connected to a router is an example of a star network topology, whereas three computers connected to each other is not.

Architects, on the other hand, primarily construct *specific models*, or a single example of a potential goal building. Indeed, an architect constructs multiple examples to explore alternate designs for a single goal building, not to generalize a single design for many possible buildings.

Constructing precise representations of specific examples is straightforward, whereas constructing precise representations of general models is difficult. For example, Gustave Eiffel’s blueprint of the Eiffel Tower specifies the precise dimensions of each crossbar

¹This bug was present but unidentified since at least 2004 when Alloy3 was released. The bug appeared in two predicates, `weaklyConnected` and `stronglyConnected`, in a graph module provided with Alloy. Why was the bug not found earlier? Perhaps users wrote their own connected predicates instead of using the provided utility modules. Perhaps `weaklyConnected` and `stronglyConnected` were unnecessary for the models being written, or singleton graphs, which the bug erroneously excludes, were excluded by other constraints.

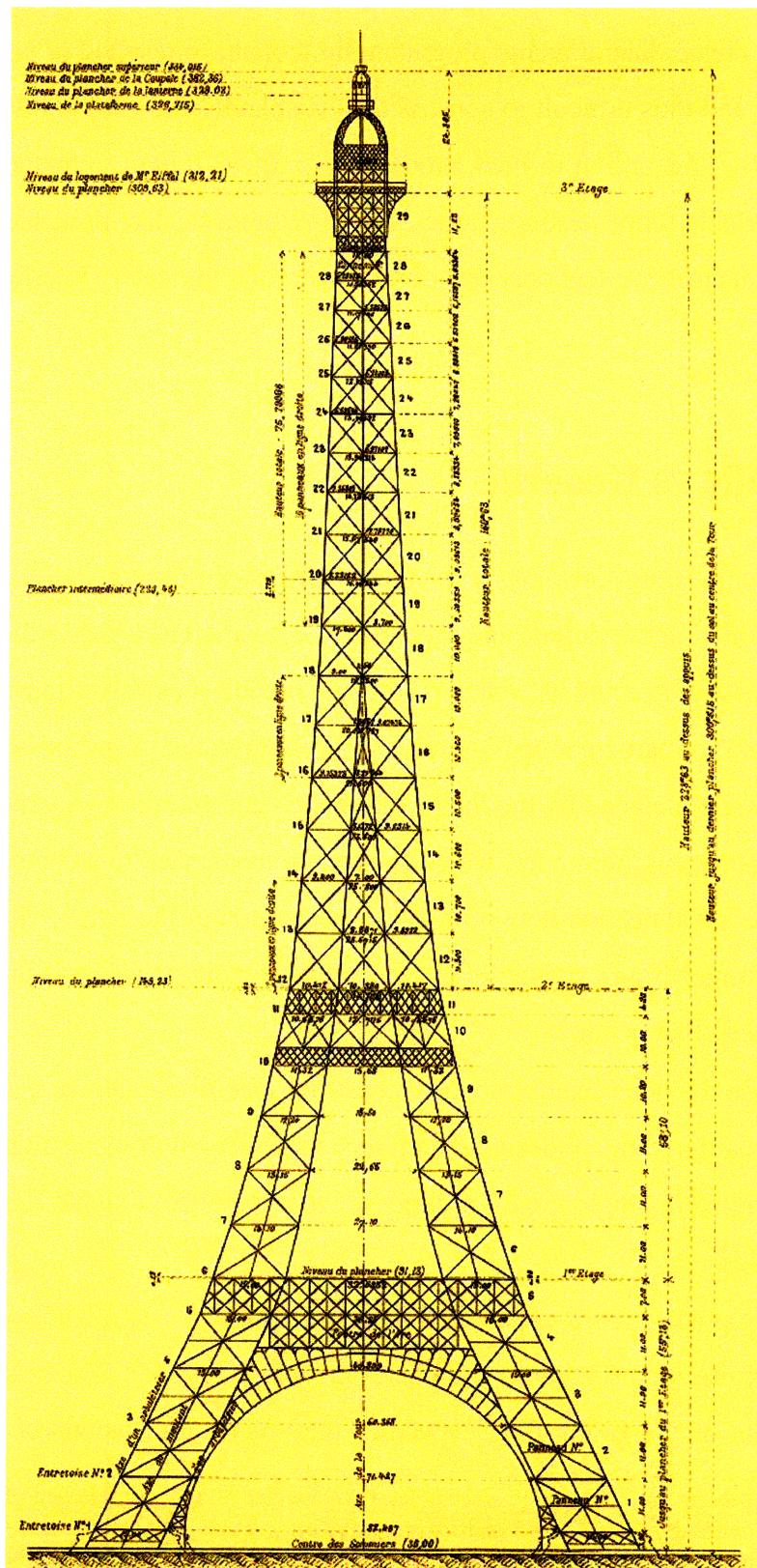


Figure 1-1: Gustave Eiffel's blueprint [14] for the Eiffel Tower

	Specific Example	General Model
Precise	Architect's model	Programmer's linguistic model
Ambiguous		Programmer's pictorial model

Table 1.1: Comparison of architectural modeling with software modeling and visualization

	Specific Example	General Model
Textual		Returned by MBE
Graphical	Constructed by MBE for users to mark	

Table 1.2: Comparison of models constructed by MBE. MBE interacts with users via specific examples in order to construct the general model it outputs.

(figure 1-1). What would be the blueprint from which we could construct not only the Eiffel Tower but also the Empire State Building, Sears Tower, Petronas Towers and Taipei 101? One straightforward representation is to reproduce 4 sub-blueprints, each a blueprint for a different tower, within the single general blueprint. Unfortunately, this representation is undesirable, if not infeasible, for models that include numerous, or even infinite, solutions.

Information in a general model is compressed compared to specific models. Rather than enumerating all of the included examples, a general model must contain abstractions that identify key features shared among included examples. Because general models contain abstractions, they are more complex than specific examples. Consequently, constructing error-free general models is more challenging than constructing specific examples. Developing pictorial models that are not only general but also precise is extremely challenging.

1.3 Contribution

Programmers want a less error-prone and less time consuming interface than text for constructing precise, general models. Pictorial models can capture essential properties precisely and compactly for specific models, but the approach does not seem to work for the general models required in software modeling (table 1.1). We developed a technique, Modeling By Example (MBE), that separates the construction of precise, textual generalizations from the user interface (table 1.2). Users of MBE interactively discriminate between pictorial examples as included or excluded by their mental model, from which MBE mechanically constructs a precise, textual generalization that is equivalent to the user's goal.

1.4 Overview

Research on human learning has shown that examples are a critical component of learning new skills [33] [32] [44] [26]. People not only prefer to learn from concrete examples over models when given a choice [34] [26], but also learn faster and with more comprehension because examples are easier to keep in working-memory and more motivating than abstract representations [6] [33] [44].

Examples play two key roles in understanding and learning a model: *prototyping* and *generalizing*. In the first role, a model is represented by a prototype, or ideal example, against which new examples are compared to determine their inclusion or exclusion by the model. Examples that are similar to the prototype are included by the model; examples that are different are excluded. Representing a model by a prototype was first introduced in Marvin Minsky’s seminal paper on knowledge representation and “frames” [28].

In the second role, near-miss and near-hit examples define the model’s boundary. A *near-hit* is an included example that would be excluded if changed slightly; it lies just inside the model’s boundary. A *near-miss* is an excluded model that would be included if changed slightly; it lies just outside the model’s boundary. Near-miss learning was introduced in Patrick Winston’s classic book on artificial intelligence [43], and example-based learning in general is a broad and well-researched area of machine learning with growing successes.

1.5 Technical Challenges

In designing MBE we faced the following three technical challenges: obtaining a complete set of examples; generalizing examples into a general model; and scaling the algorithm to complex models.

A set of examples is complete with respect to a particular model if MBE can learn a single model from the set. An incomplete set of examples specifies multiple models. MBE must learn a particular model, and thus every “aspect” of the goal model’s boundary must be precisely determined. A boundary aspect corresponds to the small portion of a model

from which classes of near-hits or near-misses are generated. MBE determines, or learns, a boundary aspect from a near-hit or near-miss generated by changing only that slight portion of the model. Thus, MBE must generate at least one example per boundary aspect to guarantee learning a single model.

Because MBE does not know the desired model, the examples it generates may be near-misses or near-hits. MBE learns from the example only after the user has marked that the desired model includes or excludes the example (near-hit and near-miss, respectively). Thus, the challenge is in generating the most discriminatory examples to minimize the number of examples viewed by the user. MBE uses repeated rounds of user interaction to guide its generation of real corner cases and avoidance of clear-hits and clear-misses.

Translating multiple examples into a single general model is challenging. Although listing all included examples defines the exact boundary of the model, we require a generalization that is more compact. To translate an example into a model we extract a general representation, or model, using the algorithm explained in section 3.2.1. The extracted representation is useful if it includes the original example, as well as excludes all significantly different examples. That is, the model should preserve essential properties of the original example and exclude examples that lack these properties. Otherwise, MBE will learn little from extracted models. Determining “significant differences” and “essential properties” relies on assumptions that are difficult to obtain early in the learning process.

The model extracted from a single example is likely overconstrained; thus, the model MBE returns is learned from many examples. An **underconstrained** model includes undesirable examples; an **overconstrained** model excludes desirable examples. Avoiding overconstrained and underconstrained models is a classic problem in example-based learning, and relies on obtaining a training data-set that sufficiently explores the state space. That is, the learning algorithm requires a complete set of examples.

Unsurprisingly, the usefulness of MBE with respect to running time and comprehensibility is inversely related to the range of models it can express. Increasing the complexity of expressible models decreases the tool’s efficiency. Thus, we make a trade-off that maximize the range of practical models we can express, while minimizing the language and grammar complexity used to describe the models. Since MBE must generate a complete

set of near-hits and near-misses, the algorithm is at least linear in the size of the generalization's boundary. Scalability is therefore a primary concern.

Our design chooses optimizations that increase the running time while still guaranteeing that the correct model will be found if it is expressible. One particularly effective optimization is to dynamically generate constraints as needed. For example, MBE initially generates constraints by applying properties to root types in the object model; only when some constraint, c_t , is not held by an included example are constraints created by applying the property to subtypes of t . By learning fine grained constraints only when necessary, learning is faster in practice without sacrificing the models that can be learned.

1.6 Inspiration

Using specific examples in place of conventional textual or general interfaces is not new. Prototype-based programming languages such as Self [37] and Subtext [12] provide an interface in which object examples are copied with small modification from prototypical objects. Subtext provides a radical WYSIWYG-like programming interface in which the program code, which consists of a tree of examples, is constantly being executed [12]. Like MBE, Subtext separates the tool's representation from the interface. Edwards's work on example centric programming provides automated IDE support for the use of examples in programming [13].

MBE's learning algorithm, and particularly the generalization phase, is inspired by Winston's work on near-misses and Seater's work on non-example generation [36] [42] [43]. Seater uses near-miss and near-hit examples to explain the role of a constraint with respect to the solution set for a particular model, whereas MBE learns the desired model's boundary by using near-miss and near-hit examples to classify which constraints are essential or overconstraints.

Chapter 2

Overview

We built Modeling by Example (MBE) to help programmers construct their intended models faster and with more confidence. Figure 2-1 summarizes how MBE learns a goal model. To begin the learning process, the user provides an object model, which is a declaration of the structure of examples included by a model (e.g., types and relations), and a prototypical example, which is a user-generated example included by the model (phase I). Next, MBE generates examples and displays them to the user. The user marks each example as included or excluded by their goal model (phase II). MBE refines its internal working model based on what it learns from the user-marked examples. Iterations of model refinement and user interaction continue until MBE terminates the process and returns the learned model (phase III).

MBE is built on top of Alloy [23], a lightweight model finding tool. The Alloy Analyzer takes a model, in this case a logical formula, written in the Alloy Language and attempts to find a binding of variables to values that makes the formula true. Using Alloy one can mechanically verify complex structural constraints and behavior.

The rest of this chapter demonstrates two approaches to model construction, first using a conventional modeling tool to construct the model directly (section 2.1), and then using MBE to construct the model indirectly (section 2.2). Both demonstrations construct the same file system model, specified below.

In the example file system model, there is one kind of file system object, called `Dir`, which corresponds to a directory. Directories contain other directories via `DirEntry`

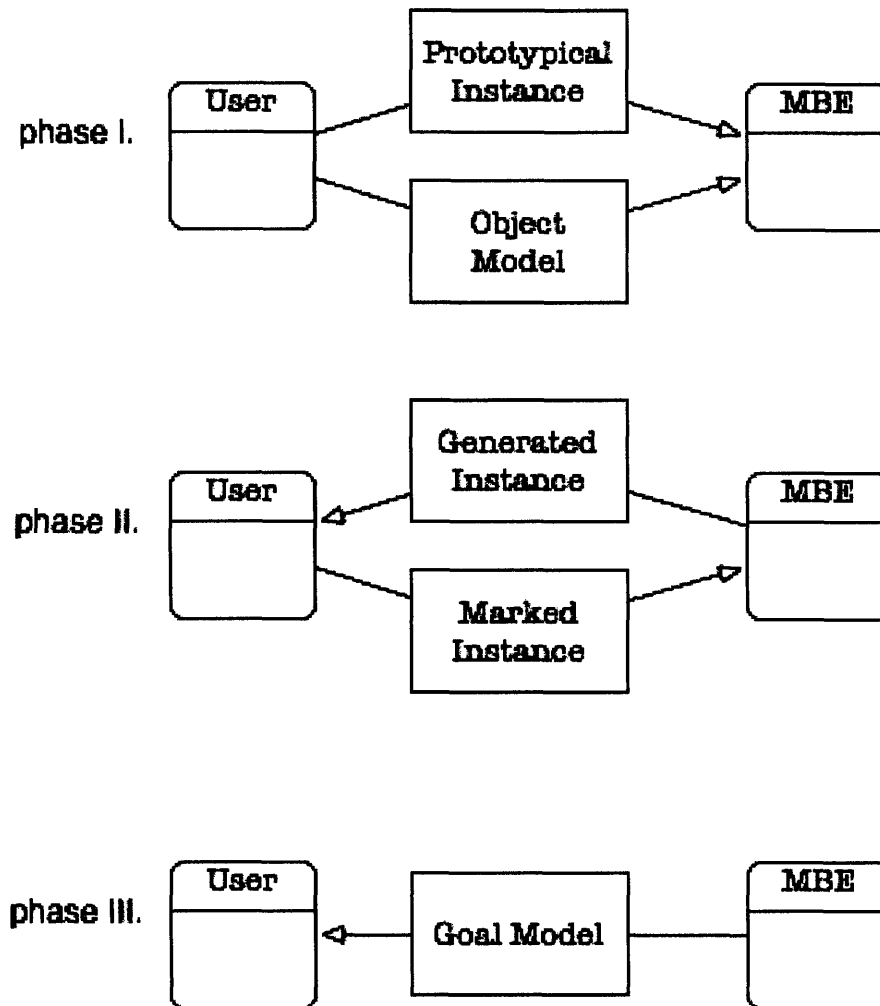


Figure 2-1: Information flow between MBE and the user in order for MBE to learn the user's goal model. To begin the learning process, the user provides an object model, which is a declaration of the structure of examples included by a model (e.g., types and relations), and a prototypical example, which is a user-generated example included by the model (phase I). Next, MBE generates examples and displays them to the user. The user marks each example as included or excluded by their goal model (phase II). MBE refines its internal working model based on what it learns from the user-marked examples. Iterations of model refinement and user interaction continue until MBE terminates the process and returns the learned model (phase III)

objects. The `entries` relation maps each `Dir` to zero or more `DirEntry`. Each `DirEntry` has one `name` of type `Name`, and one `content` of type `Dir`. The `entries` and `contents` relations are constrained to conform to a tree data structure, with a single root directory and no cycles. However, directories can be aliased; thus, a `Dir` object might be contained by multiple `DirEntry`. Finally, `Name` objects describe local names. The full path to some `Dir`, d , is formed by concatenating the names of `DirEntry` between the root directory and d .

2.1 Direct File System Model Construction

To construct the file system model directly, the user specifies, in this case in Alloy, an object model and constraints.

2.1.1 Object Model

An object model is a declaration of the types and relations that may exist in examples included by a model. The file system model contains three types: `Dir`; `DirEntry`; and `Name`; and three relations: `entries`; `contents`; and `names`.

`Dir` (directory) objects contain `DirEntry` (directory entry) objects via the `entries` relation. Each `DirEntry` object contains one `Name` object and one `Dir` object via the `name` and `contents` relations, respectively (figure 2-2).

2.1.2 Constraints

Next, the user constrains the file system model (figure 2-3).

Coming up with these four file system constraints is not trivial. In fact, the model is incorrect in two ways.

First, only examples with a single directory satisfy the model; thus, the model is over-constrained. We detect this bug by generating examples from the model (using Alloy) and observing that no examples with more than one `Dir` appear. To verify this observation we can add an additional constraint to the model so that it only includes examples with more

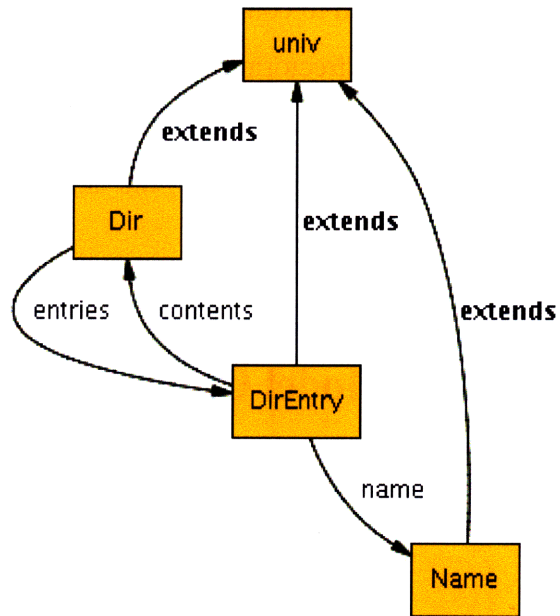


Figure 2-2: File system object model

```

-- no directory contains itself
all d: Dir | d !in d.^(entries.contents)

-- rooted
one root: Dir | Dir in root.*(entries.contents)

-- directories with the same parent have different names
all de1,de2: DirEntry |
  de1.^entries = de2.^entries => de1.name !=de2.name

-- directory entries are unique
all de: DirEntry | one de.^ entries

```

Figure 2-3: File system constraints (incorrect model)

than one `Dir`. Alloy then tells us that no examples satisfy the model.

Determining the cause of the bug is less straightforward. The third constraint excludes all examples with non-empty `entries` relations because the implication fails when `de1` and `de2` reference the same element. To correct this bug we add an equality check to the implication

```
-- directories with the same parent have different names
all de1,de2: DirEntry |
  de1 != de2 and de1.~entries = de2.~entries =>
  de1.name !=de2.name
```

Second, examples with `DirEntry` containing multiple `Dir` are falsely included. Because the first bug caused the model to be overconstrained, it masked the presence of the second bug; the model is underconstrained because it includes undesirable examples.

Like last time we use example generation and additional constraints to detect the presence of this bug. We determine the presence of this bug by generating examples included by the model and observing undesirable examples. We could also add constraints to the model so that it only includes these undesirable examples. We would then check that no example satisfies the extended model; since some would, we would know the model is underconstrained.

To correct the model we add constraints that exclude the undesirable examples

```
-- names and contents are functional
all de: DirEntry | one de.name and one de.contents
```

Although identifying and fixing the bugs in the file system model is simple, the variety and complexity of bugs increases as the number of interacting model elements increases. Bugs are identified using a test suite of mutated models. Models that are expected to generate some examples identify overconstraint bugs, while models that are expected to generate no examples identify underconstrained bugs. Fixing bugs, on the other hand, is frustrating because the user already knows which examples he wants the model to include and excluded, but finding the offending constraints and determining the corrections requires detective work and insight.

```

-- no directory contains itself
all d: Dir | d !in d.^(entries.contents)

-- rooted
one root: Dir | Dir in root.*(entries.contents)

-- directories with the same parent have different names
all de1,de2: DirEntry |
  de1 != de2 and de1.~entries = de2.~entries =>
  de1.name !=de2.name

-- directory entries are unique
all de: DirEntry | one de.~ entries

-- names and contents are functional
all de: DirEntry | one de.name and one de.contents

```

Figure 2-4: File system constraints (correct)

2.2 MBE File System Model Construction

To construct a file system model using MBE the user provides the same object model as in the conventional modeling demonstration (figure 2-2), as well as a prototypical, or standard, example of a file system (figure 2-5).

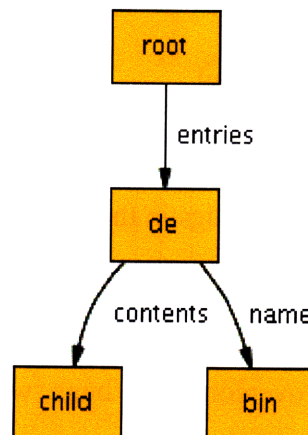


Figure 2-5: Prototypical example for file system model

2.2.1 Interactive Generalization

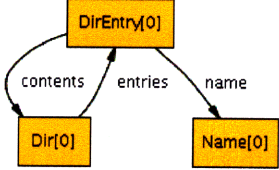
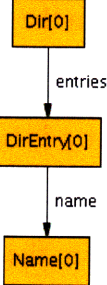
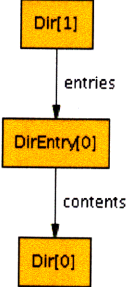

MBE uses the object model and prototype to generate interesting examples. In the first interaction round MBE generates 10 examples (table 2.1). MBE takes 9.92 seconds to initialize using the prototypical example and object model, and 35.37 seconds to generate and display all 10 examples on a dual 2 GHz PowerPC G5 machine with 4GB of RAM.

The interaction is similar to if the user asked a modeling expert to construct a file system model. After looking at the prototypical example the expert might ask, “The `entries` relation is functional in the prototype, but is that essential?” This is what MBE asks by generating the eighth example in the first interaction round (table 2.1). The user marks that example as included, from which MBE learns that `entries` is not functional. The last example MBE generates in the first interaction round (table 2.1) is equivalent to an expert asking the user whether `contents` is also not functional. The user marks this example as excluded, since `contents` is functional.

After the user marks all 10 examples, MBE refines its internal model and, in 22.47s, generates three examples in the second interaction round (table 2.2). MBE generates examples that ask more complex questions in later rounds. The second example in round two queries the user on whether directories can be aliased. The user marks this example as included, permitting `Dir` to share names.

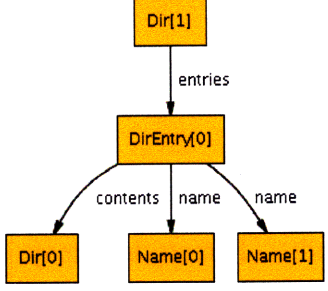
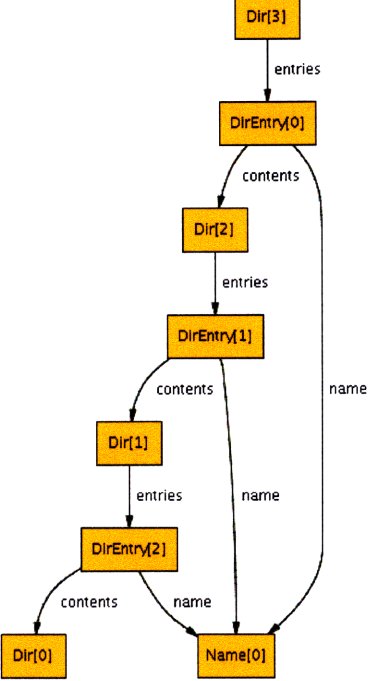
After the user marks all 3 examples in the second interaction round, MBE generates, in 11.41s, one more example (table 2.3). MBE refines its model and then reaches a termination condition in 5.40s. The total running time of the algorithm, excluding the time it takes the user to mark examples, is 84.57s. All boundary aspects of its internal model have been corrected and verified; thus, it returns the textual goal model it constructed (figure 2-6).

Table 2.1: Examples generated by MBE in the first round of interaction

Reason why example is included or excluded by goal model	Example
Excluded: Dir[0] contains itself	 <pre> graph TD DE0[DirEntry[0]] -- contents --> D0[Dir[0]] DE0 -- name --> N0[Name[0]] D0 -- entries --> DE0 </pre>
Excluded: contents is not total (DirEntry[0] does not contain some Dir object)	 <pre> graph TD D0[Dir[0]] -- entries --> DE0[DirEntry[0]] DE0 -- name --> N0[Name[0]] </pre>
Excluded: names is not total (DirEntry[0] does not contain some Name object)	 <pre> graph TD D1[Dir[1]] -- entries --> DE0[DirEntry[0]] DE0 -- contents --> D0[Dir[0]] </pre>
Excluded: names is not surjective (Name[0] is not contained by some DirEntry)	 <pre> graph TD D0[Dir[0]] N0[Name[0]] </pre>

Continued on Next Page...

Table 2.1 – Continued

Reason why example is included or excluded by goal model	Example
<p>Excluded: names is not functional (DirEntry[0] contains two Name objects)</p>	 <pre> graph TD D1[Dir[1]] -- entries --> DE0[DirEntry[0]] DE0 -- contents --> D0[Dir[0]] DE0 -- name --> N0[Name[0]] DE0 -- name --> N1[Name[1]] </pre>
<p>Included: Directories with different parents may share the same name</p>	 <pre> graph TD D3[Dir[3]] -- entries --> DE0[DirEntry[0]] DE0 -- contents --> D2[Dir[2]] DE0 -- name --> N0[Name[0]] D2 -- entries --> DE1[DirEntry[1]] DE1 -- contents --> D1[Dir[1]] DE1 -- name --> N0 D1 -- entries --> DE2[DirEntry[2]] DE2 -- contents --> D0[Dir[0]] DE2 -- name --> N0 </pre>
<p>Excluded: no root directory</p>	<p>[example with no elements]</p>

Continued on Next Page...

Table 2.1 – Continued

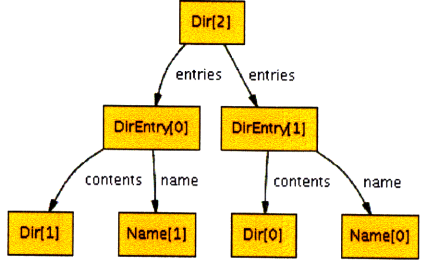
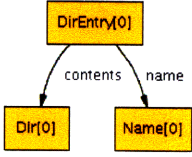
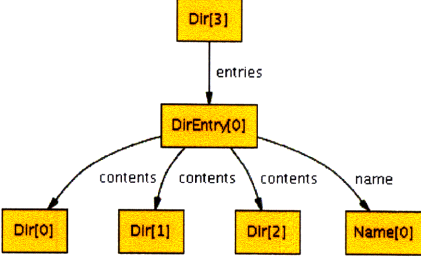
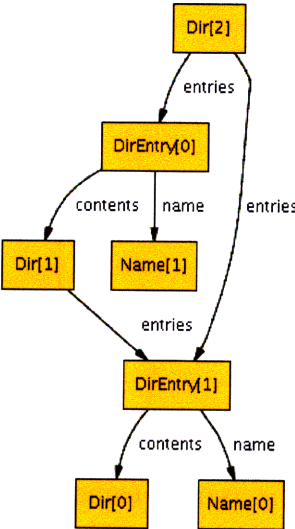
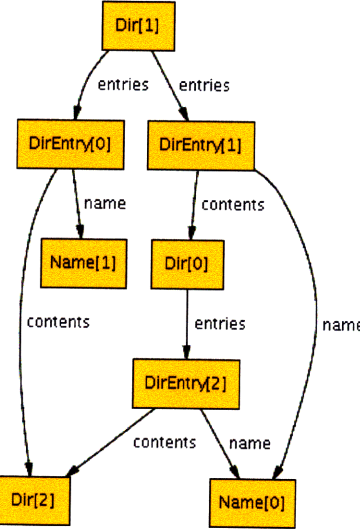
Reason why example is included or excluded by goal model	Example
Included: Directories may contain multiple directories	 <pre> graph TD D2[Dir[2]] -- entries --> DE0[DirEntry[0]] D2 -- entries --> DE1[DirEntry[1]] DE0 -- contents --> D1[Dir[1]] DE0 -- name --> N1[Name[1]] DE1 -- contents --> D0[Dir[0]] DE1 -- name --> N0[Name[0]] </pre>
Excluded: entries is not surjective (DirEntry[0] is not contained by some Dir)	 <pre> graph TD DE0[DirEntry[0]] -- contents --> D0[Dir[0]] DE0 -- name --> N0[Name[0]] </pre>
Excluded: contents is not functional (DirEntry[0] contains multiple Dir)	 <pre> graph TD D3[Dir[3]] -- entries --> DE0[DirEntry[0]] DE0 -- contents --> D0[Dir[0]] DE0 -- contents --> D1[Dir[1]] DE0 -- contents --> D2[Dir[2]] DE0 -- name --> N0[Name[0]] </pre>

Table 2.2: Examples generated by MBE in the second round of interaction

Reason why example is included or excluded by goal model	Example
<p>Excluded: <code>entries</code> is not injective (<code>DirEntry[1]</code> is contained by multiple <code>Dir</code>)</p>	 <pre> graph TD Dir2[Dir[2]] -- entries --> DirEntry0[DirEntry[0]] DirEntry0 -- contents --> Dir1[Dir[1]] DirEntry0 -- name --> Name1[Name[1]] Dir1 -- entries --> DirEntry1[DirEntry[1]] Name1 -- entries --> DirEntry1 DirEntry1 -- contents --> Dir0[Dir[0]] DirEntry1 -- name --> Name0[Name[0]] </pre>
<p>Included: Directories may be aliased</p>	 <pre> graph TD Dir1[Dir[1]] -- entries --> DirEntry0[DirEntry[0]] Dir1 -- entries --> DirEntry1[DirEntry[1]] DirEntry0 -- name --> Name1[Name[1]] DirEntry1 -- contents --> Dir0[Dir[0]] Name1 -- contents --> Dir2[Dir[2]] Dir0 -- entries --> DirEntry2[DirEntry[2]] DirEntry2 -- contents --> Dir2 DirEntry2 -- name --> Name0[Name[0]] Name0 -- name --> Dir1 </pre>

Continued on Next Page...

Table 2.2 – Continued

Reason why example is included or excluded by goal model	Example
<p>Excluded: Sibling directories share the same name (Dir[0] and Dir[1] are both Name[0]/Name[0])</p>	<pre> graph TD D2[Dir[2]] -- entries --> DE2[DirEntry[2]] DE2 -- contents --> D3[Dir[3]] D3 -- entries --> DE1[DirEntry[1]] D3 -- entries --> DE0[DirEntry[0]] DE1 -- contents --> D0[Dir[0]] DE0 -- name --> N0[Name[0]] DE0 -- name --> D1[Dir[1]] DE0 -- contents --> D1 D1 -- name --> N0 DE2 -. name .-> N0 </pre>

Table 2.3: Examples generated by MBE in the third round of interaction

Reason why example is included or excluded by goal model	Example
Included:	<pre> graph TD D0[Dir[0]] -- entries --> DE3[DirEntry[3]] D0 -- entries --> DE2[DirEntry[2]] DE3 -- contents --> D1[Dir[1]] DE2 -- name --> N0[Name[0]] D1 -- entries --> DE0[DirEntry[0]] D1 -- entries --> DE1[DirEntry[1]] DE0 -- name --> N2[Name[2]] DE1 -- name --> N1[Name[1]] DE1 -- contents --> D2[Dir[2]] DE3 --> N2 DE2 -- contents --> D2 </pre>

2.3 Equivalence of User-Constructed and MBE-Generated Textual Models

MBE generates a textual model (figure 2-6) that is behaviorally equivalent to the model written by the user in section 2.1. The wording of constraints in each model may be different, but every example that is included (excluded) by one model is included (excluded) by the other. A side-effect of MBE's process is a mutation-complete test suite for the model. The test suite is mutation-complete in that there is a test case for all boundary aspects identified by extracting an initial model from the prototypical example.

```
(functional, contents)
(functional, name)
(inner_injective, ternary[entries, name])
(injective, entries)
(surjective, entries)
(rootedOne, ternary[entries, contents])
(acyclic, ternary[entries, contents])
(total, contents)
(total, name)
(surjective, name)
```

Figure 2-6: MBE's generated textual model for file system. Alloy declarations for the mathematical predicates are given in table A.1

Chapter 3

Learning

In this chapter we explain how modeling by example works. We start by defining the components over which learning occurs, followed by an explanation of the learning process itself. We conclude by rationalizing the tradeoffs made.

3.1 Definitions

The **model** that MBE constructs is a conjunction of constraints on the types and relations declared in the object model provided by the user.

A **constraint** is the application of a **predicate** to types and relations. For example, if the object model defines a tree (figure 3.1 (a)), then

(acyclic, children)

indicates the constraint that the children relation satisfies the acyclic predicate. MBE has 16 basic graph and relation predicates from which it generates constraints for a particular object model (table A.1).

Using an overloaded form of the acyclic predicate, we can apply the acyclic predicate to a type expression, such as non-rooted nodes

(acyclic, children, node – root).

This means that the root node may have self-edges, however all other nodes are not involved in cycles.

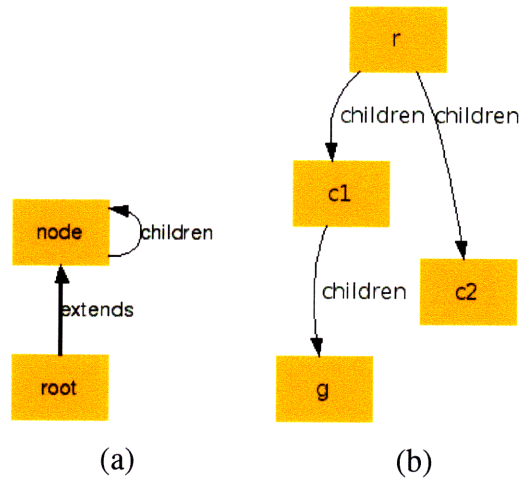


Figure 3-1: (a) Object model and (b) prototypical example for a simple tree structure

To express interaction among relations and types we apply a predicate to an expression over relations and types. For example, if the object model defines a file system (figure 3-2), then

(symmetric, parent + entries.contents)

indicates the constraint that parent and the relational join of entries and contents together satisfy the symmetric predicate. Combining this constraint with constraints against cycles in parent or entries.contents, and we constraint examples such that if directory *a* contains directory *b* (through entries.contents), then *a* is also *b*'s parent.

If our model included multiple kinds of file system objects, such as Files and Dirs, and only Dirs have parents, then we would have to restrict the range of the type expression

(symmetric, parent + entries.contents :> Dir).

MBE interacts with the user via **examples**. An example is a particular mapping of variables to atoms (elements of a type) and tuples (elements of a relation) according to some object model. A **prototypical example** is an ideal example that satisfies the user's goal model. In particular, it demonstrates a lack of compliance to undesirable models. For ex-

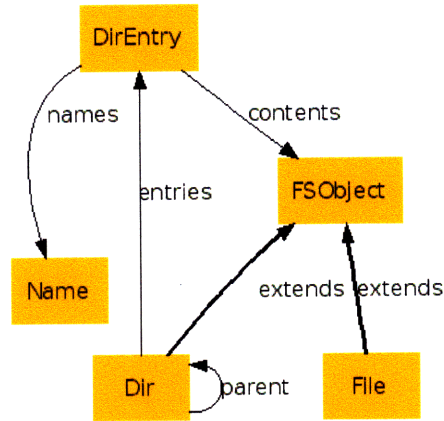


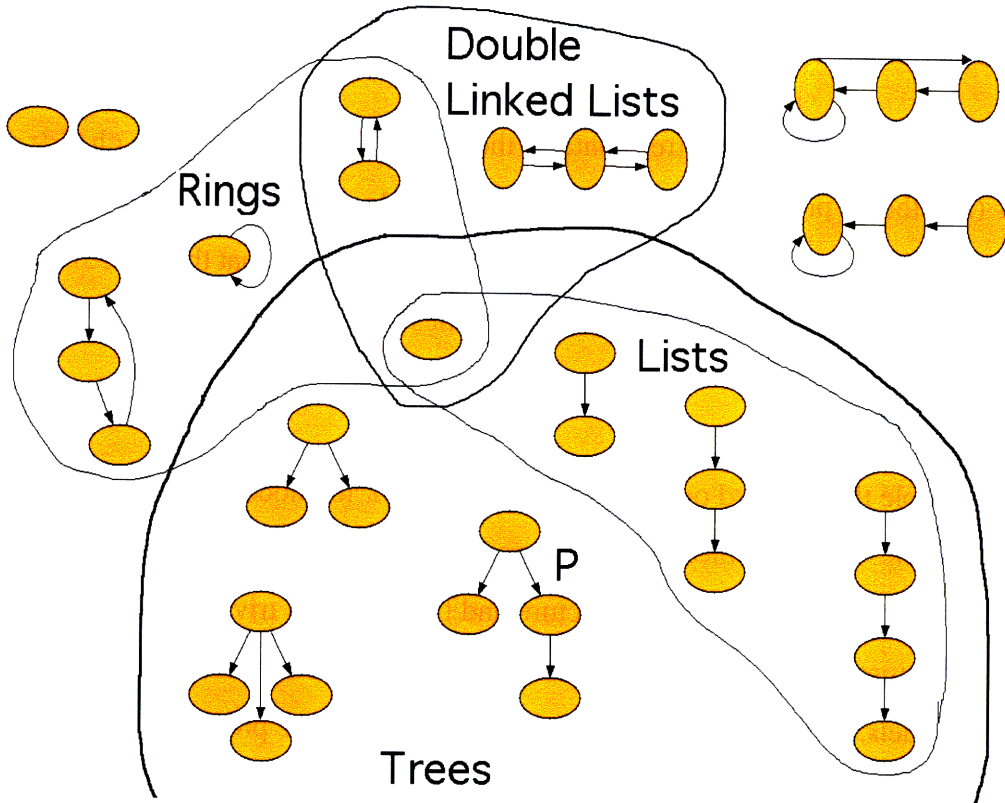
Figure 3-2: Object model for modeling a file system

ample, if an object model for a tree contains the type node and the relation children (figure 3.1 (b)), then a prototypical example might be $\text{node} = \{r, c_1, c_2, g\}$ and $\text{children} = \{\langle r, c_1 \rangle, \langle r, c_2 \rangle, \langle c_1, g \rangle\}$ (figure 3.1). This example demonstrates that the children relation is acyclic, rooted, and injective, which are properties of a tree structure, not a linked list or ring.

An example that showed only a single node atom or ten nodes in a line is not a prototypical example. Although these examples are included by the model, the line example first appears to be a more restricted list structure and the singleton trivially satisfies predicates of many structures because the relation is the empty set. Because each example also holds for other models, neither is a good prototypical example. The prototype holds for as few models as possible. Conceptually, if we consider the space of all examples in the universe, the prototypical example is at the center of the examples included by the model and is not included by any model except for strict supersets (figure 3-3).

A **marked example** is an example with an additional tag indicating whether it is included or excluded by the goal model. Like the object model, the markings on an example are supplied by the user. Marked examples are the supervised training data set that MBE uses to learn the goal model.

Figure 3-3: Metaphorical example space for tree model. The prototypical tree example, *P*, is farthest from the boundary of the tree model and is not included by the ring or list models.



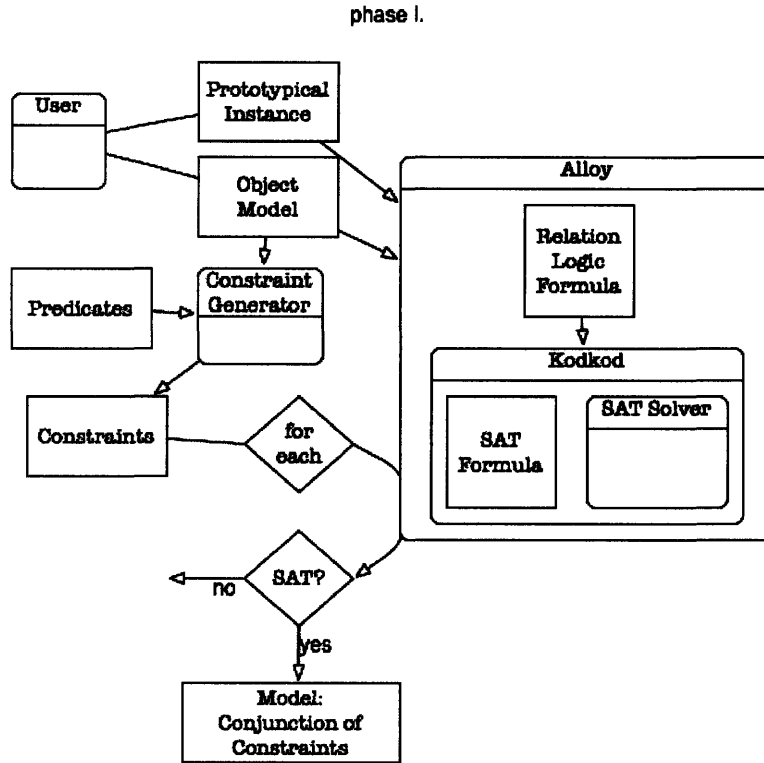


Figure 3-4: Phase 1 of the learning algorithm

3.2 Learning Algorithm

The user initiates learning by providing a object model of their goal model. MBE learns the goal model in the following three phases:

1. An initial model is extracted from the prototype (figure 3-4).
2. The extracted model is iteratively generalized using both included and excluded examples (figure 3-5).
3. The learning process is terminated when the user's goal model is either found or deemed inexpressible.

The rest of this section uses the file system model from section 2.2 as a running example at the end of each section. The complete learning process on this example is explained in section 4.4.

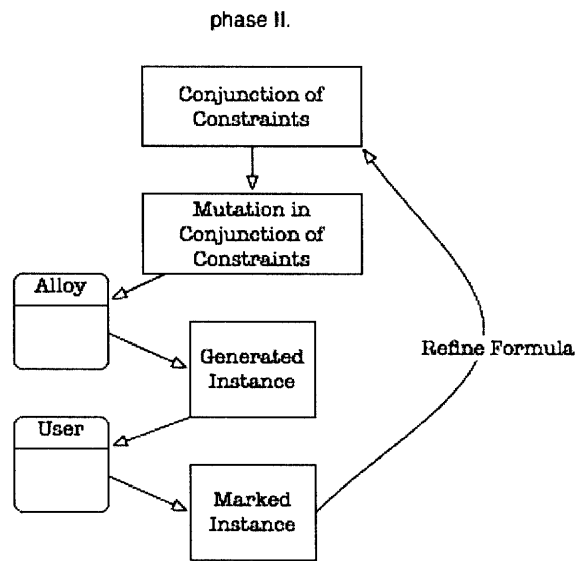


Figure 3-5: Phase 2 of the learning algorithm

3.2.1 Model Extraction

The first phase of learning is to generate constraints from the user's object model, and then extract an initial model from a user-provided prototypical example. The learning algorithm will work on any included example, however a prototypical model yields the most efficient learning process (section 3.3.1).

Constraint Generation

MBE generates constraints from an object model in two steps. First, all combinations of relations and types, called relation expressions and type expressions, are generated (figure 3-6). Then, all combinations of relation and type expressions are applied to all predefined predicates, which yields constraints. Since MBE learns from examples that are directed graphs, it has 16 predefined predicates taken from the graph and relation utility models that come with Alloy (table A.1).

Constraint generation is not straightforward. MBE's initial knowledge is in terms of predicates, which are model-*independent* properties, from which MBE generates model-*dependent* constraints. Increasing the variety of models that MBE can learn requires increasing the variety of constraints MBE generates. However, in generating some particular

$$\begin{aligned} \text{type_expression} &::= \text{type_name} \\ &\quad | \text{type_expression type_op type_expression} \\ \text{type_op} &::= + \mid - \end{aligned}$$

$$\begin{aligned} \text{relation_expression} &::= \text{relation_name} \\ &\quad | \text{relation_expression relation_op (relation_expression)} \\ &\quad | \text{type_expression} <: \text{relation_expression} \\ &\quad | \text{relation_expression} :> \text{type_expression} \\ \text{relation_op} &::= + \mid - \mid \sim \mid \cdot \end{aligned}$$

Figure 3-6: The grammar of type and relation expressions, where `type_name` and `relation_name` are types and relations defined by some object model

constraint necessary to express some model, MBE will also generate dozens of unnecessary constraints, many of which will be extracted into the initial model and must be removed by the generalization phase. Thus, increasing the expressiveness of the learning algorithm, with respect to the kinds of learnable models, greatly increases the cost of learning.

In the first step of constraint generation, MBE generates all non-trivial and non-redundant type expressions and relation expressions. Trivial expressions occur when the relation or type expression equals the empty set; e.g., $r - r$ or $t - t_0 - t_1$, where t_0 and t_1 are the only subtypes of t . Similarly, we only apply type operations that yield possibly non-empty sets; e.g., $r.s$ is only generated when the intersection of r 's range and s 's domain may be non-empty. Redundant expressions occur when equivalent sets are described by different expressions; e.g., $r + s$ and $s + r$, or $t - t_0$ and t_1 , where t_0 and t_1 are the only subtypes of t .

In the second step of constraint generation, each relation and type expression combination is applied, if applicable, to each predicate to yield a constraint. All constraints are generated except for redundant and unlikely constraints. Redundant constraints are constraints that are behaviorally equivalent; e.g., $(\text{acyclic}, r)$ and $(\text{acyclic}, \sim r)$, $(\text{acyclic}, r_1.r_2)$ and $(\text{acyclic}, r_1 + r_2)$ or $(\text{injective}, r)$ and $(\text{functional}, \sim r)$.

Unlike generating expressions, we do not know if a constraint will be trivially satisfied or unsatisfied, since that depends on the constraints in the goal model. However, we could apply domain specific knowledge to exclude constraints that are likely unnecessary for expressing a user's goal model (chapter 6). One optimization that yields the same set of expressible models as brute force and does not require that the learning algorithm backtrack is to dynamically generate constraints on subtypes in the object model. That is, MBE initially generates constraints by applying predicates to root types in the object model. Only when some constraint, c_t , is not held by an included example are constraints created by applying the predicate to subtypes of t . By constructing fine grained constraints only when necessary, learning is faster in practice without sacrificing the models that can be learned.

Figure 3-7: Predefined predicates for restricted file system example

acyclic: no element contains itself

functional: all elements contain at most one element

symmetric: for any two elements, a and b , if a points to b , then b points to a

Constraint Generation For File System Example For learning the file system model, suppose MBE has 3 predefined predicates (figure 3-7). MBE generates three constraints from the functional predicate since each declared relation could be functional. Unlike functional, the acyclic predicate only makes sense when the range and domain of the relation are the same type. Thus, only one constraint is generated from the acyclic predicate. Similarly, one constraint is generated from the symmetric predicate. This yields a total of 5 generated constraints (figure 3-8).

Extracting an Initial Model

MBE mechanically extracts an initial model from the prototypical example. The **initial model** is a conjunction of all constraints that the prototypical example individually satisfied.

Figure 3-8: Generated constrains for file system example with three predefined predicates

(acyclic, `entries.contents`) – no `Dir` contains itself

(functional, `name`) – all `DirEntry` have at most one name

(functional, `contents`) – all `DirEntry` have at most one `contents`

(functional, `entries`) – all `Dir` have at most one `entries`

(symmetric, `entries.contents`) – for two `Dir`, a and b , if a contains b then b contains a

Because the learning process iteratively refines the initial model, the learning speed is greatly impacted by the choice of initial model. MBE reduces the size of the initial model by using dynamic constraint construction. Further refinements on the initial model are explain in section 3.4.2.

Dynamic Constraint Construction MBE reduces the size of the initial model by dynamically generating constraints on subtypes in the object model. That is, MBE initially generates constraints by applying predicates to root types in the object model. Only when some constraint, c_t , is not held by an included example are constraints created by applying the predicate to subtypes of t . By constructing fine grained constraints only when necessary, learning is faster in practice without sacrificing the models that can be learned.

Extracting an Initial File System Model MBE tests whether the prototype (figure 2-5) satisfies a model containing a single constraint. This test is performed for all generated constraints (figure 3-8).

The prototypical example satisfies all of the listed constraints except for (symmetric, `entries.contents`); thus, the extracted initial model is the conjunction of the following

constraints

(acyclic, entries.contents)

(functional, name)

(functional, contents)

(functional, entries).

3.2.2 Model Generalization

The second phase of learning is to generalize the initial model. Generalization is required because the initial model may overfit the prototypical example(s). Overfitting occurs when a predicate that is true for one included example is false for another included example. In the running file system example from section 3.2.1, the initial model contained the overconstraint, (functional, entries). Although the prototype satisfies this constraint, a file system example with a root directory contain two directories does not. The overconstraint happened to be true for the prototypical example, but it is not essential or invariant to all included examples. If the overconstraint is not eliminated, then MBE's model is overconstrained with respect to the goal model.

To eliminate overconstraints MBE generalizes the model by generating examples for the user to mark, thereby forming a supervised training dataset over which MBE learns the goal model.

Obtaining Supervised Training Data

MBE generates examples that are different from both each other and the prototypical example in order to generalize the model using as few generated examples as possible. Unlike the prototypical example, which exhibits standard, or central, characteristics of the model, generated examples are near the boundary of the model, which is the part of the model MBE is least confident is right. Included examples that would be excluded were their constraints slightly modified are called **near hits**; excluded examples that would be included were their constraints slightly modified are called **near misses** [36]. Thus, MBE obtains

a supervised training data set by generating near hits and misses for the user to mark as included or excluded by the goal model.

If MBE were to generate all near hits and misses upfront for the user to mark, then information obtained by initial marks may be redundant with later marks. In order to minimize the size of the training data set supervised by the user, we employ an iterative learning process that maximizes what is learned from each marked example.

Generalization Algorithm

MBE attempts to learn the goal model using a relatively small training data set. Even with an efficient UI, we would like the user to mark as few examples as necessary. Conventional example-based learning techniques statistically generalize a model and thus rely on training data sets of hundreds or thousands of examples. The main difference between our learning algorithm and other instance-based learning techniques is that our algorithm dynamically constructs its training data set based on what it has learned so far. In this respect, our algorithm is similar to techniques for mutation testing.

MBE uses mutation to learn a formula: the generalization phase of MBE's learning process creates n mutant formulas from an initial formula of n constraints, and then tries to distinguish mutants that are real faults, which indicates the mutation of an essential constraint, from mutants that are semantically equivalent to the desired formula, which indicates the mutation of an over-constraint.

Our approach is to systematically determine for each constraint in the initial model whether it is essential to the goal model or an overconstraint. Only when MBE lacks information necessary for determining the role of a constraint does it generate the near hit or near miss example for the user to mark.

Overconstraints To learn which constraints of the initial model are overconstraints MBE tests each one. MBE creates n models, $M_{c_1}, M_{c_2}, \dots, M_{c_n}$, where n is the number of constraints in the initial model, M , and

$$M_{c_i} = M \setminus \{c_i\} \cup \{\neg c_i\} \text{ where } 1 < i \leq n \text{ and } c_i \in M.$$

For each M_{c_i} MBE finds an example, I , that satisfies M_{c_i} . If the user marks I as acceptable then c_i must be an overconstraint. Thus, MBE generalizes M to $M - \{c_i\}$.

We generalize the process of learning overconstraints with the following overconstraint detection rule:

If a mutated model generates an included example, then the negated constraints are overconstraints.

Detecting Overconstraints in File System Example In the file system example from section 3.2.1, the mutated models are constructed by negating each constraint in the initial model (figure 3-9).

Figure 3-9: Mutated models from the initial file system model extracted in section 3.2.1

$$\begin{aligned}
 M_{(\text{acyclic}, \text{entries}, \text{contents})} &= \neg(\text{acyclic}, \text{entries}, \text{contents}) \wedge \\
 &\quad (\text{functional}, \text{name}) \wedge \\
 &\quad (\text{functional}, \text{contents}) \wedge \\
 &\quad (\text{functional}, \text{entries}) \\
 M_{(\text{functional}, \text{name})} &= (\text{acyclic}, \text{entries}, \text{contents}) \wedge \\
 &\quad \neg(\text{functional}, \text{name}) \wedge \\
 &\quad (\text{functional}, \text{contents}) \wedge \\
 &\quad (\text{functional}, \text{entries}) \\
 M_{(\text{functional}, \text{contents})} &= (\text{acyclic}, \text{entries}, \text{contents}) \wedge \\
 &\quad (\text{functional}, \text{name}) \wedge \\
 &\quad \neg(\text{functional}, \text{contents}) \wedge \\
 &\quad (\text{functional}, \text{entries}) \\
 M_{(\text{functional}, \text{entries})} &= (\text{acyclic}, \text{entries}, \text{contents}) \wedge \\
 &\quad (\text{functional}, \text{name}) \wedge \\
 &\quad (\text{functional}, \text{contents}) \wedge \\
 &\quad \neg(\text{functional}, \text{entries})
 \end{aligned}$$

An example generated from $M_{(\text{functional}, \text{entries})}$ will contain a directory with more than one children. Such an example is acceptable to a user, and thus marked as included, from which MBE will learn that $(\text{functional}, \text{entries})$ is an overconstraint.

Once an overconstraint is identified MBE removes it from the working model. In this case, the presence or absence of (functional, entries) prevents the inclusion of goal examples; thus, MBE removes (functional, entries) from the working model.

Essential Constraints On the other hand, if examples generated from a mutated model, M_{c_i} , are excluded, then c_i must be essential.

Learning essential constraints is governed by the essential constraint detection rule:

If a mutated model generates an excluded example, then the negated constraint is essential.

Detecting Essential Constraints in File System Example In the file system example, no included example satisfies $M_{(\text{acyclic}, \text{entries}, \text{contents})}$. All examples in which there exists some directory that contains itself is not an example of a file system. Thus, $M_{(\text{acyclic}, \text{entries}, \text{contents})}$ is not an overconstraint, but instead is an essential constraint in the goal model.

Redundant Constraints In addition to being essential or overconstraints, constraints can also be redundant with other constraints in the model. A redundant constraint is implied by a subset of other constraints in the model. If c_j is redundant with a set of constraints, C_i , then $C_i \implies c_j$, which we can rewrite as $C_i \wedge \neg c_j$, where we expect $C_i \wedge \neg c_j$ to be unsatisfiable. Therefore, when a mutated model, M_{c_j} , satisfies no example it is because c_j is redundant with a subset of other constraints, C_i , in M_{c_j} .

Learning redundant constraint is governed by the redundant constraint detection rule:

If a mutated model generates no examples, then the negated constraints are redundant with non-negated constraints in the model.

Detecting Redundant Constraints in File System Example Suppose we expand the universe of predicates in the file system example from section 3.2.1 to include irreflexive

(no self-loops). The initial model, M , now contains the constraints

$$\begin{aligned} &(\text{acyclic}, \text{entries.contents}) \wedge \\ &(\text{functional}, \text{name}) \wedge \\ &(\text{functional}, \text{contents}) \wedge \\ &(\text{functional}, \text{entries}) \wedge \\ &(\text{irreflexive}, \text{entries.contents}) \end{aligned}$$

The mutated model $M_{(\text{irreflexive}, \text{entries.contents})}$ generates no examples because $(\text{acyclic}, \text{entries.contents})$ implies $(\text{irreflexive}, \text{entries.contents})$. All acyclic graphs necessarily contain no self-loops. Thus, $(\text{irreflexive}, \text{entries.contents})$ is a redundant constraint.

Similarly, if predicates were added for `weaklyConnected` (all elements are reachable from any other element by following the specified relation in either direction) and `rootedOne` (all elements are reachable from one root element), then the initial model would contain the redundant constraint $(\text{weaklyConnected}, \text{entries.contents})$, since all rooted graphs are necessarily weakly connected.

In this version of the file system example, the presence of redundant examples is harmless. A model with redundant constraints is behaviorally equivalent to a model without redundant constraints. That is, both models include and exclude the same examples.

Whether MBE should remove redundant constraints is tricky. Redundant constraints clutter the model, making it more difficult for users to understand and extend the goal model returned by MBE. However, we cannot assume that redundant constraints are implied by *essential* constraints.

If an overconstraint implies another overconstraint, then both constraints must be removed from the model. In the file system example from section 3.2.1, this might occur if the prototypical example were a single directory. Then the extract model would include all constraints, including the overconstraint $(\text{complete}, \text{entries.contents})$, which implies the overconstraint $(\text{symmetric}, \text{entries.contents})$.

On the other hand, if an overconstraint implies an essential constraint, then the redun-

dant constraint must not be removed from the model, and in fact will not be redundant once the overconstraint is removed. In the complex file system example from section 3.4.2, this would occur if the initial model contains the overconstraint `(complete, entries.contents)`, which implies the essential constraint `(symmetric, entries.contents + parent :> Dir)`.

Since a redundant constraint may be an overconstraint or essential, it is crucial to develop a strategy for distinguishing these two cases.

A Priori Lookup Table Some predicates always imply other predicates when applied to the same relation and type (chapter A). Thus, MBE uses a static lookup table of these predicates from which it can check whether some constraint, c_i , implies another constraint, c_j .

In order to test the role of c_j , MBE constructs a model with c_j negated and c_i removed. MBE determines c_i via the lookup table. Whenever MBE negates some constraint, c_j , it uses the lookup table to find all constraints, C_i , that imply c_j and removes them from the model. This enables MBE to generate an example for the user to mark so that it can learn the role of c_j .

Using a Lookup Table in the File System Example Using *a priori* knowledge in the file system example in section 3.4.2 allows MBE to detect that

`(symmetric, entries.contents + parent :> Dir)`

is an overconstraint immediately. Whenever

`(symmetric, entries.contents + parent :> Dir)`

is negated, `(complete, entries.contents)` is temporarily removed from the mutated model.

However, for this technique to work all implications among constraints have to be foreseen before knowing the object model. That is, we must determine potential implications among constraints knowing only how predicates interact. Unfortunately, multiple relations can interact across type hierarchies to create redundant constraints. For ex-

ample, when negating (symmetric, entries.contents), how does MBE know to negate (complete, parent) but not (complete, name)? Sometimes a handful of constraints or a dozen imply other constraints. Such complex relationships cannot be determined without an intelligent understanding of the model.

Thus, MBE uses a lookup table as an optimization rather than a solution. The solution is to figure out implications between constraints on the fly.

Dynamically Determining Implications MBE uses a guess-and-check technique to dynamically determine the implication's antecedent. MBE guesses some set of constraints, G , that might imply c_j , the redundant constraint. MBE then creates a mutated model, $M_{c_j \cup G}$, with c_j negated, and all of the constraints in G negated, as well. If $M_{c_j \cup G}$ generates no examples, then G does not imply c_j and the guess was wrong. MBE keeps guessing until $M_{c_j \cup G}$ generates examples, which indicates that G implies c_j . By systematically guessing all possible G of size n before guessing G of size $n + 1$, MBE ensures that the found G contains no unnecessary constraints.

The goal of this process is to determine whether c_j is an overconstraint or essential. Since c_j is redundant with G , the role of c_j is dependent on whether G contains only essential constraints, only overconstraints or a mix of both (table 3.1). Note that essential constraints by themselves cannot imply an overconstraint, nor overconstraints by themselves imply an essential constraint. However, a mix of essential and overconstraints may imply either.

Table 3.1: If $M_{c_j \cup G}$ generates an example, E , then E is included or excluded depending on whether G_i and c_j contain essential constraints or overconstraints

G contains . . .	c_j is an essential constraint	c_j is an overconstraint
both essential and overconstraints	E is excluded	E is excluded
overconstraints	(impossible)	E is included
essential constraints	E is excluded	(impossible)

If the examples generated from $M_{c_j - G}$ are excluded, then at least one constraint in $c_j \cup G$ is essential. If MBE knows that all of the constraints in G are essential, then it concludes

that c_j is essential, too. However, if the role of any constraint in G is unknown, then G may contain an overconstraint, and thus c_j may be an overconstraint. Thus, MBE cannot be certain that c_j is essential or an overconstraint. Instead, it continues the generalization process to determine the role of constraints in G before re-attempting to determine the role of c_j .

On the other hand, if M_{c_j-G} generates included examples, then c_j and all of the constraints in G are overconstraints. All of the generated examples are included because all essential constraints are still present. In this case, MBE can mark both c_j and all of the constraints in G as overconstraints.

3.2.3 Termination

MBE terminates the generalization process when any of the following conditions are met:

1. All constraints in the initial model have been classified into essential, overconstraint or redundant roles.
2. The learning process has reached a fix point but the roles of some constraints are unknown (section 3.2.2). The remaining unknown constraints are either essential or redundant with essential constraints.
3. The initial model generates an excluded example or the algorithm fails to make progress, indicating that the goal model is inexpressible because essential constraints are absent.
4. Inconsistencies in constraint determination indicate that the goal model is inexpressible.

The rest of this section explains each of the termination conditions in turn.

Condition I The first condition is a straightforward execution of the algorithm and is obvious to detect.

Condition II The second condition occurs because the learning process is unable to distinguish between constraints that are essential and constraints that are redundant with essential constraints until all overconstraints have been removed. Let EC be the set of constraints that are redundant with essential constraints. In order to remove the redundant constraints MBE guesses a subset of EC , EC_E , and tests whether $E \cup EC_E$ is equivalent to $E \cup EC$. $E \cup EC$ is a correct model based on the supervised training data set, so MBE removes constraints from EC so long as it does not change the behavior of the model. Thus, MBE removes constraints while

$$E \cup EC \iff E \cup EC_E.$$

holds. If the test holds, then the constraints in $EC - EC_E$ are redundant with essential constraints and can be removed. Otherwise, some constraint in $EC - EC_E$ is essential, although MBE does not know which one. MBE continues guessing until all constraints have been tested.

Termination in Linked List Example Suppose MBE is learning a linked list model with a single relation, `next`, linking nodes. After a number of generalization rounds (explained in detail in section 4.1), the end model contains the following five constraints

- (total, next)
- (injective, next)
- (surjective, next)
- (functional, next)
- (rootedAll, next)

MBE learns that (rootedAll, next) is essential, but negating any of the other four constraints generates no examples. The other four constraints are implied by each other and (rootedAll, next), and thus could be essential or redundant. When (total, next) and (injective, next) are negated, then negating any of the other constraints yields a model that generates excluded examples.

This indicates that (total, next) and (injective, next) are redundant. MBE constructs M_{sub} , which is M with (total, next) and (injective, next) removed, and tests whether the two models include and exclude the same examples ($M_{sub} \iff M_{end}$). As expected they do. The constraints (total, next) and (injective, next) are redundant, so MBE returns the most reduced form of the goal model, M_{sub} .

Condition III The third condition is detected during the first two phases of the learning process. Once an initial model is extracted from the prototypical example, all examples satisfied by the initial model are generated and given to the user to mark. If any of those examples are excluded then MBE immediately knows that there exists some essential predicate that is not in the available language. Because the model is overconstrained, the number of examples the user must view is small. Of the five models we evaluated, the initial models generated 1-4 examples.

Condition IV The fourth condition is detected when MBE finds an inconsistency in the role fulfilled by some constraint. Another way to spot the inconsistency is to monitor the following learning invariant:

The intersection of positive constraints in all models that satisfy some included example satisfy no excluded example.

The positive constraints in all models that satisfy some included example are at least the essential constraints, thus satisfying an excluded example implies that there exists an essential constraint not in expressed in the language. That is, all models are underconstrained. The missing essential constraint must be an absent predicate or else it would have been included in the initial model. Because the initial model is underconstrained, the learning process will be unable to find the goal model.

3.3 Rationale

This section rationalizes how the learning algorithm works and the tradeoffs made.

3.3.1 Prototypical Example

The learning process starts with a prototypical example to provide a context from which to learn. Without a context, a model that satisfies an excluded example tells us very little. Some of the constraints in the model may be wrong, and some essential constraints may be missing. For example, if the model is composed of three constraints, a , b and c , and the universe contains constraints a, b, \dots, y, z , then the explanation for why the model is wrong could be “ a is incorrect or b is incorrect or c is incorrect or d is essential or e is essential” The search space is over all possible constraints in the universe, where each constraint in the model may be incorrect and each constraint not in the model may be essential. If there are n constraints in the universe, then the goal model is one of 2^n possibilities. Furthermore, the information from a different model that generates only excluded examples need not narrow down the possible reasons why the original model is incorrect. In fact, even a large set of models that generate excluded examples cannot indicate the role of any constraint with complete certainty, since there may always exist some essential constraint missing from all the models.

On the other hand, once we have a model that satisfies an included example, we are guaranteed that no essential constraint is missing. If there did exist an essential constraint, c_e , that was not in the initial model it would mean that c_e , by itself, did not satisfy the prototypical example. But if c_e does not satisfy some included example then it is not essential. Thus, the initial model must contain all essential constraints.

If we know that a model is not missing any essential constraints, then if it satisfies excluded examples we know it must be because it includes an incorrect constraint, which we have previously shown to be an overconstraint. Finding a model that satisfies an included example is therefore crucial to the learning process, since then we need only ever consider the constraints that satisfy that initial model, and not all generated constraints. The total number of constraints in the universe grows exponentially in the number of relations and types because all expression and predicate combinations may be possible.

3.3.2 Grammar

Models in our approach are a conjunction of constraints.¹ This allowed the learning algorithm to assume that all essential constraints are present in any included example, permitting a straight forward learning process guaranteed to find expressible goal models. But what if the goal model is not expressible in this grammar?

When choosing the grammar we make a trade-off between usefulness with respect to the correctness and comprehensibility of the learned model and the efficiency of the learning process. As we increase the number of constraints in the universe, we increase the number of essential, redundant and overconstraints the algorithm must distinguish between. The cost of increasing the complexity of the grammar is especially high if the learning algorithm can make mistakes and have to backtrack.

One might think that avoiding disjunction, and thus implication, significantly limits what models our system can express. In fact, many models do not require disjunction or implication on the predicates that are invariant among all examples. Predicates typically describe invariants on the structure of relations, whereas model variation typically occurs on which types satisfy those predicates and not on the predicates themselves.

In the extended file system model, all `Dir` atoms have one parent except for the `Root`, a special kind of `Dir` that has no parent. Thus we have the implication

$$\begin{aligned} \forall d \in \text{Dir}, d \in \text{Root} &\implies \text{no } d.\text{parent} \wedge \\ d \notin \text{Root} &\implies \text{one } d.\text{parent}. \end{aligned} \tag{3.1}$$

A conjunction of constraints can express this kind of implication since the predicates of `no_parent` and `one_parent` are invariant on `Root` and `Dir - Root`, respectively. That

¹We have also constrained the initial and goal model to contain only positive constraints. The restriction on negating constraints is arbitrary, since it is equivalent to doubling the constraint language with the negation of each constraint.

is, 3.1 is equivalent to 3.2,

$$\begin{aligned} & \forall r \in \text{Root}, \text{ no } r.\text{parent} \wedge \\ & \forall d \in \text{Dir} - \text{Root}, \text{ one } d.\text{parent}. \end{aligned} \tag{3.2}$$

MBE expresses this invariant using `never_total`, which is equivalent to having no parent, and `function`, which is equivalent to having one parent, to the appropriate subtypes

$$(\text{never_total}, \text{parent}, \text{Root}) \wedge (\text{function}, \text{parent}, \text{Dir} - \text{Root}).$$

Models that require disjunction on the properties of relations, instead of on the properties of types, are not expressible using only conjunctions of constraints. For example, relation and type expressions are not able to express that a relation forms either a randomly accessible sequence *or* a linked list, but not other structures such as a tree. This kind of data structure complexity is unusual, but may be useful for optimization. For example, suppose we are adding elements to a data structure, and based on domain-specific knowledge we know that we will likely receive either a handful or a thousand of elements. We may want to initially allocate single linked list elements for each element that gets added; however, if the number of elements exceeds some threshold we then allocate a large chunk of randomly accessible memory. Thus, we have

$$\begin{aligned} \#e > \epsilon & \implies \text{array predicates hold} \wedge \\ \#e \leq \epsilon & \implies \text{linked list predicates hold,} \end{aligned}$$

where ϵ is some constant. There is no way to express this kind of implication in the current system.

3.4 Refinements

This section proposes techniques for improving the expressiveness, efficiency and usability of the learning algorithm.

3.4.1 Generating a Prototypical Example

If MBE is to generate the prototype mechanically then it needs a strategy for generating examples such that the task of finding an included example is not onerous. The naive approach is to apply no constraints and generate all examples within some bound on the number of atoms. A matrix of examples is presented to the user so that she may select one or more included examples. If none of the presented examples are included by the user's goal model, then a different matrix of generated examples is shown. By increasing the bounds on the number of atoms in the generated examples an included example must be identified eventually.

A more intelligent approach is to present examples that maximize two predicates: likeness of being included, so that the user is not repeatedly excluding examples that are nothing like what he wants; and differentness from previously excluded examples, so that the example space can be explored quickly. Heuristics based on common models would direct the search towards likely examples, while graph isomorphism algorithms (or less costly heuristics) would direct the search towards different examples.

Examples could also be ordered so that examples that would result in faster learning are shown first. Because learning is more efficient when the prototypical example displays more features, we would put large, more complex examples ahead of smaller ones. This must be balanced against confusing the user, as well as the cost of detecting differentness, which is higher for larger examples. MBE may also be able to learn from excluded examples which so that it can dynamically present examples with different predicates.

Complex models with many constraints on relations and types will benefit from the intelligent approach, whereas the naive approach is practical for simple model. The first included example selected may not be the best prototypical example for efficient learning, however the learning process will yield the same result regardless of the initial example. Furthermore, the user may select multiple included examples which together are equivalent to selecting a single prototypical example with respect to the initial model MBE extracts.

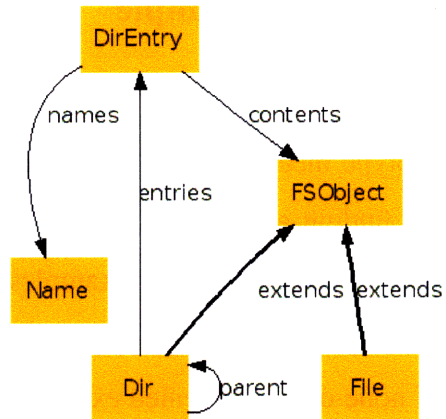


Figure 3-10: Object model for file system example containing two kinds of file system objects, *File* and *Dir*. Directories can contain other file system objects and, unlike files, have a single parent Directory

3.4.2 Multiple Prototypes

Instead of providing a single prototypical example, the user may provide multiple included examples. Extracting a single initial model from multiple included examples means taking the intersection of the models extracted individually from each included example. This is possible because MBE is already defined to learn models that are the conjunction of all constraints that hold for all included examples and do not hold for all excluded examples. Taking the intersection of a subset of included examples removes some overconstraints, thus multiple prototypes can yield a smaller initial model.

Not all included examples are prototypical examples. The learning algorithm uses the prototype to reduce the size of the model before the generalization phase. To accomplish this gain in efficiency, the prototype must demonstrate non-compliance to non-constraints. The worst prototypical example is one from which all available constraints are extracted into the initial model, since this means the prototypical example taught the learning algorithm nothing.

To compare prototypical examples we use a more complex version of the file system example than seen previously. The object model for this file system is shown in figure 3-10. The model contains two kinds of file system objects, files and directories. Directories can contain other file system objects and, unlike files, have a single parent Directory.

Table 3.2 compares the number of constraints extracted from different prototypical ex-

amples. The relations in (e) fully demonstrate file system properties, including a directory containing multiple files and directories, including a grandchild. Any overconstraint satisfied by (e) is satisfied by all other examples. Or rather, any overconstraint not satisfied by another example is not satisfied by (e), either.

Example (d) includes the overconstraints that `entries.contents` is transitive, and `contents` is functional for both directories and files. Examples (b) and (c) each demonstrate properties for only files or directories, respectively. Many overconstraints are trivially satisfied for the missing relations, and thus each is a poor choice for a prototypical example. Example (a) trivial satisfies almost all constraints since it lacks any relations.

Interestingly, the overconstraints in the sets for (b) and (c) do not completely overlap. Thus, these two poor examples could be combined to form a good example. Pictorially, (b) and (c) could be combined to form (d); alternatively, MBE could accept both as prototypical examples and then take the intersection of the extracted models.

Although accepting multiple included examples may be useful for guiding the user to provide examples that demonstrate all uses of relations and types, it is not more useful than the user providing the best prototypical example.

3.4.3 Learning Implications

Although we did not implement this algorithm, we now outline how implication and disjunction could be added to the learning process. First we describe an automatic technique that maintains the same kind of user interaction as before. Then we describe a semi-automatic technique that uses multiple prototypical examples to help MBE distinguish the conjunctive expressions in the disjunction.

Consider first how the current learning process handles implication. Let the goal model be $(a \implies b)$, where a , b and c are constraints. If a holds then b holds, or a does not hold and there are no other constraints; thus, our goal model is

$$(a \wedge b) \vee (\bar{a}).$$

Some prototypical example is provided from which MBE extracts the following initial


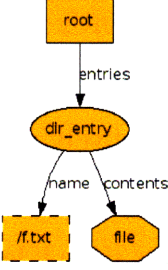
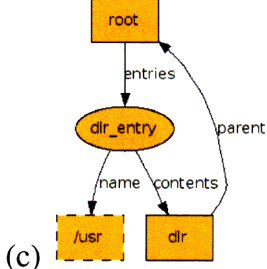
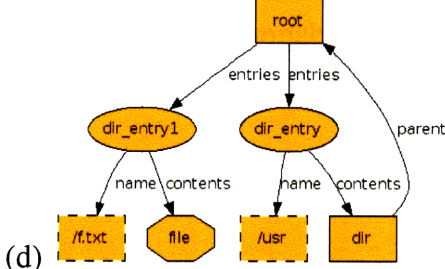
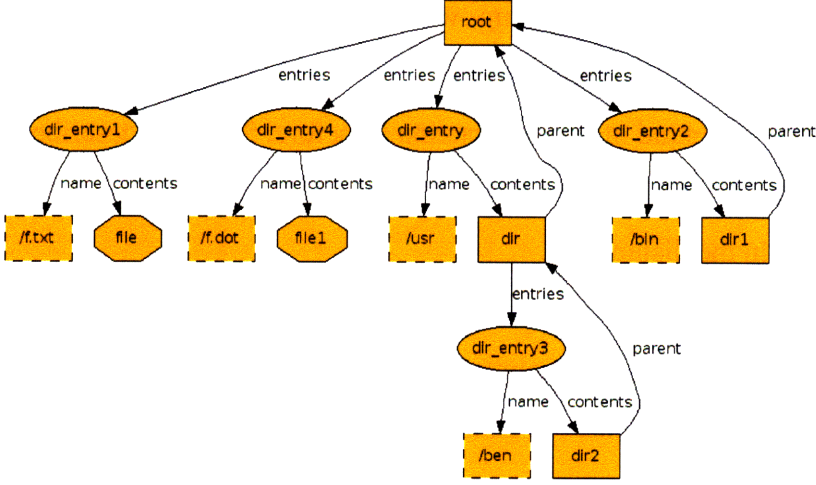
Prototypical example	Number of constraints in initial model
(a) 	193
(b) 	162
(c) 	155
(d) 	139
(e) 	137

Table 3.2: Comparison of the number of constraints extracted into the initial model for different prototypical examples. 197 constraints were generated overall by applying 16 predicates to the file system object model

model

$$a \wedge b \wedge c.$$

The negation of each constraint yields models $M_{\neg a}$, $M_{\neg b}$ and $M_{\neg c}$, which satisfy examples $I_{\neg a}$, $I_{\neg b}$ and $I_{\neg c}$, respectively (table 3.3). The user marks $I_{\neg a}$ as included, since if a is not true then any example is included. The user marks $I_{\neg b}$ as excluded, since a holds but b does not, which we know is contrary to the goal model $a \implies b$. Finally, the user marks $I_{\neg c}$ as included, since c need not necessarily hold.

Table 3.3: Constraints and the marked examples they generated

Model	Constraints			Examples satisfied
$M_{\neg a}$	$\neg a$	b	c	✓ included
$M_{\neg b}$	a	$\neg b$	c	✗ excluded
$M_{\neg c}$	a	b	$\neg c$	✓ included
$M_{\neg ab}$	$\neg a$	$\neg b$	c	✓ included
$M_{\neg bc}$	a	$\neg b$	$\neg c$	✗ excluded
$M_{\neg abc}$	$\neg a$	$\neg b$	$\neg c$	✓ included

From the user markings, MBE correctly learns that c is an overconstraint ($M_{\neg c}$ satisfies an included example), but incorrectly learns that a is an overconstraint ($M_{\neg a}$ satisfies an included example) and b is essential ($M_{\neg b}$ satisfies an excluded example). A model in which both a and b did not hold, $M_{\neg ab}$, satisfies an included example, and thus MBE would learn that b , its negation in models satisfying both included and excluded examples, is an overconstraint instead!

Implication introduces uncertainty and complexity into the learning process because the assumption that all essential constraints are in the initial model would be false. The next thing learned could force MBE to question everything it previously learned.

To learn disjunction MBE maintains multiple underlying models. The goal model is a disjunction of submodels that are conjunctions of constraints. If an example satisfies one of the submodels, it is included. The number of examples generated increases with the number of models because what is learned on one model cannot be assumed to hold on another model.

Automatic Implication Modeling

In order to learn models containing implication, the learning algorithm must not only detect that the goal model contains implication, but also the constraints involved. Some constraints may be invariant throughout, and some constraints may be in multiple antecedents.

Our approach is to assume a simple model structure, eg a conjunction of constraints, and then re-learn a more complicated model structure, eg two submodels, if an inconsistency is detected.

For example, to correctly learn the toy model $a \implies b$, the generalization proceeds as outlined in the previous section until the inconsistency with b as both essential and overconstraint is detected. Since the inconsistency is present when $\neg b$ holds, MBE looks at what is essential to models that contain $\neg b$ and are satisfied by included examples versus models that contain $\neg b$ and are satisfied by excluded examples.

In this example, all excluded examples satisfy the models containing a ($M_{\neg b}$ and $M_{\neg bc}$). On the other hand, models containing $\neg b$ that are satisfied by included examples ($M_{\neg ab}$ and $M_{\neg abc}$), also contain $\neg a$.

From this analysis, the algorithm learns that a is essential to determining whether models that contain $\neg b$ satisfy included or excluded examples.

Because c is not part of the implication's antecedent, the algorithm must determine separately in both submodels whether c is essential, redundant or an overconstraint. Models with positive a and b yield included examples regardless of whether c is negated, indicating that c is an overconstraint in the submodel when the implication antecedent (a) holds. Models with negative a yield included examples regardless of whether c is negated, indicating that c is an overconstraint in the submodel when the implication antecedent does not hold.

The implication antecedents must be correctly determined in order to distinguish correct submodels, which in turn are required for correctly analyzing marked examples. Consider the case where $(a \implies b) \vee (\neg a \implies c)$ is the goal model and the prototypical example for the first clause generates $a \wedge b \wedge c$. Until MBE knows that $\neg a \implies c$, it will believe that $\neg a \wedge b \wedge c$ satisfying an included example indicates that a is an overconstraint.

Thus, automatic implication detection requires more information than the original al-

gorithm that assumed the goal model was a conjunction of constraints. Potentially 2^n constraint combinations, where n is the number of constraints in the initial model, must be negated to use automatic implication detection. Although the exponential increase in mutated models results in an exponential increase in computational costs, the number of generated examples may not increase that quickly, since it is likely that generated examples satisfy multiple mutated models, especially those with overlapping negated constraints.

Nonetheless, the costs of detecting and learning multiple implications can be removed by relying on the user to provide separate prototypical examples for each submodel.

Semi-Automatic Implication Modeling

The cost of learning models that contain implications is largely in determining the constraints in the antecedents. Users can recognize distinct classes of included examples and provide one prototypical example from each class. This removes the cost of implication detection from the learning algorithm, and, by comparing initial models, provides useful initial knowledge for efficiently determining the implication antecedents.

In section 3.4.2 we showed how the algorithm can be improved by accepting multiple prototypical examples for the same submodel. The learning algorithm can still take advantage of this when there are multiple submodels by first assuming each extracted submodel to be unique. A submodel is extracted and generalized until there is sufficient confidence that it is redundant with another submodel, at which point it is removed. Requiring complete confidence puts this step in the termination phase of the learning algorithm; however, the amount of redundant computation during learning can be reduced by requiring only a portion of essential, redundant and overconstraints to be known when comparing whether two submodels are the same.

Chapter 4

Evaluation

We performed empirical tests on 5 models: singly linked list; doubly linked list; trivial file system; simple file system; and complex file system. The computation times are summarized in table 4.1. All reported computed times in this chapter were run on a dual 2 GHz PowerPC G5 machine with 4GB of RAM. All learning algorithm optimizations discussed in chapter 3 were used, including heuristics that prevent unlikely constraints from being generated.

Table 4.1: Summary of MBE's learning time on different models

Time in seconds:	Phase I	Round 1	Round 2	Round 3	Phase III	Total
Singly linked list	4.09	2.52	-	-	0.08	6.69
Doubly linked list	11.25	15.04	-	-	3.72	30.01
Trivial file system	4.15	2.54	2.26	-	0.15	9.10
Simple file system	9.92	35.37	22.47	11.41	5.40	84.57
Complex file system	20.17	50.35	73.97	72.19	146.68	...

4.1 Singly Linked List

MBE applies 16 predicates to the singly linked list object model (figure 4-1) to generate 17 constraints. Using the prototypical example (figure 4-2), MBE extracts an initial model with 8 constraints (figure 4-3). Together these steps take 1.71s.

MBE generates 4 examples (table 4.3) in the first round of interaction, taking 7.60s. The fourth example, which the user marks as included (acceptable), shows that (transitive, next)

Table 4.2: Summary of MBE’s learning time on different models

	Generated predicates	Constraints in initial model	Constraints in goal model	Constraints in most reduced model
Singly linked list	17	8	3	3
Doubly linked list	22	13	8	4
Trivial file system	18	8	3	3
Simple file system	69	18	10	10
Complex file system	88	28	-	(19)

is an overconstraint. The first, second and third examples, marked as excluded (unacceptable), show that (functional, next), (acyclic, next) and (rootedOne, next), are essential constraints. Using a pre-defined look-up table (appendix A), MBE removes constraints that are implied by essential constraints. This includes (antisymmetric, next) and (irreflexive, next), implied by (acyclic, next), and (weaklyConnected, next) implied by (rootedOne, next).

After the first round, only one constraint, (injective, next), is not declared essential, redundant or overconstraint. However, $M_{(injective, next)}$ generates no examples, indicating that (injective, next) is redundant. Since the rest of the constraints in M are essential, (injective, next) can safely be removed.

Finally, MBE returns the goal model (figure 4-4).

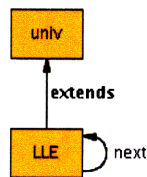


Figure 4-1: Singly linked list object model

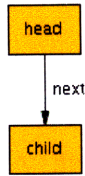


Figure 4-2: Singly linked list prototypical example

Table 4.3: Instances generated by MBE in the first round of interaction

Reason why instance is included or excluded by goal model	Instance
Excluded: <code>next</code> relation is not functional	
Excluded: <code>next</code> is not acyclic and, since acyclic implies irreflexive, <code>next</code> is not irreflexive	
Excluded: <code>next</code> is not rooted and, since acyclic implies weaklyConnected, <code>next</code> is not connected	

Continued on Next Page...

(antisymmetric, next) \wedge
 (irreflexive, next) \wedge
 (acyclic, next) \wedge
 (functional, next) \wedge
 (injective, next) \wedge
 (rootedOne, next) \wedge
 (weaklyConnected, next) \wedge
 (transitive, next)

Figure 4-3: Initial model extracted from singly linked list prototype (figure 4-2)

Table 4.3 – Continued

Reason why instance is included or excluded by goal model	Instance
Included: LLE may contain any depth of descendants (next is not transitive)	<pre> graph TD LLE3[LLE[3]] -- next --> LLE1[LLE[1]] LLE1 -- next --> LLE0[LLE[0]] LLE0 -- next --> LLE2[LLE[2]] </pre>

4.2 Doubly Linked List (good prototypical example)

One fun example of a doubly linked list is the ceiling and floor model in Alloy's example folder. We tested MBE on constructing a model that met the ceiling and floor requirements:

$$\begin{aligned}
&(\text{acyclic}, \text{next}) \wedge \\
&(\text{functional}, \text{next}) \wedge \\
&(\text{rootedOne}, \text{next})
\end{aligned}$$

Figure 4-4: Singly Linked List goal model constructed by MBE

1. Every man's ceiling is another man's floor
2. Two different men cannot share the same ceiling or floor
3. All men have exactly one ceiling and one floor

MBE applies 16 predicates to the ceiling and floor object model (figure 4-5) to generate 17 constraints. Using the prototypical example (figure 4-6), MBE extracts an initial model with 13 constraints (figure 4-7). Together these steps take 2.69s.

MBE generates 4 examples (table 4.4) in the first round of interaction, taking 10.60s. All of these examples are generated from models with two or three negated constraints. Because of the symmetry of ceiling and floor, no single constraint can be negated. For example, if a constraint generated from the surjective predicate is negated, then all other constraints generated from the surjective predicate must also be negated or removed. Furthermore, had the generated relation expression been more expressive, eg. `ceiling+~floor` and `ceiling.~floor`, then 63 constraints would have been generated, 31 constraints would be in the initial model, and it would take 25.62s to generate the two instances in the first of ten rounds. By preventing transpose, we prevent generating relation expressions with identical range and domain, which prevents applying a number of graph predicates.

Each of the generated examples negates matching constraints on the `floor` and `ceiling` relations. This is how `(functional, ceiling)`, `(total, ceiling)`, `(injective, ceiling)` and `(surjective, ceiling)`, and the matching constraints for `floor` and `ceiling+floor` are learned to be essential constraints.

Although the model is quite simple, this symmetry is actually quite difficult for the algorithm to handle. When a mutated model generates an excluded instance, it means the negated constraint is essential. If more than one constraint is negated, MBE can draw no

conclusion about which is the essential constraint. However, MBE does extra detective work—removing the matching constraints instead of negating them—to determine that the matching constraints are redundant with the negated one (section 3.2.2). MBE cannot always assume that negated constraints are redundant with each other. In particular, although it can recognize matching predicates and matching relations within expressions, it cannot assume that the constraint with a complex relation expression is not an overconstraint. That is, it could not assume that floor and ceiling interacted on an essential property. In complex models, although some relations do interact, others do not.

MBE does not generate an example for one pair of constraints, (irreflexive, ceiling) and (irreflexive, floor). Although $M_{(\text{irreflexive}, \text{ceiling})}$ would generate an example, the prototypical example also satisfies $M_{(\text{irreflexive}, \text{ceiling})}$. MBE first checks if any existing examples satisfy a mutated model. There is a quick way to determine that (irreflexive, ceiling) and (irreflexive, floor) are implied by the other constraints in the model, which after round one are all essential.

Since all redundant and essential constraints have been determined, MBE returns the goal model (figure 4-9) in 3.72s, which contains only 8 constraints.

Although the initial model contained no overconstraints, determining that it contained no overconstraints required involved detective work. The final step would be to check for even more redundancy. There are two most reduced models for ceiling and floor that partition the 8 constraint goal model in half.

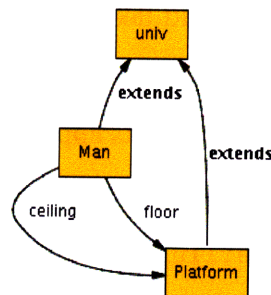


Figure 4-5: Doubly linked list object model

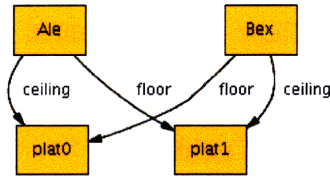


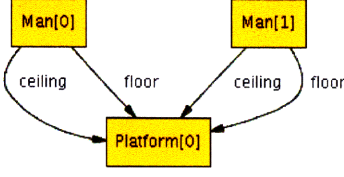
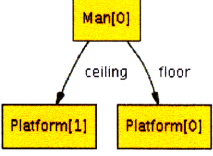
Figure 4-6: Good doubly linked list prototypical example

Table 4.4: Instances generated by MBE in the first round of interaction

Reason why instance is included or excluded by goal model	Instance
Excluded: A man cannot have multiple ceilings or floors (floor and ceiling are not functional)	<pre> graph TD Man0[Man[0]] -- ceiling --> Plat0[Platform[0]] Man0 -- floor --> Plat1[Platform[1]] Man0 -- ceiling --> Plat1 Man0 -- floor --> Plat0 </pre>
Excluded: A man cannot have no ceiling or floor (floor and ceiling are not total)	<pre> graph TD Man1[Man1] -- ceiling --> Platform[Platform] Platform -- floor --> Man1 Man0[Man0] </pre>

Continued on Next Page...

Table 4.4 – Continued

Reason why instance is included or excluded by goal model	Instance
<p>Excluded: Multiple men cannot share the same ceiling or floor (floor and ceiling are not injective)</p>	 <pre> graph TD Man0[Man[0]] -- ceiling --> Platform0[Platform[0]] Man0 -- floor --> Platform0 Man1[Man[1]] -- ceiling --> Platform0 Man1 -- floor --> Platform0 </pre>
<p>Excluded: One man's ceiling is not another man's floor (floor and ceiling are not surjective)</p>	 <pre> graph TD Man0[Man[0]] -- ceiling --> Platform1[Platform[1]] Man0 -- floor --> Platform0[Platform[0]] </pre>

4.3 Trivial File System

The simplest file system is has a single file system object, a directory, capable of containing other directories. MBE applies 16 predicates to the trivial file system object model (figure 4-10) to generate 18 constraints. Using the prototypical example (figure 4-11), MBE extracts an initial model with 8 constraints (figure 4-13). Together these steps take 1.41s.

MBE generates 4 examples (table 4.5) in the first round of interaction, taking 4.34s. The first example negates (rootedOne, contains), which generates an excluded instance, indicating that (rootedOne, contains) is essential. Since (rootedOne, contains) implies (weaklyConnected, contains), (weaklyConnected, contains) is removed from the

(irreflexive, ceiling) \wedge
 (functional, ceiling) \wedge
 (total, ceiling) \wedge
 (injective, ceiling) \wedge
 (surjective, ceiling) \wedge
 (irreflexive, floor) \wedge
 (functional, floor) \wedge
 (total, floor) \wedge
 (injective, floor) \wedge
 (surjective, floor) \wedge
 (irreflexive, ceiling + floor) \wedge
 (total, ceiling + floor) \wedge
 (surjective, ceiling + floor)

Figure 4-7: Initial model extracted from ceiling and floor prototype (figure 4-6)

(functional, ceiling) \wedge
 (total, ceiling) \wedge
 (injective, ceiling) \wedge
 (surjective, ceiling) \wedge
 (functional, floor) \wedge
 (total, floor) \wedge
 (injective, floor) \wedge
 (surjective, floor)

Figure 4-8: Ceiling and floor goal model constructed by MBE

- (total, ceiling) \wedge
 (injective, ceiling) \wedge
 (total, floor) \wedge
 (a) (injective, floor)
- (surjective, ceiling) \wedge
 (functional, ceiling) \wedge
 (surjective, floor) \wedge
 (b) (functional, floor)

Figure 4-9: Two reduced ceiling and floor goal models

working model.

The second and third examples are included, indicating that (functional, contains) and (transitive, contains) are overconstraints. Had a better prototypical example been provided (figure 4-12), these two constraints would not have been extracted into the initial model.

The fourth example in the first round of generalization negates (acyclic, contains). Since the generated example is excluded, (acyclic, contains) is essential. Both of the implied constraints, (antisymmetric, contains) and (irreflexive, contains), are removed from the working model.

Of the remaining three constraints in the working model (acyclic, contains) and (rootedOne, contain) are essential and (injective, contains) is unclassified. MBE negates (injective, contains) in the second round of interaction and generates and excluded example (table 4.9) in 2.26s.

Since all essential, redundant and overconstraints have been identified, MBE returns the goal model (figure 4-14).

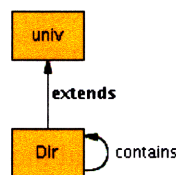


Figure 4-10: Trivial file system object model



Figure 4-11: Trivial file system prototypical example

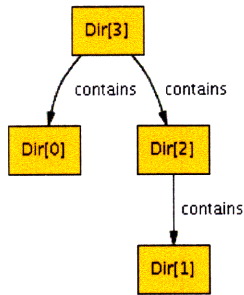


Figure 4-12: Better trivial file system prototypical example

Table 4.5: Instances generated by MBE in the first round of interaction

Reason why instance is included or excluded by goal model	Instance
Excluded: no root directory	[example with no elements]
Included: Directories may contain multiple directories	

Continued on Next Page...

Table 4.5 – Continued

Reason why instance is included or excluded by goal model	Instance
Included: contains is not transitive	<pre> graph TD D2[Dir[2]] -- contains --> D1[Dir[1]] D1 -- contains --> D0[Dir[0]] D0 -- contains --> D3[Dir[3]] </pre>
Excluded: contains is not acyclic	<pre> graph TD D0[Dir[0]] -- contains --> D0 </pre>

Table 4.6: Instances generated by MBE in the second round of interaction

Reason why instance is included or excluded by goal model	Instance
Excluded: contains is not injective	<pre> graph TD D1[Dir[1]] -- contains --> D0[Dir[0]] D1 -- contains --> D2[Dir[2]] D0 -- contains --> D2 </pre>

(injective, contains)
 (transitive, contains)
 (antisymmetric, contains) \wedge
 (acyclic, contains) \wedge
 (functional, contains) \wedge
 (weaklyConnected, contains) \wedge
 (rootedOne, contains) \wedge
 (irreflexive, contains)

Figure 4-13: Initial model extracted from trivial file system prototype (figure 4-11)

(injective, contains) \wedge
 (acyclic, contains) \wedge
 (rootedOne, contains)

Figure 4-14: Trivial file system goal model constructed by MBE

4.4 Medium Complexity File System

The file system model from section 2.2 has constraints with relation expression on two relations, thus we consider it to be a medium complexity model. MBE applies 16 predicates to the file system object model (figure 2-2) to generate 38 constraints in 2.52s. Using the prototypical example (figure 2-5), MBE extracts an initial model with 18 constraints in 7.57s.

After the user marks all 10 instances, MBE refines its internal model and, in 22.47s, generates three instances in the second interaction round (table 2.2). MBE generates instances that ask more complex questions in later rounds. The second instance in round two queries the user on whether directories can be aliased. The users marks this instance as included, permitting `Dir` to share names.

After the user marks all 3 instances in the second interaction round, MBE generates, in 11.41s, one more instance (table 2.3). MBE refines its model and then reaches a termination condition in 5.40s. The total running time of the algorithm, excluding the time it takes the user to mark examples, is 1min:27.53s. All boundary aspects of its internal model have been corrected and verified; thus, it returns the textual goal model it constructed (figure 2-6).

```
(functional, contents)
(functional, name)
(inner;injective, ternary[entries, name])
(injective, entries)
(surjective, entries)
(rootedOne, tternary[entries, contents])
(acyclic, ternary[entries, contents])
(total, contents)
(total, name)
(surjective, name)
```

Figure 4-15: Medium complexity file system goal model constructed by MBE

4.5 Full Complexity File System

The extended file system model has 4 interacting relations and 6 types, with 4 types involved in a 3-level type hierarchy. MBE applies 16 predicates to the file system object model (figure 2-2) to generate 88 constraints in 20.17s. Using the prototypical example (figure 2-5), MBE extracts an initial model with 28 constraints in 50.35s.

Although the initial model looks only slightly bigger than the medium complexity file system model, it is too complex for MBE to solve. In the previous models, the constraints in the goal model are present in the initial model; in this model, the constraints in the goal model (figure 4-19) are variations on the constraints in the initial model (figure 4-18). In particular, we use dynamic constraint construction so that specific type expressions and domain and range restriction are only constructed when constraints with general type and relation expressions are found to be overconstraints (section 3.2.1). Otherwise, 197 constraints would be generated and the initial model would have 137 constraints.

A general constraint uses the root type for type expressions and domain and range restriction. In the complex file system model there are 4 possible subtypes: `Dir`; `File`; `Object-Root`; and `Dir-Root`. These subtypes can be applied both to the type expression, e.g.,

$$(\text{surjective}, \text{contents}, \text{Object} - \text{Root}), \quad (4.1)$$

and restriction, e.g.,

$$(\text{surjective}, \text{Object} - \text{Root} <: \text{contents}). \quad (4.2)$$

Constraint 4.1 means that only `Object - Root` elements have `surjective contents`. Constraint 4.2 means that all `Object` elements have `surjective contents` when only tuples with `Object - Root` domain are considered.

Unfortunately, all of the general constraints are overconstraints. Since dynamic constraint construction occurs while determining implication antecedents, all of the 137 potential constraints are part of at least one antecedent guess. Whereas in previous models only a single example was need to show whether a property applied to a relation, now up to

125 examples may be necessary since there are 5 possible type expressions in each of the three type expression locations in a constraint. Some combinations are very unlikely, e.g., restricting both the domain and the range, or restricting the range on the `total` predicate. Nonetheless, the goal model demonstrates that all three locations are important, and finding that particular, but essential, combination requires generating several more constraints than without subtypes.

Furthermore, the model requires adding two additional predicates, `never_surjective` and `never_total`. `never_surjective` means that all elements in the range map to zero elements in the domain, which generates the constraint

$$(\text{never_surjective, contents } :> \text{Root, Root}),$$

which prevents other directories from containing root. `never_total` means that all elements in the domain map to zero elements in the range, which generates the constraint

$$(\text{never_total, Root } <: \text{parent, Root}),$$

which prevents the root from having a parent. These additional predicates are required in order to ensure that the root of the file system be a `Root` element and not some other `Dir` that happens to exhibit those properties. Since `never_surjective` and `never_total` conflict with `surjective` and `total`, they are not included in the initial model but instead added during dynamic constraint generation.

MBE generates 3 examples in 50.35s in the first round of interaction. In the second round it generates 3 more examples in 73.97s. In the third round it generates 1 example in 72.19s. MBE continues to spend several minutes per round generating 1 or 2 examples. These examples are all incorrect antecedent guesses, and thus obviously (and frustratingly) excluded. Since excluded examples from antecedent guesses teach MBE nothing, no progress is made (section 3.2.2). It may be that MBE is capable of learning the complex file system model if given enough time, or there may be a problem with the algorithm. However, since we know that a possible goal model contains variations on the constraints in the initial model, we suspect that MBE's problem is due to scaling.

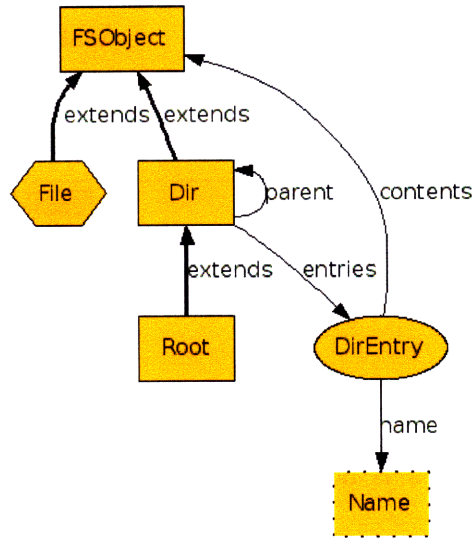


Figure 4-16: Complex file system object model

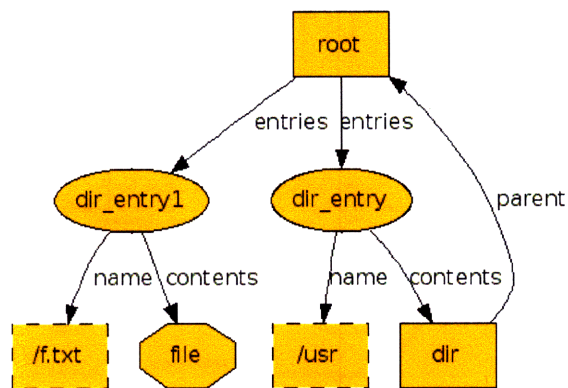


Figure 4-17: Complex file system prototypical example

Table 4.7: Instances generated by MBE in the first round of interaction

Reason why instance is included or excluded by goal model	Instance
Included: contents is not total	<pre> graph TD Root0[Root[0]] -- entries --> DirEntry0[DirEntry[0]] DirEntry0 -- name --> Name0[Name[0]] </pre>
Excluded: entries.contents is cyclic	<pre> graph TD DirEntry0[DirEntry[0]] -- contents --> Root0[Root[0]] Root0 -- entries --> DirEntry0 Root0 -- entries --> DirEntry1[DirEntry[1]] DirEntry1 -- name --> Name1[Name[1]] DirEntry1 -- contents --> Dir0[Dir[0]] Dir0 -- parent --> Root0 Dir0 -- entries --> DirEntry2[DirEntry[2]] DirEntry2 -- name --> Name0[Name[0]] DirEntry0 -- entries --> Name2[Name[2]] </pre>

Continued on Next Page...

Table 4.7 – Continued

Reason why instance is included or excluded by goal model	Instance
<p>Included: <code>entries.contents</code> is not transitive</p>	

Table 4.8: Instances generated by MBE in the second round of interaction

Reason why instance is included or excluded by goal model	Instance
<p>Excluded: <code>Root</code> contains itself</p>	

Continued on Next Page...

Table 4.8 – Continued

Reason why instance is included or excluded by goal model	Instance
Excluded: contents is not total	<pre> graph TD DE0[DirEntry[0]] -- contents --> R0[Root[0]] DE0 -- entries --> R1[Root[1]] R0 -- name --> DE1[DirEntry[1]] R0 -- entries --> DE2[DirEntry[2]] R0 -- entries --> DE3[DirEntry[3]] DE1 -- name --> N2[Name[2]] DE2 -- name --> N1[Name[1]] DE3 -- name --> N0[Name[0]] </pre>
Excluded: parent has wrong direction	<pre> graph TD D0[Dir[0]] -- entries --> DE0[DirEntry[0]] DE0 -- name --> N3[Name[3]] DE0 -- contents --> R0[Root[0]] R0 -- entries --> DE1[DirEntry[1]] R0 -- entries --> DE2[DirEntry[2]] R0 -- entries --> DE3[DirEntry[3]] DE1 -- name --> N2[Name[2]] DE2 -- name --> N1[Name[1]] DE3 -- name --> N0[Name[0]] </pre>

Table 4.9: Instances generated by MBE in the third round of interaction

Reason why instance is included or excluded by goal model	Instance
<p>Excluded: <code>DirEntry</code> not injective</p>	<pre> graph TD DE0[DirEntry[0]] -- contents --> R0[Root[0]] DE0 -- name --> N1[Name[1]] R0 -- parent --> R1[Root[1]] R1 -- parent --> DE0 R0 -- entries --> DE1[DirEntry[1]] R1 -- entries --> DE1 DE1 -- name --> N0[Name[0]] </pre>

(acyclic, entries.contents)
(acyclic, parent)
(functional, contents)
(functional, name)
(functional, parent)
(injective, contents)
(injective, entries)
(injective, name)
(irreflexive, contents)
(irreflexive, entries)
(irreflexive, name)
(irreflexive, parent)
(rootedOne, entries.contents)
(rootedOne, parent)
(rootedOne, ternary[entries, contents])
(inner_injective, ternary[entries, contents])
(inner_injective, ternary[entries, name])
(surjective, contents)
(surjective, entries)
(surjective, names)
(surjective, parent)
(symmetric, parent + entries.contents)
(total, contents)
(total, entries)
(total, name)
(weaklyConnected, parent)

Figure 4-18: Fully complex file system initial model constructed by MBE

(acyclic, entries.contents :> Dir, Dir)
 (acyclic, parent)
 (functional, contents)
 (functional, names)
 (functional, parent)
 (injective, contents :> Dir, Dir)
 (injective, entries)
 (injective, names)
 (never_surjective, contents :> Root, Root)
 (never_total, Root <: parent, Root)
 (rootedOne, entries.contents)
 (inner_injective, ternary[entries, contents :> Dir])
 (inner_injective, ternary[entries, names])
 (surjective, contents :> (Object - Root), Object - Root)
 (surjective, entries)
 (symmetric, parent + entries.contents :> Dir)
 (total, (Dir - Root) <: parent, Dir - Root)
 (total, contents)
 (total, names)

Figure 4-19: Possible goal model for fully complex file system

Chapter 5

Related Work

5.1 Examples and learning

Research on learning has shown that examples are a critical component of learning new skills [33] [32] [44] [26]. People not only prefer to learn from concrete examples over formulas when given a choice [34] [26], but also learn faster and with more comprehension because examples are easier to keep in working-memory and more motivating than abstract representations [6] [33] [44]. Why and how examples enhance human learning is not clear, but human inspired artificial intelligence often performs better than computational methods when humans are to interact with the process and understand the end result.

MBE uses examples in two ways. First, the user provides a prototypical example to initiate the learning algorithm. Representing a formula by a prototype was first introduced in Marvin Minsky's seminal paper on knowledge representation and "frames" [28]. A frame encodes chunks of information (i.e., situations and objects) by representing a single prototype.

The second use of examples in MBE is for generalization of the initial formula. One competing theory for how humans learn via examples is that problem solving rules are generalized from examples [27] [40]. Rules or heuristics are used to relax or restrain a formula based on positive and negative examples. example-based learning is a well-known learning technique in the machine learning community with a plethora of algorithms for combatting overfitting, including example-based generalization [25] [3] [10] [30] [15] [29],

relevance-based learning [39] [2] [35] and knowledge-based inductive learning or inductive logic programming [41] [38] [24] [31].

5.2 Prototypical concept description

Prototypical concept description is a learning algorithm that partitions supervised training datasets into categories from which prototypical examples, which are not necessarily members of the training dataset, are extracted [9]. New examples are categorized by finding the closest matching prototypical example based on underlying concepts. Overfitting is partially mitigated by the continuous example space between categories, but largely relies on inputting a sufficiently varied example space. MBE's learning algorithm guarantees no overfitting because it forces a varied example space by dynamically generating its training dataset in the generalization phase.

In the prototypical concept view, MBE would first collect included and excluded examples, and then extract prototypical examples for both categories. Since the correct formula must be learned without generating too many examples, determining the right similarity function, in terms of constraints, is critical for relaxing overfit constraints without including false positives. Unfortunately, weighting constraints on how likely they are to be overconstraints versus essential constraints relies on subjective heuristics, and thus MBE may incorrectly learn complex or unusual formulas. Instead, user interaction permits MBE to detect and correct errors in the initial formula during the generalization phase. By tapping into information (the prototypical example) easily provided by the user, more onerous information (marking included and excluded examples) is reduced.

In addition to simple models, prototypical concept description learns categories that are a disjunction of prototypical examples [8]. We, too, propose using multiple prototypical examples to express disjunction. In order to learn the multiple prototypes for a particular category, [8] uses k-means clustering. If MBE were to determine prototypes on its own, then it could use a similarity metric on examples in place of a distance metric to determine when to increase the prototypical examples in the formula. By guessing different k, one can find the minimum number of prototypical examples for accurately formulaing a category.

Unfortunately, comparing examples is non-trivial. Instead, MBE would rely on making logical deductions based on the formulas that satisfy marked examples (section 3.3.2).

5.3 Dynamic invariant detection

Daikon, a tool for dynamic invariant detection, learns program invariants from program executions by instrumenting the target program to trace the variables of interest [16]. Dynamic invariant detection and MBE face similar challenges in three areas: generating domain-specific constraints from generic properties; detecting false positives (overconstraints) and redundant invariants; and increasing the tool’s expressiveness without significantly increasing the cost.

Like MBE, Daikon generates program-specific properties (like MBE’s constraints) from templates of basic properties. Initially, Daikon assumes all properties are true, which is an enormous state-space. However, false invariants are falsified quickly. Similarly, MBE extracts an initial formula from the prototypical example, thereby quickly reducing the number of constraints under consideration in the generalization phase. Nonetheless, increasing the number of properties (or potential invariants) is likely to increase false positives and redundant invariants without similarly increasing the number of expressible formulas (or interesting invariants found) [11]. MBE carefully constructs possible constraints since most generated constraints are redundant and increase the time of the learning algorithm. In both cases, knowledge about the kinds of formula constraints or program invariants likely to be interesting, combined with removing redundant and trivial generated constraints before learning begins, mitigates the problem.

The primary difference between these algorithms is that MBE actively detects false positives, while Daikon, which is at the mercy of the provided test suite for exercising all program behavior. Thus, MBE is concerned less with reducing obfuscation of interesting constraints and more with reducing the cost of learning. Daikon detects overconstraints by statistically determining whether the invariant could easily have occurred by chance. This approach requires a large training dataset, and MBE explicitly seeks a minimal test suite to reason over instead of computing statistical likelihood.

Doodoo et al. expand the grammar of Daikon by selecting predicates for conditional invariants [11]. Since it is infeasible to check $a \implies b$ for every a and b , the analysis restricts which a are checked. Daikon will return as many interesting invariants as are expressible by the language and grammar; restricting either component will reduce how much useful information is returned by the analysis. On the other hand, MBE either finds the desired formula or not, so restricting the language or grammar is a significant trade-off against the utility of the tool. MBE can be selective by ordering potential antecedents, however in the worst case all possible subsets must be tested. Thus, MBE may rely on heuristics for efficiency instead of expressibility.

5.4 Mutation testing

Mutation testing is a well-known technique for evaluating a program's test suite [20] [21] [5]. A program is seeded with faults, which creates mutant versions of the program. A test case that distinguishes between the original program and the mutant is said to "kill" the mutant. A test suite that kills all mutants is complete with respect to that mutant set. A major topic of research is how to determine if a mutant is semantically different from the original program.

MBE uses mutation to learn a formula: the generalization phase of MBE's learning process creates n mutant formulas from an initial formula of n constraints, and then tries to distinguish mutants that are real faults, which indicates the mutation of an essential constraint, from mutants that are semantically equivalent to the desired formula, which indicates the mutation of an over-constraint. This process could be generalized into learning specifications for a program.

Of particular interest to us is the mutation of specifications. Although their goal is to generate program test suites, Black et al. mutate SMV specifications [4] and Fabbri et al. mutate Statecharts [17]. The mutations of specifications, as with programs, rely on mutation operators that make slight but reasonable changes to the original text. If MBE were to use mutation operators then the formula it learns would be more like the formula constructed by typical users, allowing the user to switch between textual and pictorial formula

constructing interfaces. This would greatly enlarge the state space of predicates and grammar, and would be probably be impractical unless the user supplied a partially constructed (over-constrained) formula.

Chapter 6

Conclusion and Future Work

Modeling by example is an interesting new technique that relies on the conjunction of predicates to express some of the same models expressible by a first order logic. By altering the underlying predicate language, the learning technique may be applicable to other modeling languages (eg, SMV, Spin), as well. By using active, iterative data collection, MBE minimizes its training data set size and interaction with the user. Based on empirical tests (tables 4.1 and 4.2), MBE correctly learns simple structural models in under 2 minutes, fast enough for users to use MBE in place of conventional modeling techniques.

Unfortunately, MBE has problems scaling, especially when constraints rely on specific subtype expressions. Thus, MBE may not be a practical solution for constructing complex models. The bottleneck occurs in the generalization phase when numerous constraints are redundant with other constraints. The complex file system example (section 4.5) has 2 overlapping groups of interacting relations and 3 subtypes. Although MBE extracts an initial model, MBE learns very little because the large number of generated constraints makes guessing implication antecedents impractical. Although we can manually determine the correct goal model, MBE cannot.

6.1 Future Work

Using domain-specific knowledge, either supplied by the user or based on accumulated statistics, is the most likely way to make MBE scalable. The algorithm's running time can

be improved by directing the overconstraint and implication searches towards likely targets first. Directed search cannot improve the worst case running time, but it can significantly improve the practical running time of the algorithm.

Constraint Tiers

A simple way to direct the search towards likely targets first is to restrict the total number of constraints in tiers. Tiers could restrict predicates, expression generation rules and constraint generation rules. MBE would initially learn using a tightly restricted tier. If the model cannot be learned, MBE tries again using the next tier.

An obvious disadvantage to this approach is that failing on multiple small tiers can take longer than learning on one large tier. This is exacerbated by the observation that constraints are not strictly ordered, and thus a formula that requires a constraint in a very high tier may not need any of the constraints introduced in middle tiers. This is particularly true when determining type expression generation (section 4.5). Whether MBE could mechanically determine why it failed is a difficult learning problem.

User Guidance

Alternatively, MBE could rely on user-guidance. Whether the user can tell MBE which tier to use or why learning failed depends on how much the user knows about the constraint language and how well MBE can deal with misinformation. In addition, the user can indicate structural properties of the model, e.g., `entries` and `contents` form a tree.

MBE's largest problem is finding the antecedents of implications, especially since the guess-and-check algorithm often generates excluded examples. In order to learn from excluded examples, MBE needs to be told why the example is excluded, since it only knows that some constraint in the negated set is essential. The user knows which edges and nodes cause the exclusion, either by their presence or absence. MBE may be able to use the user provided mutated included example to determine the essential constraints.

Incremental Learning

Finally, it may be more efficient to learn the formula gradually or in parts. Complex formulas often have smaller modules that interact in limited ways. Learning each part separately could reduce the number of constraints under consideration without limiting the predicates or constraint generation rules. The challenge is combining formulas without introducing all possible relation and type expressions among constraints in both subformulas. In order to reduce the overall search space, MBE must recognize which constraints to reconsider and which to freeze.

Appendix A

Predicate Library

MBE learns formulas and examples that are directed graphs with labeled nodes and edges. Thus, it is natural to choose graph and relation properties for MBE's predefined predicate set. MBE, which is built on Alloy [23], uses 16 predicates, defined in table A.1 and taken from the Alloy relation and graph utilities modules.

In section 3.2 we show that the learning algorithm's running time depends on finding satisfiable examples when testing the role of each constraint. Since constraints are negated when tested, examples in which negated constraints are implied by other (non-negated) constraints are unsatisfiable. Finding the antecedents in these implications is expensive. Although most implications between constraints depends on the object model, we can prove that some predicates always imply other predicates. MBE uses these predetermined facts to recognize implications between constraints over the same relation, increasing the running time for finding a satisfiable model.

Table A.1: Predicates from Alloy's graph and relation utilities modules

<p>acyclic: no cycles or self-loops</p> <pre> pred acyclic [r :univ->univ , t :univ] { all x:t x !in x.^r } </pre>
<p>antisymmetric: no two-node cycles</p> <pre> pred antisymmetric [r :univ->univ] { ~ r & r in iden } </pre>
<p>complete: all possible tuples exist</p> <pre> pred complete [r :univ->univ , t :univ] { all x,y:t x!=y => (x->y in r && y->x in ~r) } </pre>
<p>functional: all elements in the domain map to at least one element in the range</p> <pre> pred functional [r :univ->univ , t :univ] { all x:t lone x.r } </pre>
<p>injective: all elements in the range are mapped to by at most one element in the domain</p> <pre> pred injective [r :univ->univ , t :univ] { all x:t lone r.x } </pre>
<p>inner injective: equivalent to defining $r = \text{set } A \rightarrow \text{one } B \rightarrow \text{set } C$</p> <pre> pred inner_injective [r :univ->univ] { all x:r.univ.univ injective[x.r, univ.(x.r)] } </pre>

Continued on Next Page...

Table A.1 – Continued

<p>irreflexive: no self-loops</p> <pre> pred irreflexive [r :univ->univ] { no iden & r } </pre>
<p>reflexive: all nodes have self-loops</p> <pre> pred reflexive [r :univ->univ , t :univ] { t <: iden in r } </pre>
<p>rootedAll: all elements in domain can reach all elements in range</p> <pre> pred rootedAll [r :univ->univ , t :univ] { all root:t t in root.*r } </pre>
<p>rootedOne: one element in domain can reach all elements in range</p> <pre> pred rootedOne [r :univ->univ , t :univ] { one root:t t in root.*r } </pre>
<p>stronglyConnected: all elements in domain can reach all elements in range</p> <pre> pred stronglyConnected [r :univ->univ , t :univ] { all d,g:t d != g => d in g.^r } </pre>
<p>surjective: all elements in the range are mapped to by at least one element in the domain</p> <pre> pred surjective [r :univ->univ , t :univ] { all x:t some r.x } </pre>
<p>symmetric: undirectional relations</p> <pre> pred symmetric [r :univ->univ] { ~r in r } </pre>

Continued on Next Page...

Table A.1 – Continued

<p>total: all elements in the domain map to at most one element in the range</p> <pre> pred total [r :univ->univ , t :univ] { all x:t some x.r } </pre>
<p>transitive: every node is directly connected to all reachable nodes</p> <pre> pred transitive [r :univ->univ] { r.r in r } </pre>
<p>weaklyConnected: all elements in domain reach or are reachable by all elements in range</p> <pre> pred weaklyConnected [r :univ->univ , t :univ] { all d,g:t d != g => d in g.^(r + ~ r) } </pre>

The lookup table for predicate implications was mechanically generated. For example, to determine that `acyclic` implies `reflexive` we defined, in Alloy, a simple relation, `r`, over a type, `t`, and checked the assertion that if `r` is `acyclic`, `r` is also `reflexive` (figure A-1). In this case, executing the check finds a counter-example, a single node and empty relation. Thus, `acyclic` does not imply `reflexive`. Since executing the second check finds no counter-example, we conclude that `acyclic` does imply `irreflexive`.

We mechanically generated Alloy code for all 16 predicate pairs, and then the 540 checks were executed. Those that found no counter-example (figure A-2) were identified and given to MBE to use in a lookup table.

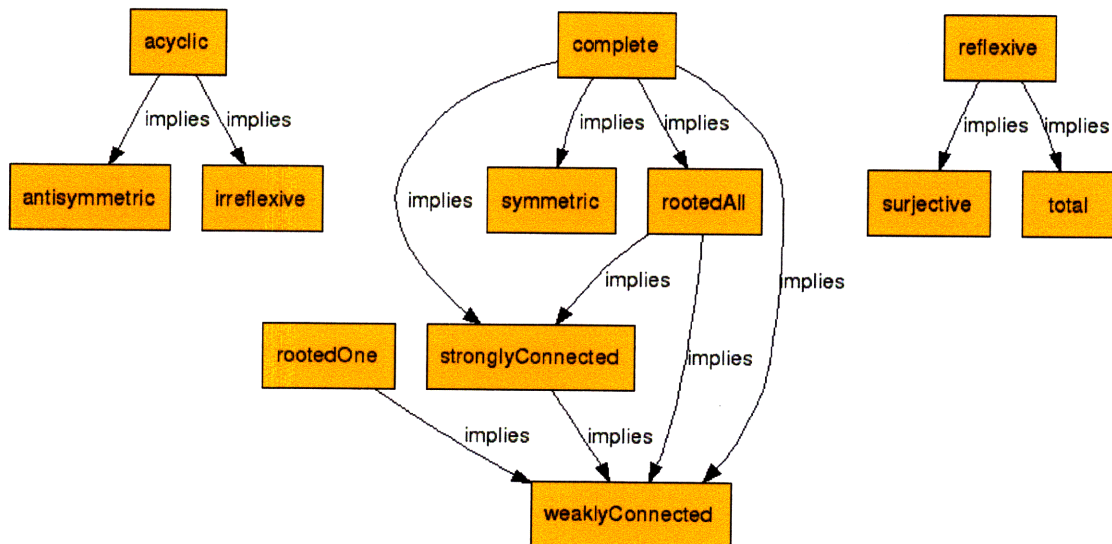
Figure A-1: Alloy code to check whether acyclic implies reflexive or irreflexive

```

sig t { r : set t }
pred acyclic[ r:univ->univ, t:univ ]{ all x:t | x !in x.^r}
pred irreflexive[ r:univ->univ ]{ no iden & r }
pred reflexive[ r:univ->univ, t:univ ]{ t <: iden in r }
check { acyclic[r,t] => reflexive[r,t] } for 5 expect 1
check { acyclic[r,t] => irreflexive[r] } for 5 expect 0

```

Figure A-2: Graph of implications between predicates



Bibliography

- [1] Edward Allen and Waclaw Zalewski. *Shaping Structures: Statics*. John Wiley & Sons, NY, 1998.
- [2] H. Almuallim and T. Dietterich. Learning with many irrelevant features. In *Proc. 9th National Conference on Artificial Intelligence*, volume 2, pages 547–552, Anaheim, CA, 1991. AAAI Press.
- [3] J. R. Anderson. *The architecture of cognition*. 1983.
- [4] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In *Automated Software Engineering*, pages 81–87, 2000.
- [5] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, New York, NY, USA, 1980. ACM Press.
- [6] Michelene T. H. Chi, Nicholas de Leeuw, Mei-Hung Chiu, and Christian LaVancher. Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3):439–477, 1994.
- [7] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499, 1999.

- [8] Piew Datta and Dennis Kibler. Learning symbolic prototypes. In *Proc. 14th International Conference on Machine Learning*, pages 75–82. Morgan Kaufmann, 1997.
- [9] Piew Datta and Dennis F. Kibler. Learning prototypical concept descriptions. In *International Conference on Machine Learning*, pages 158–166, 1995.
- [10] G. DeJong. Generalizations based on explanations. In *Proc. 7th International Joint Conference on Artificial Intelligence*, pages 67–69, Vancouver, British Columbia, 1981. Morgan Kaufmann.
- [11] N. Dodoo, A. Donovan, L. Lin, and M. Ernst. Selecting predicates for implications in program analysis, 2002.
- [12] J. Edwards. *Subtext: uncovering the simplicity of programming*, 2005.
- [13] Jonathan Edwards. Example centric programming. *SIGPLAN Not.*, 39(12):84–91, 2004.
- [14] Gustave Eiffel. Official site of the eiffel tower, July 2007.
- [15] Thomas Ellman. Explanation-based learning: a survey of programs and perspectives. *ACM Comput. Surv.*, 21(2):163–221, 1989.
- [16] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, 2000.
- [17] Sandra Camargo Pinto Ferraz Fabbri, Jose Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. Mutation testing applied to validate specifications based on statecharts. In *ISSRE '99: Proceedings of the 10th International Symposium on Software Reliability Engineering*, page 210, Washington, DC, USA, 1999. IEEE Computer Society.
- [18] Eugene S. Ferguson. *Engineering and the mind's eye*. MIT Press, Cambridge, MA, USA, 1992.
- [19] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

- [20] Mark Harman, Rob Hierons, and Sebastian Danicic. The relationship between program dependence and mutation analysis. pages 5–13, 2001.
- [21] Robert M. Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification & Reliability*, 9(4):233–262, 1999.
- [22] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
- [23] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. pages 62–73. ESEC / SIGSOFT FSE, 2001.
- [24] R. D. King, S. H. Muggleton, R. A. Lewis, and M. J. E. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure activity relationships of trimethoprim analogues binding to dihydrofolate reductase. In *Proceedings of the National Academy of Sciences of the United States of America*, pages 11322–11326, 1992.
- [25] J. E. Laird, P. S. Rosenbloom, and A. Newell. Chunking in soar: The anatomy of a general learning mechanism. 1:11–46, 1986.
- [26] Jo-Anne LeFevre and Peter Dixon. Do written instructions need examples? *Cognition and Instruction*, 3(1):1–30, 1986.
- [27] Clayton Lewis. Why and how to learn why: Analysis-based generalization of procedures. *Cognitive Science: A Multidisciplinary Journal*, 12(2):211–256, 1988.
- [28] Marvin Minsky. A framework for representing knowledge. Technical report, Cambridge, MA, USA, 1974.
- [29] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Mach. Learn.*, 1(1):47–80, 1986.

- [30] Tom M. Mitchell and Sebastian Thrun. Explanation based learning: A comparison of symbolic and neural network approaches. In *International Conference on Machine Learning*, pages 197–204, 1993.
- [31] S. H. Muggleton. Inductive logic programming. pages 295–318, 1991.
- [32] Owen, Elizabeth and Sweller, John. Should problem solving be used as a learning device in mathematics? *Journal for Research in Mathematics Education*, 20(3):322–328, may 1989.
- [33] Peter Pirolli. Effects of examples and their explanations in a lesson n recursion: A production system analysis. *Cognition and Instruction*, 8(3):207–259, 1991.
- [34] Margaret M. Recker and Peter Pirolli. Modeling individual differences in students’ learning strategies. *Journal of the Learning Sciences*, 4(1):1–38, 1995.
- [35] S. J. Russell. Tree-structured bias. In *Proc. 7th National Conference on Artificial Intelligence*, volume 2, pages 641–645, St. Paul, Minnesota, 1988. Morgan Kaufmann.
- [36] Robert Seater. Core extraction and non-example generation: Debugging and understanding logical models. Masters thesis, MIT, Computer Science and Artificial Intelligence Laboratory, November 2004.
- [37] Randall B. Smith and David Ungar. Self: The power of simplicity. Technical report, Mountain View, CA, USA, 1994.
- [38] A. Srinivasan, S. H. Muggleton, R. D. King, and M. J. E. Sternberg. Mutagenesis: Ilp experiments in a non-determinate biological domain. In *Proc. 4th International Workshop on Inductive Logic Programming*, volume 237, pages 217–232, 1994.
- [39] P. Tadepalli. Learning from queries and examples with tree-structured bias. In *Proc. 10th International Workshop on Machine Learning*, pages 322–329, Amherst, MA, 1993. Morgan Kaufmann.
- [40] J. Trafton and B. Reiser. The contributions of studying examples and solving problems to skill acquisition, 1993.

- [41] M. Turcotte, S. H. Muggleton, and M. J. E. Sternberg. Automated discovery of structural signatures of protein fold and function. pages 591–605, 2001.
- [42] Patrick H. Winston. Learning structural descriptions from examples. Technical report, Cambridge, MA, USA, 1970.
- [43] Patrick H. Winston. Artificial intelligence. Technical report, Reading, MA, USA, 1992.
- [44] Xinming Zhu and Herbert A. Simon. Learning mathematics from examples and by doing. *Cognition and Instruction*, 4(3):137–166, 1987.