Pattern-Based Debugging of Declarative Models

Vajih Montaghami and Derek Rayside
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
{vmontagh, drayside}@uwaterloo.ca

Abstract—Pattern-based debugging compares the engineer’s model to a pre-computed library of patterns, and generates discriminating examples that help the engineer decide if the model’s constraints need to be strengthened or weakened. A number of tactics are used to help connect the generated examples to the text of the model. This technique augments existing example/counter-example generators and unsatisfiable core analysis tools, to help the engineer better localize and understand defects caused by complete overconstraint, partial overconstraint, and underconstraint. The technique is applied to localizing, understanding, and fixing a defect in an Alloy model of Dijkstra’s Dining Philosopher’s problem. Automating the search procedure remains as essential future work.

I. INTRODUCTION

Debugging can be a cumbersome and time-consuming task that persists throughout the software lifecycle [11, 22, 23]. Bugs in the specification, whether due to incompleteness or unsoundness, can be especially pernicious [10, 13].

Modern software modelling languages, such as Alloy [8], often have associated analysis tools that assist the engineer in confirming the soundness and completeness of the specification. These tools might include both an example/counter-example generator, and highlighting of the unsatisfiable core of the model when no example or counter-example can be generated. Such tools are a tremendous assistance to model engineers, and our work builds on these them.

An example/counter-example generator helps the engineer understand when the model is under-constrained. The generated example might be one the engineer wishes to reject, and so refines the model to prevent that example. However, common tools take the first available example, which is not necessarily an interesting one nor one the engineer wants to reject. Our technique produces discriminating examples, which help the engineer decide whether a property should be strengthened or weakened (or stay as is).

When the model is fully overconstrained — that is, there are no instances possible — an unsatisfiable core analysis highlights the parts of the model that are in conflict. With high probability, if the model is not intended to be overconstrained, the fix must be made somewhere in the core. But where? Our technique will produce discriminating examples that help the engineer localize which part of the core needs fixing.

Partial overconstraint occurs when some instances that the engineer would accept are prohibited by the model, whereas others are correctly generated. This circumstance is underserved by existing tools. Again, discriminating examples can help the engineer decide if certain properties need to be strengthened or weakened.

The need to debug arises because the expressed meaning of the model differs from the intended meaning of the model, but the engineer does not know where or why [12]. We propose pattern-based debugging as a semi-automatic technique to assist with localization and understanding of these differences. Pattern-based debugging comprises pattern-based semantic inference, semantic mutation, and a collection of tactics.

In this context a pattern is a general idea such as acyclicity. When a pattern is instantiated with respect to a particular relation we call it a property, e.g., r is acyclic. Pattern-based semantic inference is the process of discovering which properties the model implies. The conjunction of these properties is an approximation of the expressed meaning of the model. Semantic mutation is the process of changing this inferred meaning by either strengthening or weakening individual properties. We have computed both an implication graph and a conjunction graph that are used to guide this reasoning process. The implication graph records which patterns strengthen or weaken which other patterns. The conjunction graph records which patterns are mutually satisfiable, and which patterns are in conflict (not mutually satisfiable).

The mutated model is used to produce examples for the engineer to accept or reject. Through this dialogue the mutated model will move closer to the engineer’s intentions.

The engineer is informed of how the original model differs from his or her intentions in three ways. First, the models can be compared directly, to generate discriminating examples that are true in one but not the other. Second, these discriminating examples can be compared with the original model using unsat core. Finally, the mutations themselves can be reported: relax property $p$ to $p'$, strengthen property $q$ to $q'$, etc.

Through the dialogue with the engineer the debugger uses a number of tactics. These are divided into generation tactics, used to generate examples or counter-examples, and correspondence tactics, used to highlight to the engineer where generated examples (or counter-examples) correspond or diverge from the expressed model. The generation tactics include antecedent satisfaction, disjunction vacuity detection, and conjunction vacuity detection. The correspondence tactics include quantifier unrolling, and function and predicate inlining. These tactics will be explained individually as they are used in the presented case study.

The Alloy Analyzer has shipped with a model of Dijkstra’s
dining philosopher’s problem for several years. This model had a partial over-constraint bug that went undetected until 2012. The bug was originally discovered — and fixed — when a translator from Alloy to the KeY theorem proving system was written [21]. However, the depth of reasoning required to fix this bug was not adequately explored in previous publications. We demonstrate that pattern-based debugging can guide the engineer through this complex, multi-step reasoning.

II. PATTERN-BASED SEMANTIC INFERENCE

A pattern is a general concept such as monotonicity, whereas a property is a pattern applied to a specific relation in the model. Pattern-based semantic inference checks which properties, from the cross-product of instantiating every pattern on each relation, are implied by the model. The conjunction of these properties is an approximation of the meaning of the model.

Pattern-based semantic inference is guided by both an implication graph and a conjunction graph of patterns. These graphs are computed once for the library of patterns and then stored for use in analyzing individual models. The implication graph records which patterns imply others, whereas the conjunction graph records which patterns are mutually satisfiable. If pattern \( p \Rightarrow q \), and the model \( m \Rightarrow p[r] \) (that is, the model implies pattern \( p \) applied to relation \( r \)), then there is no need to check \( m \Rightarrow q[r] \). Similarly, if pattern \( p \) and pattern \( x \) are not mutually satisfiable (i.e., if there is no edge between them in the conjunction graph), and the model \( m \Rightarrow p[r] \) then there is no need to check \( m \Rightarrow x[r] \).

III. A LIBRARY OF TERNARY PATTERNS

We present a library of ternary patterns. A previous study of Alloy models showed that ternary relations are often used in temporal models [18]: a third column is added, at either end, to index a binary relation through time. Previously, Mendel [15] developed a library of binary patterns.

The library of patterns is generated from a five-dimensional cross-product (Figure 1). There are 450 patterns produced by this cross-product. An analysis with Alloy reveals that only 180 of these are satisfiable. These can be grouped into 160 equivalence classes (two patterns are in the same equivalence class if they are equisatisfiable). The implication graph for this library has 12 source nodes, 6 sink nodes, and a maximum path length from source to sink of 6. There are some nodes that are reachable from only a subset of the source nodes.

Consider the ternary relation \( r \), with column Left, Middle, and Right: \( \text{sig} \, \text{Left}(r; \text{Middle} \rightarrow \text{Right}) \). Table I shows example values for this relation that conform with four different patterns.

Consider the pattern \( \text{ContractTailOfRight_MiddleStatic} \). We see that \( L_0 \rightarrow M_0 \rightarrow \{ R_0, R_1, R_3 \} \); in the next time step we see that the tail of the right column (atom \( R_0 \)) has been removed (contracted), while the middle column remains the same: \( L_1 \rightarrow M_0 \rightarrow \{ R_1, R_3 \} \). The double ‘t’ on ‘Contract’ indicates that the contraction must happen each time step.

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

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

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

Meta-programming techniques are used to generate the formulas for patterns in this library, as shown below for ExpandHeadOfRight_MiddleStatic. The highlighting shows which parts of the formulas are generated from which concepts in the pattern.

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

The library of formulas is online at https://goo.gl/WhwKiR

IV. DINING PHILOSOPHERS CASE STUDY

The Alloy Analyzer ships with a model of Dijkstra’s dining philosophers problem. The purpose of the model is to show that Dijkstra’s criteria for ordering the mutexes prevents deadlock. Previously, another research group [21] used a theorem prover to discover that the original model (which shipped with Alloy for several years) was over-constrained: deadlock was prevented because no instances were possible. They provided a fix, but did not document the complex and subtle reasoning required to create that fix.

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

```plaintext
1 sig Process {}
2 sig Mutex {}
3 sig State { holds, waits: Process \rightarrow Mutex }
```

The following assertion checks the absence of deadlock:

```plaintext
1 assert DijkstraPreventsDeadlocks { 2 some Process and GrabOrRelease and GrabbedInOrder 3 implies not Deadlock }
```

As will be explained, the root of the problem is that the original GrabbedInOrder predicate is over constrained, so while the assertion about absence of deadlock is true, it is so vacuously:

```plaintext
1 pred GrabbedInOrder { 2 all pre: State \rightarrow so/last | 3 let post = so/next[pre] | 4 let had = Process.(pre.holds), have = Process.(post.holds) |
```

The remainder of this section describes a sequence of four fixes that lead to the final solution, which is logically equivalent (although syntactically different) to the solution previously presented in the literature [21].

A. The First Fix: Removing Overconstraint

Since the absence of deadlock assertion is satisfied, the engineer does not even realize that there is a problem with this model. The debugger sees that the assertion is written as an implication, and applies the Antecedent Satisfaction tactic to generate and check the following assertion. The idea is that if the engineer wrote an implication, they expected the antecedant to be true at least some of the time. The tactic will generate variants of the following assertion, potentially adding existence conditions such as some holds. In this model, holds is the most frequently mentioned relation. Again, if the engineer put a relation in the model, they probably intend that it has the possibility of being non-empty.

```plaintext
1 pred DijkstraPreventsDeadlocksAntecedent { 2 some Process and GrabOrRelease and 3 GrabbedInOrder and some holds }
```

The unsatisfiable core for this assertion reveals that GrabOrRelease and GrabbedInOrder are in conflict. Which one needs to be fixed? Pattern-based semantic inference discovers that there is a property, ExpandRight_FirstLeftEmpty, that is implied by GrabOrRelease and is inconsistent with GrabbedInOrder. This property is used to generate a discriminating example:

```plaintext

holds waits

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$P_0$</td>
<td>$M_0$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$P_1$</td>
<td>$M_1$</td>
<td>$S_1$</td>
</tr>
</tbody>
</table>
```

The engineer accepts this example, which localizes the problem to GrabbedInOrder. Had the engineer rejected this example then the problem would have been localized to GrabOrRelease.

The debugger now applies the quantifier unrolling tactic to GrabbedInOrder, which has a universal quantifier over a set of states. The unsat core of the unrolled GrabbedInOrder with the discriminating example further localizes the problem: GrabbedInOrder is inconsistent with the first state of the discriminating example.

GrabbedInOrder comprises a single implication. For that implication to be in conflict with the discriminating example it must be the case that the antecedent is satisfied and the consequent is where the conflict is. So the problem has been localized to the following expression:

```plaintext
grabbed in mo/nexts[had]
```

The debugger also reports to the engineer that had is empty. Now the engineer realizes that the problem is that the nexts function will not return any successors for the empty set (had).

The engineer replaces this faulty expression with a conditional that returns all mutexes when had is empty and mo/nexts[had] otherwise.
The debugger attempts to produce discriminating examples to help the engineer decide if the model is a faithful expression of his or her intent. The debugger localizes the problem to, again, the consequent in `GrabbedInOrder_Fixed_1`. It is able to do this because it knows the property that is too weak, what it needs to be strengthened to, and how these relate to parts of the model in terms of the implication and conjunction graphs.

Here the engineer needs to think of the fix: how can the expression `grabbed in mo/nexths[had]` permit this example? The problem is that `had` refers to a set that includes `M_0`, and `M_1` is in the `nexths` of `M_0`. The engineer decides to rewrite this expression in terms of `prevs` instead of `nexths`:

```
1 pred GrabbedInOrder_Fixed_2 {  
2   all pre: State − so/last | 
3     let pre = so/first | let post = so/next[pre] | 
4     let had = Process.(pre.holds), have = Process.(post.holds) | 
5     let grabbed = have − had | 
6       (some grabbed) ⇒ (grabbed in (no had implies Mutex 
7       else mo/nexths[had])))  
8 }  
```

The engineer rejects the discriminating example: their intent is that mutexes be acquired in order, and so acquiring `M_1` after `M_2` should be forbidden.

The debugger explores another hypothesis that the model is underconstrained by generating an example to determine if the property `ExpandHeaddOfRight[holds]` should be strengthened to `ExpandHeaddOfRight.MiddleStatic[holds]` by generating an example consistent with the former but not the latter.
The engineer rejects this discriminating example because in state $S_4$, process $P_2$ takes mutex $M_2$ when it already has mutex $M_3$, thereby violating the intention that mutexes are acquired in order. The debugger explains that, from its perspective, the issue is that it changed the middle column (processes) while keeping the right column (mutexes) fixed, and that the syntactic difference between the predicates it is experimenting with is a quantifier over the middle column (see §III).

The engineer realizes that the set have should be defined for each process individually, rather than for all processes collectively, and introduces a quantifier to do this:

```
1  pred GrabbedInOrder_Fixed_3 {
2    all p:Process | all pre: State = so/last |
3      let pre = so/first | let post = so/next[pre] |
4      let had = Process.(pre.holds), have = p.(post.holds) |
5      let grabbed = have = had |
6      (some grabbed) ⇒ (no (grabbed & mo/prevs[pre]))
7  }
```

### D. The Fourth Fix: Strengthening Underconstraint

Unfortunately, the revised model now permits deadlock. Alloy’s regular example generator produces the following instance. The engineer examines this instance, and realizes that it is almost acceptable: if only the last tuple of the waits relation were changed from $S4 → P_0 → M_0$ to $S4 → P_0 → M_2$.

The debugger finds that this corrected example implies $\text{ExpanddHeadOfRight}$ over the union of waits and holds. This union also turns out to be the only non-vacuous combination of these relations: their intersection and differences are empty. This new property is also aligned with the engineer’s intentions on all of the previously generated examples and counter-examples. The property is reported to the engineer, who chooses to simply conjoin it with the existing predicate:

```
1  pred GrabbedInOrder_Fixed_4 |
2    GrabbedInOrder_Fixed_3 |
3      ExpanddHeadOfRight[waits + holds])
```

With this fourth fix deadlock is prevented.

### E. Termination of the Debugger

This fourth fix is equisatisfiable with the previously published solution [21], and so the case study ends here. In addition to this social criterion, the debugger knows that the model prevents deadlock and is consistent with the engineer’s intent on all of the discriminating examples generated so far.

The debugger additionally checks that the fourth fix does not introduce accidental partial overconstraint — such an accident was the original cause for this entire case study, after all. The debugger uses Alloy to confirm that the third model and the fourth model are equivalent so long as the possibility of deadlock is excluded. Therefore, the fourth model is no more overconstrained than the third model.

In this early stage of the research, the debugger search procedure operates by the researchers manually applying tactics. If the search procedure were fully automated, then one might imagine some further checks the debugger might want to perform in addition to the ones described above. For example, the debugger might check that every one-step semantic mutation of the model is killed by one of the discriminating examples that the engineer has already accepted or rejected. The debugger might also check that its example generation tactics — antecedent satisfaction, disjunction vacuity detection, and conjunction vacuity detection — produce no new information. If all of these conditions pass then the engineer could have a fairly high degree of confidence that the model — now including the collection of generated examples — is an accurate expression of his or her intent.

### V. Conclusion and Future Work

Pattern-based debugging has the potential to enhance existing model debugging tools, including example/counter-example generators and unsatisfiable core analyzers. This potential derives from a capacity to create semantically meaningful variants of the engineer’s model, and to explain those variants to the engineer semantically, syntactically, and by example. This process can guide the engineer towards a model that more accurately reflects their intent.

Pattern-based debugging is in the spirit of example-driven modelling: an approach that emphasizes the importance of examples, in addition to abstractions, for creating complete models [1]. A key idea in pattern-based debugging is the discriminating example: an example that helps the engineer decide if a property should be strengthened or weakened. Discriminating examples additionally help to focus unsatisfiable cores on where the fix should be.

This paper reports on the idea of pattern-based debugging, a library of patterns for ternary relations, and a case study of a subtle and complex fix for an Alloy model of Dijkstra’s dining philosophers problem. Pattern-based debugging comprises pattern-based semantic inference, semantic mutation, and a collection of tactics.

Essential future work includes automating the semantic mutation and tactic application search procedures, as well as the tactics themselves. There are also possibilities to expand the library of patterns.
VI. RELATED WORK

There are a variety of existing libraries of specification patterns, including for LTL [5], real-time systems [6, 14], and service-based applications [7, 9, 17]. Closer to our work, Mendel [15] created a library for binary relations.

There are also debuggers and visualizers for LTL and CTL [2, 4, 13]. Some of these make use of Reiter’s general theory of diagnosis [19].

Aluminum has a similar goal of improving example exploration, but is based on mathematical definitions of minimality rather than discriminating between variants of a property [16].

Seater [20] proposed non-examples to help explain the purpose of each constraint in a specification. This is complementary to our idea of discriminating examples.

Pattern-based debugging is inspired by mutation testing [3].

ACKNOWLEDGMENT

The authors thank Steven Stewart for his assistance with the prose, and the reviewers for their thoughtful comments. This work was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES