Debugging Overconstrained Declarative Models Using Unsatisfiable Cores

Ilya Shlyakhter
MIT CSAIL, Cambridge, MA
ilya_shl@mit.edu

Robert Seater
MIT CSAIL, Cambridge, MA
rseater@mit.edu

Daniel Jackson
MIT CSAIL, Cambridge, MA
dnj@lcs.mit.edu

Manu Sridharan
UC Berkeley, Berkeley, CA
manu_s@cs.berkeley.edu

Mana Taghdiri
MIT CSAIL, Cambridge, MA
taghdiri@mit.edu

Abstract

Declarative models, in which conjunction and negation are freely used, are susceptible to unintentional overconstraint. Core extraction *is a new analysis that mitigates this problem in the context of a checker based on reduction to SAT. It exploits a recently developed facility of SAT solvers that provides an "unsatisfiable core" of an unsatisfiable set of clauses, often much smaller than the clause set as a whole. The unsatisfiable core is mapped back into the syntax of the original model, showing the user fragments of the model found to be irrelevant. This information can be a great help in discovering and localizing overconstraint, and in some cases pinpoints it immediately. The construction of the mapping is given for a generalized modelling language, along with a justification of the soundness of the claim that the marked portions of the model are irrelevant. Experiences in applying core extraction to a variety of existing models are discussed.*

1. Introduction

Along with its many benefits, declarative modelling brings the risk of overconstraint. An overconstrained model trivially satisfies safety properties; in the extreme, it has no bad transitions because in fact it has no transitions at all.

The risk of overconstraint in declarative specification languages such as Z [18] and VDM [12] was recognized long ago, but only very limited automatic tool support exists to mitigate it. In Z, preconditions are implicit; it is regarded as good style for an operation's precondition to appear explicitly in the text of the operation's schema. This discipline results in proof obligations (that the explicit conditions imply any implicit preconditions). Checking these obligations is no easier than checking any other property of a Z specification[1]. Similarly, in VDM, the 'implementability' criterion leads to a similar obligation. Because of the difficulty of discharging proof obligations automatically, most tools for Z and VDM simply extract the proof obligations but leave the user to determine their validity.

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

Even when a modeler suspects an overconstraint, identifying the conflicting constraints is often frustrating. Currently, the only systematic technique for finding causes of conflict in a declarative model is to manually disable individual constraints until the culprits are identified. This task can be lengthy and runs the risk of introducing new errors into the model. The model checker provides no help to the user in finding the overconstraint, other than to report whether a given version of the model is still overconstrained. As discussed in Section 5, 'vacuity testing' addresses this problem [15, 1, 3, 20], but does not apply to declarative models and helps debug only overconstrained properties, not overconstrained models.

This paper presents *core extraction*, a new approach to addressing the problem of overconstraints in declarative models. Satisfiability solvers have recently developed a facility for extracting the *unsatisfiable core* of a CNF formula: that is, a subset of the clause set sufficient to cause a contradiction [21, 7]. For declarative model analyses that can be cast as satisfiability instances, the unsatisfiable core can be mapped back onto the model. In other words, we can identify the parts of the model responsible for producing the unsatisfiable CNF core. Those parts, by themselves, suffice to

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

produce an overconstraint, and their identification can help the user find the overconstraint.

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

Our presentation is set in the context of an analysis for first-order relational logic that has the flavor of model checking. In short, a system is specified as a formula, whose models (which correspond to the behaviours of the system) assign values to relations of various arities. A safety property is checked by conjoining its negation to this formula; solutions to this new formula are counterexamples. If there are no solutions, and the model is correct, then the property being checked holds (up to the specified scope). The formula is translated to a propositional formula by bounding the carrier sets from which the relations draw the values of their atoms. Models of the propositional formula are found by a SAT solver, and translated back into the relational domain for display to the user. This analysis scheme has been described previously [10]; until now, if no counterexample were found, no further information would be given. This lack of information has been the a complaint from users of our tool.

Although we developed these techniques in the context of the declarative modeling language Alloy, which we will use to present case studies and examples. However, both the technique and its implementation were intentionally kept much more general; they are sufficiently modular to apply to any language which is reducable to SAT in a structure-preserving fashion [2]. Concequently, our techniques should also apply in related settings such as BMC [2] and planning [13]. We provide a simple semantic guarantee of correctness, assuring the user that deleting constraints identified as irrelevant will preserve unsatisfiability of the model.

The contributions of this paper include:

- A recognition of the problem of overconstraint in declarative models, with discussion of why existing techniques, including vacuity testing and the checking of liveness properties, do not eliminate it;
- A simple but effective method for discovering and localizing overconstraint in the context of checkers based on reduction to SAT, by exploiting the unsatisfied core facility of SAT solvers;
- A justification of the semantic guarantee made by the analysis at the level of the source of the model, for a generalized modelling language; and
- Some reports on preliminary experiments applying an implementation of the method to a variety of models, demonstrating its promise and highlighting challenges for future work.

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

The problem of overconstraint and the benefits of core extraction are illustrated first on a toy example (Section 2). The framework is described for a generic constraint language, along with an argument for its soundness (Section 3). A variety of experiences in using our implementation of core extraction are discussed (Section 4). We compare our technique to *vacuity testing*, which provides similar information to operational modelers (Section 5). For readers unfamiliar with declarative modeling, an appendix explains its relationship to operational modeling, and the circumstances in which it is particularly beneficial.

2. A Toy Example

To give a flavor of declarative modelling, and to illustrate the use of the new analysis, consider the problem of checking the design of a web caching scheme. Our challenge is to design the `Get` operation that obtains a page from its owner, or from a cache. The correctness of `Get` will be contingent on some assumptions about the freshness of pages delivered by the owner; we will record these as an invariant.

A complete (albeit simplified) Alloy model for this problem is shown below:

```
module webcache

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

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

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

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

check Freshness
```

A detailed knowledge of Alloy is not needed to grasp the point of the example, but some explanation is in order. A fuller explanation of Alloy may be found elsewhere [11].

Each signature (labelled `sig`) introduces a set of atoms: `Time` for the atoms representing moments in time, `URL` for the URL's of documents, `Page` for the contents of documents, and `Server` for the caches and owners. A field declared within a signature is simply a relation of some arity, whose columns are the sets given, and implicitly, in the leftmost position, the signature itself. Thus `life` is a binary relation from `Page` to `Time`, associating with each page the set of times for which it is current, and `page` is a relation of arity 4, containing a tuple `(s,t,u,p)` when server `s` maps URL `u` to page `p` at time `t`. The question mark in the declaration of the field `page` indicates that at most one page is associated with a given server, time and URL.

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

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

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

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

The times `t` and `t'` represent the moment just before and just after the `Get` occurs. The function as a whole can be read as follows: the mapping from URL's to pages in the cache after (`cache.page[t']`) is a subset of the union of the mapping before (`cache.page[t]`) and the mapping in the owner (`owner.page[t]`), minus all entries whose pages are no longer fresh (`{u:URL, p:Page | t' not in p.life}`). In other words, the cache is at liberty to drop any entries, and to add any entries from the owner, so long as no stale pages remain afterwards. The fact is the crucial invariant recording the assumption that pages in a server are always fresh.

A counterexample to an assertion is a model of its negation. This model gives values to the set and relation constants. Because our tool skolemizes quantifiers, it gives witnesses too. In this case, for example, if the assertion were not valid, the tool would give values for the two relations `page` and `life`, and witnesses for the particular times `t` and `t'`, cache, owner, page and URL. Note that there is no built-in notion of state machine; the 'states' before and after the operation are obtained by examining how the relations map the times `t` and `t'`.

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

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

To fix the model, we partition the set of servers into caches and owners, modify the fact to constrain only owners, and declare the arguments of `Get` to belong to the appropriate subsets:

```
module webcache

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

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

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

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

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

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

in the first line of `Get`, however, is marked as irrelevant. This makes sense though; since all stale pages are removed from the cache (by the set subtraction that follows), the source of additional pages is irrelevant. That the remaining formulas are deemed relevant gives us some confidence that our model is no longer vacuous; the irrelevance of this particular expression suggests that an assertion about the authenticity of a page is needed.

As another example, suppose we erroneously wrote

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

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

3. The Core Extraction Algorithm

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

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

The correctness criterion for the algorithm is that the claim of irrelevance is sound. More precisely, the function computed at any AST node marked as irrelevant can be replaced by any function that leaves the AST well formed (in particular a constant function of the appropriate type), without making the resulting AST satisfiable. An irrelevant child of an `AND` node, for example, can be read as the constant `true`. In some cases, as here, this means that the node can be removed entirely. A roadmap of how the algorithm fits into the overall use of the tool algorithm is shown in Figure 1.

3.1. Constraint Language

We define an abstract constraint language for expressing formulas on a collection of variables $v_i \in V$. A formula is expressed as an abstract syntax tree, in which each node computes a predefined function of its children. The root node computes a Boolean function, which becomes the value of the formula as a whole. The leaf nodes are variables. We denote the universe of computable values by U.

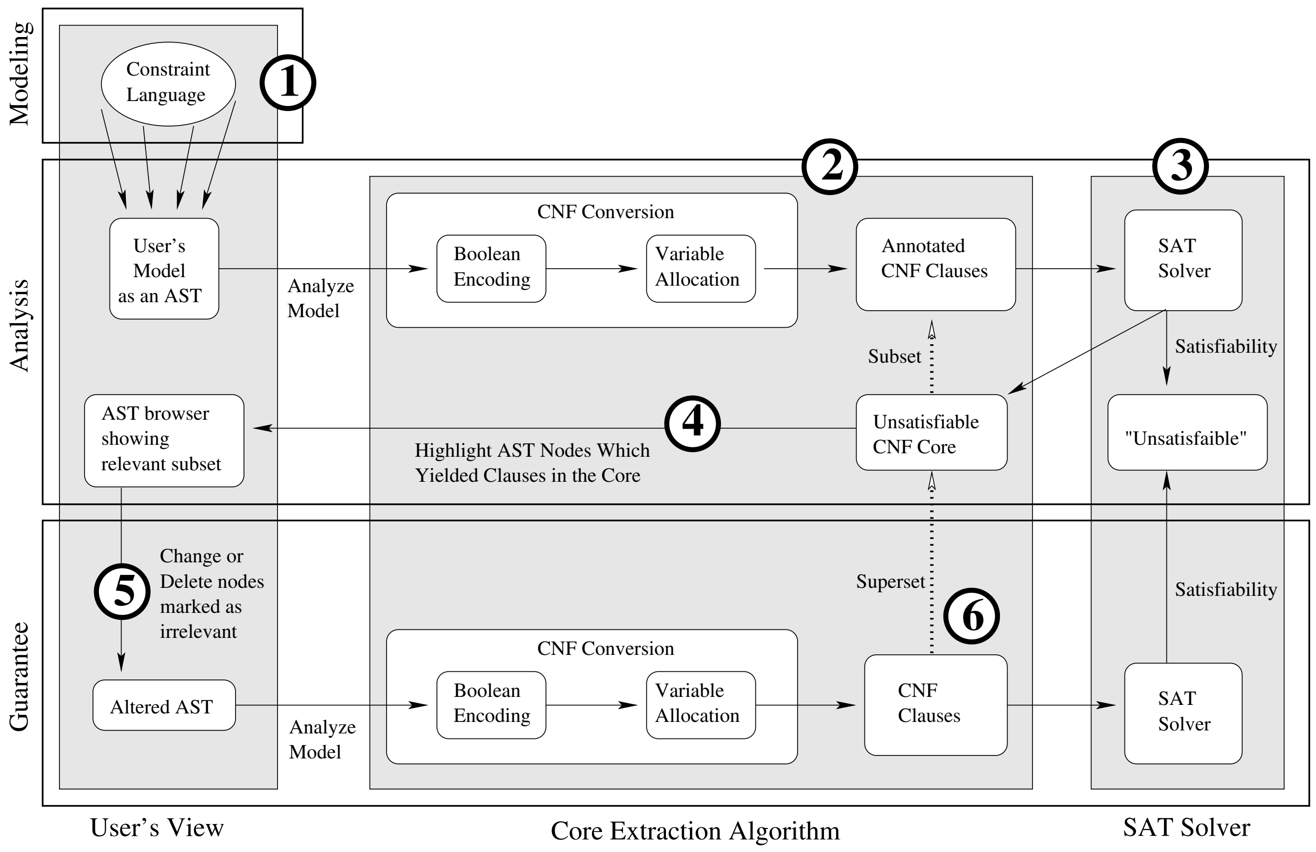

Figure 1. *A Roadmap to Core Extraction.* (1) A model is created in any constraint language which is reducible to SAT in a structure preserving fashion. (2) During translation to CNF, each clause generated is annotated with the AST node from which the clause was produced. (3) A SAT solver (used as a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

The set F of node functions has elements of the form $f_i : U^* \rightarrow U$ Trees are thus defined by $Tree = F \times Tree^* + V$. [3]. An assignment from U to each $v_i \in V$ induces an assignment from U to each AST node; the value of a leaf node is the value of the AST variable at that node, while the value of a node $n = Tree(f, ch)$ is computed by applying the node function f to the sequence of values assigned to the children ch. Testing satisfiability of the AST involves finding an assignment to the variables v_i which induces the value $true$ in the root node, or determining that none exists.

3 The sets U and F are determined by the semantics of the particular constraint language. For example, for Alloy [11], U contains relational and Boolean values, and F includes relational and Boolean operators. The particular semantics are unimportant for this paper.

3.2. Translation

Satisfiability of the formula can be tested by converting it to a Boolean formula in conjunctive normal form (CNF). The translation framework is illustrated in Figure 2.

To convert an AST to CNF, we allocate to each AST node $n \in Tree$ a sequence of Boolean variables $bv(n) \in BV^*$ representing the node's value. The sequences of Boolean variables allocated to two nodes are identical if these are leaf nodes with the same AST variable, otherwise the sequences are disjoint. We define functions $enc : U \to Bool^*$ and $dec : Bool^* \to U$ for encoding and decoding values in U as binary strings. An assignment of Boolean values to all the Boolean variables allocated for AST nodes thus corresponds to assigning a value from U to each AST node. An assignment of U values to AST nodes is consistent if the value at each non-leaf node equals the result of applying the node's node function to the sequence of U values assigned to the node's children. We translate an AST to CNF by generating CNF clauses on the Boolean variables allocated to AST nodes, so that the conjunction of these clauses is true of a given assignment to Boolean variables iff the Boolean assignment corresponds to a consistent assignment of U values to AST nodes.

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

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

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

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

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

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

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

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

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

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

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

3.3. Mapping Back

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

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

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

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

For any node whose node function has not changed, $transl$ will thus output the same clause set. Any node n whose clause set contributed to the core will still have the same node function, and $transl$ will output the same clause set for that node. Each clause of the unsatisfiable core is thus present in the translation of the mutated AST, meaning that the mutated AST is still unsatisfiable.

3.4. Complications

The description of the AST above was made rather abstract to make it clear that although we have implemented the scheme for Alloy, it could be applied straightforwardly to any constraint language that can be reduced to SAT. The description of the translation is likewise more abstract, because this allows it to accommodate more advanced translations. The Alloy Analyzer, for example, identifies opportunities for sharing among subformulas [17], so that the AST is in fact not a tree but a DAG. This can be modelled by having the bv function allocate the same Boolean variables to

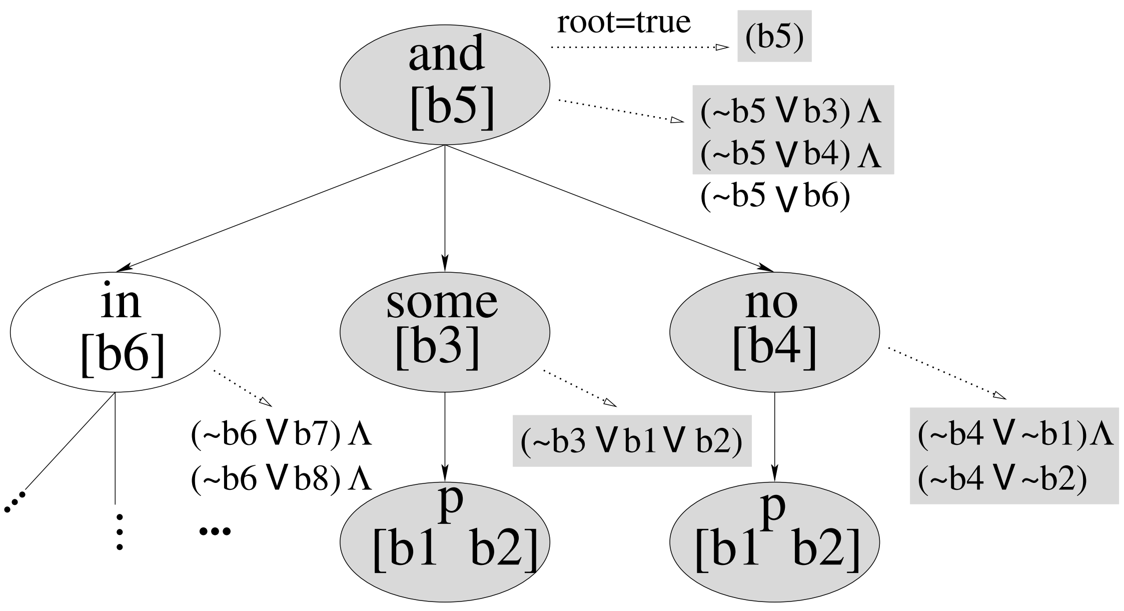

Figure 2. Translation of AST to CNF, and mapping back of unsatisfiable core. The AST is for the (trivially unsatisfiable) Alloy formula of the form "(some p) && (no p) && ...". To each node, a sequence of Boolean variables (b1 through b6) is allocated to represent the node's value. From each inner node, translation produces a set of clauses relating the node's Boolean variables to its childrens' Boolean variables. The highlighted clauses form an unsatisfiable core, which is mapped back to the highlighted AST nodes.

different AST nodes. Similarly, like most tools that generate CNF, the analyzer uses auxiliary Boolean variables to prevent CNF explosion; this can be modeled by having *bv* allocate additional Boolean variables to each AST node.

4. Experience

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

4.1. Common Mistakes

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

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

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

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

to say that each person has a unique name. But this formula is always false unless the set `Person` is empty, since it cannot be satisfied when `p = q`. The extracted core would likely include the property being checked[4], a constraint requiring non-emptiness of `Person`, and this quantified formula. Writing instead

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

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

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

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

A similar blunder is sometimes made in trace-based models in which the modeler constraints transitions between states. We often lift a constraint on state pairs to a constraint on traces so that we can analyze traces using bounded model checking. An example of such a lifting formula is the following:

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

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

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

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

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

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

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

4.2. Locating Known Overconstraints

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

Iolus The more successful case involved an analysis we performed [19] of Iolus, a scheme for secure multicasting [16]. In Iolus, nodes multicast messages to other nodes within a group whose membership changes dynamically. Scalability is achieved by partitioning groups into subgroups, arranged in a tree, each with its own Key Distribution Server (KDS) maintaining a local encryption key shared with members of the subgroup. When a member joins or leaves a subgroup, its KDS generates a new local key and distributes it to the updated list of subgroup members. This was modelled by specifying that after a member joins or leaves, there is a key shared by the new members, and no others. By mistake, the model said the key was shared by the members of the entire *group* – thus including all nodes in contained subgroups. This severely restricted the trace set, potentially masking errors.

We attempted to detect this overconstraint using our constraint core functionality. We first checked an assertion stating that no node can read messages sent to the group when that node was not a group member, one of the correctness properties of the system. There was no counterexample, and unfortunately, the extracted core included most of the constraints in the model. This result can be explained as follows. The error in the model is only a *partial* overconstraint; while the error excludes some legal traces of the system, it still allows many traces violating the correctness property. Therefore, it is not surprising that most of the other constraints in the system are still required to establish correctness. Just because the core contains most of a model does not, unfortunately, imply that the model is free of overconstraint.

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

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

Firewire A model of the widely studied Firewire 'tree identify' protocol [9] suffered from a modelling blunder that produced a nastily subtle overconstraint. The declarative form of the model allowed it to include a topological constraint (that the links between nodes form a connected, acyclic graph), so that analysis would cover all possible topologies involving a given number of nodes. Most model checking analyses, in contrast, hardwire a particular topology.

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

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

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

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

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

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

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

4.3. Blunders Discovered

Running our core extractor revealed flaws in two models that we had believed not to be overconstrained. We explain one of them here.

In a different version of the Firewire tree identify protocol, we had added a stuttering operation at the last minute, but failed to adjust the scopes of the analyses. The declaration of operations read:

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

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

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

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

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

4.4. Performance

For core extraction we have used a recent modification of the Zchaff satisfiability solver that added core extraction functionality [21]. We found that Zchaff's performance supports interactive identification of overconstraints. The modified solver's performance on unsatisfiable instances was comparable to the performance of the original solver. We have also done some experiments with the BerkMin solver [7, 6]; preliminary experiments indicate that BerkMin's performance is similar to Zchaff's.

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

5. Related work

The problem of detecting when a property is vacuously satisfied by a model has been addressed in the context of temporal model checking [15, 1, 3, 20]. Given a temporal logic formula, these methods produce a "witness" formula that is satisfied if and only if the original formula is vacuously satisfied. Thus, vacuous satisfaction can be detected with an additional model checking run. Several characteristics of these methods prevent them from solving the problem of overconstraint in declarative models. First, overconstraint occurs most often in the definition of the model-checked algorithm rather than in the specification of correctness properties. Published vacuity detection methods may alert the user to the presense of an overconstraint (by showing that the entire correctness property is irrelevant), but cannot pinpoint the location of overconstraint within the model. Second, these methods were described for temporal logic formulas, and either assume a particular form of the formula [1] or require a separate model-checking run to test for irrelevance of each subformula [20]. These limitations preclude published vacuity detection methods from being effective on our problem.

6. Conclusions

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

Acknowledgements

This work was supported by NSF ITR grant #0086154 and by a National Defense Science and Engineering Graduate Fellowship.

References

[1] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

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

[3] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. *Lecture Notes in Computer Science*, 2031:528–??, 2001.

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

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

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

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

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

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

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

[11] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, September 2001.

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

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

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

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

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

[17] I. Shlyakhter, M. Sridharan, R. Seater, and D. Jackson. Exploiting subformula sharing in automatic analysis of quantified formulas. http://ilya.cc/sharing.ps, May 2003.

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

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

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

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

Appendix: Declarative Modelling

Roughly speaking, there are two ways to model a transition system. In the *operational* idiom, transitions are expressed using assignment statements, either with the control flow of a conventional imperative program (as in Promela, the language of the Spin model checker [8]), or using a variant of Dijkstra's guarded commands (as in Murphi [5] and SMV [4]). In the *declarative* idiom, transitions are expressed with constraints, either on whole executions, or, more often, on individual steps. This idea is rooted in the early work on program verification; the operation specifications of the declarative languages VDM, Larch and Z are essentially the pre- and post-conditions of Hoare triples.

For readers unfamiliar with these idioms, it may help to think of an operational specification as one that gives a recipe for constructing new states from old ones, and a declarative specification as one that gives a fact that can be observed about the relationship between old and new states. An operational modeller asks 'how would I make X happen?'; a declarative modeller asks 'how would I recognize that X has happened?'.

The advantage of the operational idiom is its executability. A simulation, either random or guided by inputs from the user, can give useful feedback to a designer. In model checking, the ability to generate a state's successor in a single computational step makes it possible to explore the reachable state space by depth-first search (as in Spin). In contrast, declarative models have been viewed as not executable, and less amenable to automatic analysis in general, since even generating successors requires search. Recently, however, we have developed an analysis based on SAT that allows both simulation and systematic exploration of declarative models [10]. A common form of analysis that we perform is similar to bounded model checking [2]: the SAT solver is used to find traces that violate specified properties. In fact, earlier symbolic methods could also handle models with declarative elements. The earliest versions of SMV, for example, provided a construct for expressing transitions implicitly. Its analysis, being symbolic, was not hindered by the inability to generate successors of a state constructively.

The advantage of the declarative idiom is its expressibility. For some kinds of problem, especially the control-intensive aspects of a system, the operational idiom can be more natural and direct. But in many cases, especially for software systems, the declarative idiom is more flexible, more natural, and sometimes, surprisingly, more amenable to analysis.

• *Partial Descriptions*. The declarative idiom better supports partial descriptions. Sometimes, only one operation is of interest. In a study of a name server [14], for example, only the lookup operation was modelled and analyzed, since the operations for storing and distributing name records were straightforward. An explicit invariant on the structure of the name database took the place of the operations that in an operational model would define the reachable states implicitly. Even if the lookup operation were not written declaratively, the need to account for the invariant in generating initial states makes the description essentially declarative.

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

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

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

• *Topology Constraints*. Sometimes one particular aspect of a system has a declarative flavour. For example, many distributed algorithms are designed to work only if the network's topology takes some form, such as a ring or tree. A declarative model can be constructed that constrains the network appropriately, but does not limit it to a single topology. The analysis will then account for all executions over all acceptable topologies (for a network of some bounded size). The Firewire example described in Section 4 exploits this.

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

The very mechanisms that give declarative modelling its power – conjunction and negation – also bring a curse: the risk of overconstraint. It is unfortunately easy to write a model that has fewer behaviours than intended. A check of a safety property may then pass only because the offending behaviour has been accidentally ruled out (probably along with many other behaviours).

The risk can be mitigated by working carefully. One can exploit the ability to build and analyze a model incrementally, adding as few (and as weak) constraints as possible to establish the required safety properties. One can simulate the model extensively, adding conditions to force execution of interesting cases. And of course one can formulate and check liveness properties, at least ruling out the most egregious overconstraints, such as those that lead to deadlock.

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