Finding Minimal Unsatisfiable Cores of Declarative Specifications

Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson

MIT Computer Science and Artificial Intelligence Laboratory
{emina,fschang,dnj}@mit.edu

Abstract. Declarative specifications exhibit a variety of problems, such as inadvertently overconstrained axioms and underconstrained conjectures, that are hard to diagnose with model checking and theorem proving alone. *Recycling core extraction* is a new coverage analysis that pinpoints an irreducible unsatisfiable core of a declarative specification. It is based on resolution refutation proofs generated by *resolution engines*, such as SAT solvers and resolution theorem provers. The extraction algorithm is described, and proved correct, for a generalized specification language with a *regular* translation to the input logic of a resolution engine. It has been implemented for the Alloy language and evaluated on a variety of specifications, with promising results.

1 Introduction

As Dijkstra famously noted, testing can only show the presence of errors and not their absence. Establishing the absence of errors has been a major motivation for more complete analyses, such as model checking and theorem proving. Yet, despite the advantages such analyses often bring in bug-detecting ability, it is not always clear what level of confidence is warranted when no bugs are reported.

The main reason for doubting the result of a successful analysis is simply that the theorem being checked might not be the right one, and might fail to capture the notion of correctness that will actually be required in the context of use. When the artifact being checked is a model (rather than the actual implementation of a system), there is an additional concern that the model may not be faithful to the system it purports to represent.

It may seem that this problem is not amenable to a technical solution. In fact, however, the most common faults in a model or theorem that undermine the credibility of an analysis can be exposed by a kind of 'coverage analysis' that highlights those portions of the model and theorem that were used to establish that the theorem held for the model. Portions that are not highlighted, contrary to the expectations of the user, are evidence that the analysis was inadequate.

This idea has been explored as "vacuity detection" [1, 2] in the context of model checking, although the very definition of the problem is somewhat intricate. In the context of checking declarative specifications (as written in languages such as Alloy, Z, VDM, B, OCL, and so on), the notion of coverage has a particularly simple formulation. A constraint, whether occurring in the model

J. Cuellar and T. Maibaum (Eds.): FM 2008, LNCS 5014, pp. 326–341, 2008.

being checked or in the theorem being asserted, is covered (and subsequently highlighted) if it was used in the proof that the theorem follows from the model.

This approach has been implemented as a feature of the Alloy Analyzer [3,4], but until recently has not been particularly useful since the highlighting has been too conservative, often including constraints that were not in fact used. This paper presents a new algorithm, RCE, that has been incorporated into the tool, and which gives superior results. RCE is proven to give results that are sound (meaning that constraints that are not highlighted are definitely irrelevant) and minimal (meaning that removing the highlighting on a constraint would make the result unsound). Its performance is compared to three simpler algorithms: OCE, the one previously implemented in the Alloy Analyzer, which runs faster than RCE but is not minimal, and typically highlights 2 to 3 times as many constraints; and NCE and SCE, which are sound and minimal, but run much more slowly than RCE.

As illustrated in the next section, coverage analysis mitigates a variety of problems that can arise in practice: inadvertently overconstraining the model (so that behaviours that should be included are de facto excluded); using a theorem that is not strong enough (so that bad behaviours are accepted); and setting the analysis bounds too small, so that the analyzer does not examine a sufficiently large space of possibilities. This last problem is a liability only of checkers (such as the Alloy Analyzer) that artificially bound the space, and is not suffered by theorem provers. Nevertheless, provers do suffer from the other two problems, and the algorithm presented here will therefore work for them too.

The underlying mechanism used is *unsat core extraction*, a facility of some SAT solvers. The core of an unsatisfiable formula (presented in CNF as a set of boolean clauses) is a subset of the formula that is also unsatisfiable. Every unsatisfiable formula is its own core, but a smaller core is more useful. SAT solvers do not generally provide minimal cores, which would require too much computation to produce.

Exploiting an unsat core facility is not straightforward, however, since the core returned by the SAT solver must be translated back into the high-level specification language before being shown to the user. Efficient compilations into SAT employ a variety of elaborations and transformations that result in a complex relationship between the original specification formula and the boolean formula passed to the solver. Consequently, a small core at the boolean level may be translated back to a large core at the specification level.

The new algorithm has two key ideas. The first idea is that, rather than attempting to minimize the core at the boolean level, to map the core back and apply reductions (by testing the removal of candidate constraints) at the specification level. The second idea is to identify, using the proof returned by the solver, and the mapping between levels, those boolean clauses that were generated during a proof of unsatisfiability, and which will still hold when a specification-level constraint is removed. By adding these clauses to the formula presented to the SAT solver, the algorithm allows the solver to reuse the results

of inferences that were previously made. It is well known that careful exploitation of learned clauses is essential for improving SAT solver performance in general, so it is not surprising that it plays an important role in this application also.

Although the scheme was developed for analyzing coverage of Alloy specifications that are translated to boolean formulas, it has more general applicability. The paper therefore defines the context rather abstractly. The source language can take any form so long as its translation to the target language satisfies some basic properties that the paper defines. The target language can be any clausal language, and any prover is suitable if it can return a proof as a resolution graph.

2 A Small Example

As a motivating example, consider the problem of formalizing a key ingredient in our core extraction algorithm—a proof of unsatisfiability expressed as a resolution graph. To make the problem more concrete, our challenge is to specify what it means to refute a set of propositional clauses via resolution. A more generic definition that also applies to first order clauses is given in §3.2.

Figure 1 shows an Alloy [5] solution to this problem.[1] The keyword "sig" introduces a set of atoms, called a signature. A field within a signature defines a relation of some arity whose leftmost column is the signature itself. For example, neg is a function from literals to their negations, and assign is a ternary relation that maps each Instance to a partial function from Literals to Booleans. The keyword "extends" specifies a containment relationship between sets. So, True and False are subsets of Boolean. The constraints that immediately follow a signature declaration hold for all atoms of that signature. For example, the constraint on line 18 means that the edges of every Refutation are free of cycles.

A Refutation has three components: sources, resolvents, and edges. The sources relation maps a Refutation to the nonempty set of clauses that it refutes. These clauses cannot include the conflict clause. The resolvents relation defines the set of clauses that are derivable from the sources via resolution, defined by the resolve predicate. The resolvents of a valid refutation must include the conflict clause. The edges relation describes the resolution relationships among the sources and resolvents of a refutation. Every resolvent is a target of some edge, and the source of that edge is a clause used in resolution derivation of the target. The remaining definitions are straightforward.

2.1 Sample Analyses

We validate an Alloy model against an assertion that we believe to be true by instructing the Alloy Analyzer [6] to check that the conjunction of the model and the negation of the assertion is unsatisfiable. The check is performed with respect to a finite *scope*, which bounds the number of atoms that the Analyzer may assign to each signature in the model. If the assertion is invalid in the given

[1] A simpler example motivating the use of unsatisfiable cores, with a slower-paced introduction to Alloy, can be found in the paper by Shlyakhter et al. [3].

scope, the Analyzer produces a *counterexample*—an assignment of values to sets and relations that satisfies the model but violates the assertion. The absence of a counterexample, however, does not necessarily constitute a proof of validity. Rather, it indicates one of the following:

1. the assertion is valid but the model is too strong,
2. the assertion and the model are both valid,
3. the assertion is too weak, or
4. the scope is too small.

Each of these cases leads to an identifiable pattern of minimal cores, discussed below.

Case 1: The Model is Too Strong. The first case is probably the most common. It happens when a part of the model itself is overconstrained, admitting either no solutions or just the uninteresting ones. As a result, many assertions follow trivially from the model.

```
1  abstract sig Boolean {}                          // The set of booleans is partitioned into
2  one sig True, False extends Boolean {}           // singleton sets True and False.

3  sig Literal { neg: Literal }                     // Each literal has an associated negation.
4  fact { neg = ~neg ∧ (no iden ∩ neg) }            // Negation is symmetric and irreflexive.

5  sig Clause { lits: set Literal }                 // Each clause contains a set of literals.
6  one sig Conflict extends Clause {} { no lits }   // One empty clause is denoted Conflict.
7  fact { ∀ c: Clause \ Conflict | some c.lits }    // Every clause other than Conflict is nonempty.
8  fact { ∀ c: Clause | no c.lits ∩ c.lits.neg }    // No clause has both a literal and its negation.

9  pred resolve [c1, c2, r: Clause] {               // Resolving clauses c1 and c2 yields r if
10    ∃ x: c1.lits ∩ c2.lits.neg |                  // c1 contains some literal x, c2 contains !x,
11      r.lits = (c1.lits ∪ c2.lits) \ (x ∪ x.neg)  // and r is a union of c1 and c2 minus x and !x.
12 }

13 sig Refutation {                                 // Each refutation consists of
14    sources: some Clause \ Conflict,              // a set of nonempty clauses called 'sources,'
15    resolvents: set Clause,                       // a set of clauses called 'resolvents,' and
16    edges: (sources ∪ resolvents)→resolvents      // a set of edges from clauses to resolvents,
17 }{                                               // such that
18    no ^edges ∩ iden                              // 1) The edge relation is acyclic;
19    ∀ r: resolvents | some edges.r                // 2) Every resolvent has some incoming edges;
20    Conflict ⊆ resolvents                         // 3) The empty clause is a resolvent;
21    ∀ n1, n2: sources ∪ resolvents |              // 4) For every source or resolvent n1 and n2
22     ∀ r: resolvents |                            //      for every resolvent r
23      ((n1 ∪ n2)→r ⊆ edges                        //      there are two edges ⟨n1, r⟩ and ⟨n2, r⟩
24        ⇔ resolve[n1, n2, r])                     //      if and only if n1 and n2 resolve to r.
25 }

26 sig Instance {
27    clauses: some Clause,                         // Each instance has a nonempty set of clauses,
28    assign: Literal→lone Boolean                  // and each literal is assigned at most one value.
29 }{
30    ∀ lit: clauses.lits |                         // Each mentioned literal is assigned a value,
31      assign[lit] = Boolean \ assign[lit.neg]     // and its negation has the opposite value.
32    ∀ c: clauses | True ⊆ assign[c.lits]          // Each clause has at least one true literal.
33 }
```

Fig. 1. A buggy formalization of resolution refutation

The example in Fig. 1 contains a bona fide error that one of the authors made in the first version of the model. It was revealed by checking that a set of clauses cannot have both an instance and a refutation:

```
check { ∀ i: Instance | ∄ ref: Refutation | ref.sources = i.clauses } for 3
```

The Analyzer confirms that the assertion has no counterexamples in a scope of 3, and highlights these constraints as a minimal cause of unsatisfiability:

```
 5  sig Clause { lits: set Literal }
 8  fact { ∀ c: Clause | no c.lits ∩ c.lits.neg }
13  sig Refutation {
16     edges: (sources ∪ resolvents)→resolvents
17  }{
19     ∀ r: resolvents | some edges.r
20     Conflict ⊆ resolvents
21     ∀ n1, n2: sources ∪ resolvents |
22      ∀ r: resolvents |
23        ((n1 ∪ n2)→r ⊆ edges
24          ⇔ resolve[n1, n2, r])
25  }
    check { ∀ i: Instance | ∄ ref: Refutation | ref.sources = i.clauses } for 3
```

Increasing the analysis scope to 4, 5, and 6 yields the same result: the definition of Instance is not needed to prove the assertion. What's wrong?

Examining the highlighted lines more closely reveals that the definition of refutation edges is too strong. It forces each Refutation to have at least one resolvent (line 20) and to therefore include at least one edge (line 19). But, the constraints on lines 21-24 and line 8 prevent any edge from existing. To see why, let $\langle c_1, c_2 \rangle$ be an edge between some clauses c_1 and c_2. The formula on lines 21-24 simplifies to $(c_1 \cup c_1) \rightarrow c_2 \subseteq \mathsf{edges} \Leftrightarrow \mathrm{resolve}[c_1, c_1, c_2]$ when c_2 is substituted for r and c_1 for n1 and n2. By our hypothesis, $\langle c_1, c_2 \rangle \subseteq \mathsf{edges}$, so $\mathrm{resolve}[c_1, c_1, c_2]$ must be true. The definition of resolution (Fig. 1, lines 10-11), however, says that c_1 must contain both a literal and its negation, which contradicts the constraint on line 8. A revised definition of edges is given below:

```
21  edges = {                                  // For every source or resolvent n, for every
22     n: sources ∪ resolvents, r: resolvents |  // resolvent r, ⟨n, r⟩ is an edge if there is
23       one edges.r \ n ∧                       // a unique clause m!=n such that ⟨m, r⟩
24       resolve[n, edges.r \ n, r]  }           // is an edge, and n and m resolve to r.
```

Case 2: The Model and Assertion are Both Valid. A valid model and a valid assertion produce cores that highlight both the assertion and all the definitions to which it pertains. When we revise the definition of edges and check the previous assertion against the revised model, the Analyzer, once again, finds no counterexample within a scope of 3. But, the derived core now includes the entire definition of Clause, Refutation, and Instance. Moreover, it remains the same with increasing scope, suggesting that the model and the assertion are both valid.

Case 3: The Assertion is Too Weak. A valid assertion that exercises only a small portion of a model is called *weak*. By themselves, weak assertions are not

harmful, but they can be misleading. If the modeler believes a weak assertion covers all or most of the model, he can miss real errors in the parts of the model that are not exercised. For example, the following assertion is supposed to validate the Instance definition. It states that, if an instance satisfies a set of clauses, then it must also satisfy all subsets of those clauses:

```
check { ∀ i: Instance, cs: set i.clauses | cs ⊆ lits.(i.assign.True) } for 3
```

The Analyzer finds no counterexample, but produces the following minimal core that, once again, does not include more constraints as the scope is increased:

```
26  sig Instance {
29  }{
32    ∀ c: clauses | True ⊆ assign[c.lits]
33  }
    check { ∀ i: Instance, cs: set i.clauses | cs ⊆ lits.(i.assign.True) } for 3
```

The problem here is that the assertion covers only the highlighted part of the Instance definition, when the intention was to cover the definition in its entirety. That is, the assertion was intended to fail if any part of the Instance definition was wrong. But, if we had, for example, accidentally omitted the "lone" keyword from the declaration of assign (Fig. 1, line 28), which ensures that each literal gets at most one value, checking this assertion would not produce a counterexample.

Case 4: The Scope is Too Small. The last case is the easiest to diagnose: if the scope is too small, the minimal core usually increases when the analysis is repeated in a larger scope. In the case of a valid assertion, the core will stop increasing after a while. For an invalid one, the core will often continue to grow with scope until the scope becomes large enough to reveal a counterexample. The following assertion, which states that the edges of a resolution graph never point to source clauses, illustrates this scenario:

```
check { ∀ ref: Refutation | no (ref.edges).(ref.sources) } for 2
```

In the search scope of 2, no counterexample exists and the unsatisfiable core includes only the assertion and the definition of resolution edges:

```
13  sig Refutation {
17  }{
21    edges = {
22      n: sources ∪ resolvents, r: resolvents |
23        one edges.r \ n ∧
24        resolve[n, edges.r \ n, r]  }
25  }
    check { ∀ ref: Refutation | no (ref.edges).(ref.sources) } for 2
```

As we increase the scope, however, the core expands to include more and more of the model—Refutation, Clause, and Literal definitions—until a counterexample is found in a scope of 5. The assertion is invalid because the sources of a refutation graph can be redundant; i.e. they can include a clause that is derivable from other source clauses via resolution.

3 Finding Minimal Cores

The *Simple* and *Recycling Core Extractor* (SCE and RCE) are new algorithms for finding minimal unsatisfiable cores of declarative specifications. They were developed in the context of the Alloy language and SAT-based analysis, but are independent of either. Both SCE and RCE are applicable to any specification language that can be translated to the input language of some resolution engine as described in §3.1-3.3. Unlike the alternatives (§3.4), they guarantee minimality (§3.6) at a reasonable cost (§4).

3.1 Specifications and Cores

A *declarative specification* is a conjunction of constraints on variables $v_i \in V$ that range over a universe U of values. A *model* or an *instance* of a *satisfiable* specification is a binding of $v_i \in V$ to elements of U that makes the specification true. An *unsatisfiable* specification has no models, but it has one or more *unsatisfiable cores*—subsets of the specification's constraints which are themselves unsatisfiable. Such a core is *minimal* if removing any one of its constraints causes the remainder of the core to become satisfiable.

We assume that a declarative specification $S = s_1 \wedge \ldots \wedge s_k$ is encoded in a language $\mathcal{L}$ as a directed, acyclic Abstract Syntax Graph (ASG) with k roots. The remaining constraints on the structure of ASGs capture the usual syntactic rules for declarative languages. In particular, the leaves of the ASG are variables $v_i \in V$ and constants in U, and each internal node n computes a predetermined function $f : U^{|n|} \rightarrow U$ of its children, $c_1, \ldots, c_{|n|}$.

The meaning of an ASG node n with respect to a binding $b : V \mapsto U$ is computed by applying the function f to the values of n's children: $[\![n]\!]b = f([\![c_1]\!]b, \ldots, [\![c_{|n|}]\!]b)$. The root nodes compute Boolean functions whose conjunction is the value of S as a whole. Hence, S is satisfiable if there is a binding for the variables $v_i \in V$ that induces the value *true* in the roots of its ASG. In the remainder of the paper, we will take S to mean "the ASG of S."

3.2 Resolution Engine

Invalidity of a specification can be proved by converting it to a clausal logic and then applying a suitable *resolution engine* to the generated clauses. At its simplest, a resolution engine is a procedure that applies resolution to a set of clauses in conjunctive normal form until it detects a conflict or determines satisfiability. Because resolution is refutation complete [7], a resolution engine is guaranteed to terminate on an unsatisfiable clause set with a proof of its unsatisfiability. This proof takes the form of a *resolution refutation* (Fig. 2), defined as follows:

Definition 1 (Resolution refutation). *Let C and R be sets of clauses such that C is unsatisfiable and $R \setminus C$ contains the empty (conflict) clause, denoted by $c_\emptyset$. Let E be a set of edges from $C \cup R$ to R. A directed acyclic graph $G = (C, R, E)$ is a resolution refutation of C iff*

1. *the sources of G are in C;*
2. *each $r \in R$ is the result of resolving some clauses $s_0, s_1, \ldots, s_k \in C \cup R$, represented by $\langle s_0, r\rangle, \ldots, \langle s_k, r\rangle \in E$ (which are the only edges in E); and,*
3. *$c_\emptyset$ is a sink of G.*

The sources of a resolution refutation $G = (C, R, E)$ that are connected to $c_\emptyset$ form an unsatisfiable core of C. The core of C with respect to (C, R, E) is denoted by $\{c \in C \mid c_\emptyset \in E^*(\!|c|\!)\}$, where E^* is the reflexive transitive closure of E and $E^*(\!|c|\!)$ is the relational image of c under E^*.

The behavior of a resolution engine on an arbitrary clause set depends on the decidability of its input language. For example, a SAT solver [8,9,10] will eventually produce a model or a refutation for every set of propositional clauses, while a theorem prover [11, 12, 13] will run forever on some sets of first order clauses. We abstract away from the particulars of the concrete engines' behavior with a partial function $\mathcal{E} : \mathbb{P}(C) \rightharpoonup G$ that maps each unsatisfiable clause set to a resolution refutation. The remaining sets in the domain of $\mathcal{E}$ are taken to resolution graphs that do not include $c_\emptyset$ (indicating satisfiability).

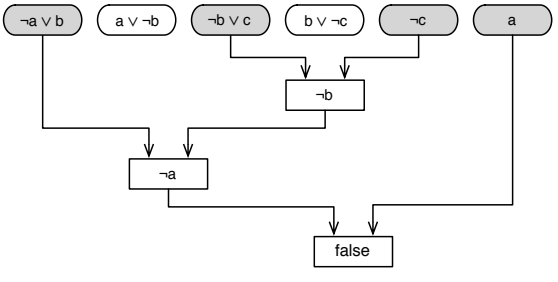

Fig. 2. Resolution refutation of $(a = b) \wedge (b = c) \wedge \neg(a \Rightarrow c)$. Core clauses are shaded in gray. The false square designates the conflict clause.

3.3 Translation

There are many ways to translate an ASG to a set of clauses in conjunctive normal form (e.g. [14,15,16]). The details of such a translation are unimportant for its use with our core extraction algorithms, as long as it is *regular* in the following sense:

Definition 2 (Regular Translation). *A procedure $\mathcal{T} : \mathcal{L} \to \mathbb{P}(C)$ is a regular translation from the specification language $\mathcal{L}$ to the clausal logic $\mathbb{P}(C)$ iff*

1. *a specification $S \in \mathcal{L}$ is unsatisfiable iff $\mathcal{T}(S)$ is unsatisfiable;*
2. *the translation of a specification $S \in \mathcal{L}$ is the union of the translations of its constraints: $\mathcal{T}(S) = \mathcal{T}_S(roots(S)) = \cup_{s \in roots(S)} \mathcal{T}_S(s)$, where $\mathcal{T}_S(s)$ is the translation of the constraint s in the context of the specification S; and,*
3. *the translation of the constraints $\sigma = roots(S) \cap roots(S')$ is context independent up to a renaming: $\mathcal{T}_S(\sigma) = r(\mathcal{T}_{S'}(\sigma))$ for some bijection r over the symbols (i.e. variable, constant, function, and predicate names) used in $\mathcal{T}(S) \cup \mathcal{T}(S')$, lifted to clauses and sets of clauses in the obvious way.*

Informally, a regular translation takes a specification to an equisatisfiable set of clauses, in a context independent way. For example, suppose that we have two specifications $S = \exists x.p(x)$ and $S' = (\forall x.q(x)) \wedge (\exists x.p(x))$ whose free variables range over a universe of two atoms, $\{a_0, a_1\}$. A regular translation $\mathcal{T}$ of these specifications to propositional logic might generate the clauses $\mathcal{T}(S) = (v_0 \vee v_1)$ and $\mathcal{T}(S') = v_0 \wedge v_1 \wedge (v_2 \vee v_3)$. In the context of S, the value of the predicate p on atoms a_0 and a_1 is represented by boolean variables v_0 and v_1, respectively. In the context of S', p is represented by v_2 and v_3. As a result, the translation of the constraint $\exists x.p(x)$ is not context-free. But, it is context independent, because $\mathcal{T}_S(\exists x.p(x))$ and $\mathcal{T}_{S'}(\exists x.p(x))$ are equivalent up to the renaming of v_0 to v_2 and v_1 to v_3.

3.4 Basic Core Extraction Algorithms

The *Naive Core Extractor* (NCE) is the most basic algorithm for extracting minimal cores of declarative specifications (Fig. 3a). It starts with an initial core K that contains all roots of the unsatisfiable specification S (line 1). The initial core is then pruned, one constraint at a time, by discarding all constraints u for which a regular translation of $K \setminus \{u\}$ is unsatisfiable (lines 3-8). This pruning step is sound since the regularity of the translation guarantees that $\mathcal{T}(K \setminus \{u\})$ and $K \setminus \{u\}$ are equisatisfiable. In the end, K contains a minimal core of S.

Because it calls the computationally expensive resolution procedure once for each constraint, NCE tends to be unacceptably slow for large specifications with small, hard cores. Shlyakhter et al. [3] addressed this problem with the *One-Step Core Extractor* (OCE) algorithm which sacrifices minimality for scalability. OCE (Fig. 3b) simply returns all roots of S whose translations include clauses connected to the conflict clause $c_\emptyset$ in a refutation of $\mathcal{T}(S)$. The set of constraints computed in this way is an unsatisfiable core of S (§3.6, Thm. 1), but it is usually not minimal.

3.5 Simple and Recycling Core Extraction

The *Simple Core Extractor* (SCE) combines the core-pruning loop of NCE with the core-extraction technique of OCE (Fig. 3c). In particular, SCE is NCE with the following modifications: initialize K with a core of S instead of S (line 14), and reduce K to a core of $K \setminus \{u\}$ instead of $K \setminus \{u\}$ itself in the iterative step (line 21). Correctness and minimality of SCE's output are discussed in §3.6.

Although it avoids unnecessary calls to the resolution engine, SCE is still wasteful. By applying $\mathcal{E}$ solely to $\mathcal{T}_S(K \setminus \{u\})$ on line 19, it discards all the clauses that $\mathcal{E}$ has learned about $\mathcal{T}_S(K \setminus \{u\})$ in previous iterations (while refuting $\mathcal{T}_S(K)$). When these clauses are recycled, SCE turns into the *Recycling Core Extractor* algorithm (RCE).

The pseudocode for RCE is shown in Fig. 3d. As before, line 24 initializes K to the unsatisfiable core of S extracted from $\mathcal{E}(\mathcal{T}(S))$. Lines 29-30 construct $\mathcal{T}(K \setminus \{u\})$ (from the already computed translations of the roots of S) and collect the clauses, called *resolvents*, that $\mathcal{E}$ had already learned about $\mathcal{T}(K \setminus \{u\})$. These are simply all resolvents reachable from $\mathcal{T}(K \setminus \{u\})$ but not from the other clauses previously fed to $\mathcal{E}$. If they include the conflict clause $c_\emptyset$, u is discarded (line 32)

```
NCE(S: L, T: L → P(C), E: P(C) → G)
1  K ← roots(S)
2  M ← {}
3  while K ⊈ M do
4    u ← pick(K \ M)
5    M ← M ∪ {u}
6    (C, R, E) ← E(T(K \ {u}))
7    if c∅ ∈ R then
8      K ← K \ {u}
9  return K
```

(a) Naive Core Extractor

```
OCE(S: L, T: L → P(C), E: P(C) → G)
10  (C, R, E) ← E(T(S))
11  K ← {s ∈ roots(S) | c∅ ∈ E*(⦇T_S(s)⦈)}
12  return K
```

(b) One-Step Core Extractor

```
SCE(S: L, T: L → P(C), E: P(C) → G)
13  (C, R, E) ← E(T(S))
14  K ← {s ∈ roots(S) | c∅ ∈ E*(⦇T_S(s)⦈)}
15  M ← {}
16  while K ⊈ M do
17    u ← pick(K \ M)
18    M ← M ∪ {u}
19    (C, R, E) ← E(T_S(K \ {u}))
20    if c∅ ∈ R then
21      K ← {s ∈ K \ {u} | c∅ ∈ E*(⦇T_S(s)⦈)}
22  return K
```

(c) Simple Core Extractor

```
RCE(S: L, T: L → P(C), E: P(C) → G)
23  (C, R, E) ← E(T(S))
24  K ← {s ∈ roots(S) | c∅ ∈ E*(⦇T_S(s)⦈)}
25  M ← {}
26  while K ⊈ M do
27    u ← pick(K \ M)
28    M ← M ∪ {u}
29    C′ ← T_S(K \ {u})
30    R′ ← R \ E*(⦇C \ C′⦈)
31    if c∅ ∈ R′ then
32      K ← K \ {u}
33    else
34      (C″, R″, E″) ← E(C′ ∪ R′)
35      if c∅ ∈ R″ then
36        (C, R, E) ← (C′, R′ ∪ R″, E″ ∪ (E ▷ R′))
37        K ← {s : K \ {u} | c∅ ∈ E*(⦇T_S(s)⦈)}
38  return K
```

(d) Recycling Core Extractor

Fig. 3. Core extraction algorithms. S is an unsatisfiable specification, $\mathcal{T}$ is a regular translation, and $\mathcal{E}$ is a resolution engine. Star (*) means reflexive transitive closure, $r(\!|X|\!)$ is the relational image of X under r, and $\triangleright$ is range restriction.

because there must be some other constraint in $K \setminus \{u\}$ whose translation contributes the same or a larger set of clauses to the core of C as u. Otherwise, line 34 applies $\mathcal{E}$ to $\mathcal{T}(K \setminus \{u\})$ and its resolvents. If the result is a refutation, the invalidity of K can be proved without u. Before we can extract the u-free core from (C'', R'', E''), however, we have to fix it: (C'', R'', E'') is not a valid refutation of S because its sources include the resolvents for $\mathcal{T}(K \setminus \{u\})$. So, lines 36-37 fix the proof and set K to the corresponding core, which excludes at least u.

3.6 Correctness and Minimality of SCE and RCE

Both SCE and RCE rely on OCE's core extraction technique to reduce the number of calls to the resolution engine. Establishing the correctness of OCE's output is therefore the first step to proving the correctness of SCE and RCE:

Theorem 1. *Let $G = (C, R, E)$ be a resolution refutation for $C = \mathcal{T}(S)$, a regular translation of the unsatisfiable specification S. Then, $K = \{s \in roots(S) \mid c_\emptyset \in E^*(\!|\mathcal{T}_S(s)|\!)\}$ is an unsatisfiable core of S.*

Proof. Let S' be a specification whose roots are K, i.e. $roots(S') = K$. Because $\mathcal{T}$ is regular, $\mathcal{T}(S') = \mathcal{T}_{S'}(roots(S')) = \mathcal{T}_{S'}(K) = r(\mathcal{T}_S(K))$ for some renaming r. Let $C_K = \mathcal{T}_S(K)$, $R_K = E^*(\!|C_K|\!)$ and $E_K = E \triangleright R_K$, where $\triangleright$ is range restriction. By Def. 1 and the construction of K, the graph $G_K = (C_K, R_K, E_K)$ is a resolution refutation of $\mathcal{T}_S(K)$, and, letting $r(G_K)$ denote G_K with r applied to all of its vertices, $r(G_K)$ is a resolution refutation of $r(\mathcal{T}_S(K)) = \mathcal{T}(S')$. Hence, by regularity of $\mathcal{T}$, S' is unsatisfiable and, by the semantics of ASGs, so is K. □

We can now show that RCE produces a minimal unsatisfiable core of the input specification S, if the input engine terminates on each invocation. Since RCE reduces to SCE when R' is set to the empty set on line 30, the following is also a proof of SCE's correctness and minimality:

Theorem 2. *If it terminates, RCE(S, $\mathcal{T}$, $\mathcal{E}$) returns a minimal unsatisfiable core of S, where S is an unsatisfiable specification, $\mathcal{T}$ is a regular translation, and $\mathcal{E}$ a resolution engine.*

Proof. Let K be the output of RCE(S, $\mathcal{T}$, $\mathcal{E}$). We first show that K is unsatisfiable and then that it is minimal. By Thm. 1, the constraints assigned to K by line 24 form an unsatisfiable core of S. The only other lines that assign K are lines 32 and 37. Suppose that the condition on line 31 is true. Then, by Def. 1 and construction of C' and R', $(C', R', E \triangleright R')$ is a resolution refutation for $C' = \mathcal{T}_S(K \setminus \{u\})$ which, by regularity of $\mathcal{T}$, is equivalent (up to a renaming) to $\mathcal{T}(K \setminus \{u\})$. Hence, $K \setminus \{u\}$ is unsatisfiable, so line 32 will never remove a relevant constraint from K. For line 37 to execute, the condition on line 35 must hold. If it does, line 36 executes first, establishing (C, R, E) as a resolution refutation for $C = C' \equiv_r \mathcal{T}(K \setminus \{u\})$ (Defs. 1, 2). This and Thm. 1 ensure that the constraints assigned to K in line 37 form an unsatisfiable core of $K \setminus \{u\}$.

Now, suppose that K is not minimal. Then, there is a constraint $s \in K$ such that $K \setminus \{s\}$ is unsatisfiable. Lines 26 and 28 ensure that s is picked at least once on line 27. Because $K \setminus \{s\}$ is invalid, either the condition on line 31 or that on line 35 holds, causing s to be removed from K—a contradiction. □

4 Experimental Results

We have implemented both SCE and RCE for the Alloy language, with MiniSat [8] as the resolution engine and Kodkod [14] as the (regular) translation procedure from Alloy to propositional clauses. These implementations were evaluated against the basic algorithms (NCE and OCE) on two sets of problems: fifteen TPTP [17] benchmarks and six problems from the Alloy4 distribution [6]. The chosen problems come from a variety of fields (software engineering, medicine, geometry, etc.), include 4 to 59 constraints, and exhibit a wide range

	problem	size	scope	variables	clauses	transl (sec)	solve (sec)	initial core	min core	OCE (sec)	NCE (sec)	SCE (sec)	RCE (sec)	tRCE (sec)
Alloy	Trees	4	7	407396	349384	10	98	4	4	1	7	7	7	7
	RingElection	10	8	59447	187381	1	49	9	9	2	59	7	8	9
	Lists.empties	12	60	2547216	7150594	74	12	7	6	9	196	89	86	86
	Lists.reflexive	12	14	34914	91393	1	23	10	5	3	134	120	158	96
	Lists.symmetric	12	8	7274	17836	0	27	12	7	3	150	115	93	85
	Hotel	59	5	22407	55793	0	0	53	29	0	27	14	11	11
TPTP	ALG212	6	7	1072203	1027000	7	63	6	5	1	103	104	107	98
	COM008	14	9	6154	9845	0	1	14	10	0	190	193	235	166
	NUM374	14	3	6874	18938	0	0	14	6	0	3	3	3	3
	TOP020	14	10	2554114	4262733	21	113	2	2	6	826	10	10	10
	SET943	18	5	5333	12541	0	0	14	4	0	19	18	15	13
	SET948	20	14	339132	863889	5	36	10	6	1	754	247	359	254
	SET967	20	4	14641	45112	0	0	10	2	0	454	181	142	142
	GEO091	26	10	106329	203303	9	108	24	7	3	1129	652	105	105
	GEO092	26	8	48500	91285	3	7	24	7	0	120	99	70	51
	GEO158	26	8	46648	88234	3	38	25	7	2	175	107	45	45
	GEO115	27	9	109002	188782	6	85	25	7	2	675	278	63	86
	LAT258	27	7	205621	336912	2	11	26	20	0	95	87	70	70
	GEO159	28	8	87214	195200	10	57	24	7	1	223	83	50	50
	MED007	41	35	130777	265702	2	67	24	7	1	>3600	>3600	176	91
	MED009	41	35	130777	265703	2	71	26	7	1	>3600	>3600	85	76

Fig. 4. Experimental results. The notation ">3600" means that an algorithm was unable to produce a core for the specified problem in the given scope within one hour. Gray shading highlights the best running time among NCE, SCE, RCE, and trained RCE (tRCE).

of behaviors. In particular, eleven are theorems (i.e. unsatisfiable conjunctions of axioms and negated conjectures); four are assumed to be (counter)satisfiable but have no known finite models; two are unsatisfiable in some universes and satisfiable in others; and four have neither an assumed status nor any known finite models.

Each problem p was tested for satisfiability in scopes of increasing sizes until a failing scope $s_{\text{fail}}(p)$ was reached in which either a model was found or all three minimality-guaranteeing algorithms failed to produce a result for that scope within 5 minutes (300 seconds). Then, because our implementation of RCE is parameterized by a "resolution distance" d that controls which resolvents are reused in each iteration[2], RCE was automatically trained using a scope of $0.75 * (s_{\text{fail}}(p) - 1)$ to estimate the best d for the problem p. Once the experimental parameters were determined, the algorithms were tested on each problem using a scope of $s_{\text{fail}}(p) - 1$. All experiments were performed on a 2×3 GHz Dual-Core Intel Xeon with 2 GB of RAM, with a cut-off time of one hour (3600 seconds).

The results are shown in Fig. 4. The first three columns show the name of the problem, the number of constraints it contains, and the scope in which it was tested. The next two columns contain the number of propositional variables and clauses produced by the translator. The "transl (sec)" and "solve (sec)" columns show the time, in seconds, taken by the translator to generate the problem and the SAT solver to produce the initial refutation. The "initial core" and "min

[2] A relevant resolvent $r \in R'$ (Fig. 3d, line 30) is recycled if all paths from r to a source in C' (Fig. 3d, line 29) contain at most d edges.

	problem	N-score	NCE / RCE	NCE / tRCE	average speed-up
easy	NUM374	-0.34	1.04	0.95	RCE: 1.48x tRCE: 1.56x
	SET943	0.12	1.29	1.44	
	SET967	0.53	3.20	3.20	
	Trees	0.59	1.08	1.08	
	COM008	0.60	0.81	1.14	
medium	Hotel	1.08	2.54	2.54	RCE: 2.45x tRCE: 2.59x
	RingElection	1.73	7.13	6.31	
	ALG212	1.82	0.96	1.05	
	Lists.empties	1.87	2.28	2.29	
	LAT258	1.89	1.36	1.36	
	Lists.symmetric	2.13	1.61	1.77	
	GEO092	2.14	1.73	2.35	
	Lists.reflexive	2.21	0.85	1.40	
	SET948	2.70	2.10	2.97	
	GEO158	2.86	3.90	3.89	
hard	GEO159	3.08	4.49	4.48	RCE: 29.04x tRCE: 32.65x
	TOP020	3.13	85.26	85.76	
	GEO115	3.23	10.74	7.82	
	GEO091	3.32	10.78	10.77	
	MED007	3.36	20.51	39.54	
	MED009	3.38	42.47	47.53	

(a) RCE and tRCE versus NCE

	problem	S-score	SCE / RCE	SCE / tRCE	average speed-up
easy	NUM374	-0.34	1.05	0.96	RCE: 1.08x tRCE: 1.11x
	SET943	-0.03	1.21	1.36	
	SET967	0.18	1.28	1.28	
	TOP020	0.35	0.99	1.00	
	Trees	0.59	1.08	1.08	
	COM008	0.60	0.82	1.16	
	RingElection	0.65	0.87	0.77	
	Hotel	0.99	1.31	1.31	
medium	Lists.empties	1.12	1.04	1.04	RCE: 1.27x tRCE: 1.43x
	ALG212	1.82	0.97	1.06	
	LAT258	1.82	1.24	1.24	
	Lists.reflexive	2.06	0.76	1.25	
	GEO092	2.09	1.43	1.94	
	Lists.symmetric	2.13	1.24	1.36	
	SET948	2.16	0.69	0.97	
	GEO158	2.83	2.39	2.39	
	GEO159	2.99	1.67	1.66	
hard	MED007	3.06	20.51	39.54	RCE: 18.41x tRCE: 24.13x
	MED009	3.13	42.47	47.53	
	GEO115	3.18	4.42	3.22	
	GEO091	3.27	6.23	6.22	

(b) RCE and tRCE versus SCE

Fig. 5. Comparison of minimal core extractors based on problem difficulty

core" columns present the number of constraints in the initial core found by OCE and the minimal core found by the minimality-guaranteeing algorithms. The remaining columns show core extraction time, in seconds, for each algorithm.

On average, RCE outperforms NCE and SCE by a factor of 10 and 4, respectively; its trained variant (tRCE) is roughly 11 times faster than NCE and 6 times faster than NCE. These overall averages, however, do not take into account the wide variance in difficulty among the tested problems. A more useful comparison of the minimality-guaranteeing algorithms is given in Fig. 5, where we classify the problems according to their difficulty for NCE (Fig. 5a) and SCE (Fig. 5b), and then report how well the RCE variants perform on the problems deemed as "easy", "moderately hard" or "hard" for the competing algorithms.

To assess the difficulty of a given problem for NCE, we compute its *N-score*, and rate it as easy if the score is less than 1, hard if the score is 3 or more, and moderately hard otherwise. The N-score for a specification S is $\log_{10}((s-m)*t+m*t*.01)$, where s is the size of the specification, m is the size of its minimal core, and t is the time, in seconds, taken by the SAT solver to determine that S is unsatisfiable. Note that the N-score for a problem measures how much work NCE has to do to eliminate irrelevant constraints from the specification, which is approximated by predicting that NCE will take $(s-m)*t$ seconds to prune away the $(s-m)$ irrelevant constraints. (The formula assumes that it takes only 1 percent of the initial time to throw out a relevant constraint because of the ability of modern SAT solvers to find satisfying assignments very quickly.) The difficulty of a problem for SCE is computed in a similar way; the S-score of a given problem is $\log_{10}((s'-m)*t+m*t*.01)$, where s' is the size of the initial (one-step) core.

Unsurprisingly, OCE outperforms both SCE and RCE in terms of execution time. However, it generates cores that are on average 2.4 times larger than the corresponding minimal cores. For 20 out of 21 (95%) of the tested problems, the OCE core included at least 50% of the original constraints. In contrast, only 7 out of 21 (33%) minimal cores included at least half of the original constraints.

5 Related Work

The problem of finding unsatisfiable cores of sets of constraints has been studied in the context of linear programming [18], propositional satisfiability [19, 20, 21, 22, 23, 24, 25], and finite model finding [3]. Chinneck and Dravnieks' [18] deletion filtering algorithm for linear constraints is similar to NCE: given an infeasible linear program LP, the algorithm tests each functional constraint for membership in an Irreducible Infeasible Subset (i.e. minimal unsatisfiable core) by removing it from LP and applying a linear programming solver to the result. If the reduced LP is infeasible, the constraint is permanently removed, otherwise it is kept. The remaining algorithms in [18] are specific to linear programs, and there is no obvious way to adapt them to other domains.

Most of the work on extracting small unsatisfiable cores comes from the SAT community. Several practical algorithms [20, 24, 25] have been proposed for finding small, but not necessarily minimal, cores of propositional formulas. Zhang and Malik's algorithm [25], for example, works by extracting a core from a refutation, feeding it back to the solver, and repeating this process until the size of the extracted core no longer decreases. A few proposed algorithms provide strong optimality guarantees—such as returning the smallest minimal core [22, 23] or all minimal cores [21, 26, 27, 28] of a boolean formula—at the cost of scaling to problems that are orders of magnitude smaller than those handled by the approximation algorithms. The Complete Resolution Refutation (CRR) algorithm by Dershowitz et al. [19] strikes an attractive balance between scalability and optimality: it finds a single minimal core but scales to large real-world formulas. CRR was one of the inspirations for our work and is, in fact, an instantiation of RCE for propositional logic, with a SAT solver as a resolution engine and the identity function as the translation procedure.

The work by Shlyakhter et al. [3] is most closely related to ours. It proposes the One-Step Core Extractor (OCE) for declarative specifications in a language reducible to propositional logic. As discussed in previous sections, OCE is faster than RCE but produces cores that are two to three times larger than the corresponding minimal cores.

6 Conclusions

We have presented *recycling core extraction*, a new method for finding minimal unsatisfiable cores of declarative specification, and compared it to two simpler algorithms, NCE and SCE. On hard problems, the base recycling extraction algorithm (RCE), which reuses all available learned clauses, is about 29x faster

than NCE and 18x faster than SCE. But even greater speed-ups can be achieved with a simple variant of RCE that is trained to recycle a fixed subset of the available resolvents in each iteration.

RCE has so far been used as a coverage analysis for hand-crafted formal models within the interactive modeling environment of the Alloy Analyzer. It seems likely, however, that RCE will be applicable in other settings, particularly those involving large, automatically generated specifications, enabling its use for coverage analysis in code checking [29, 30, 31] and declarative configuration [32].

References

1. Chockler, H., Kupferman, O., Vardi, M.Y.: Coverage metrics for temporal logic model checking. LCNS 2031, 528 (2001)
2. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. In: Conference on Correct Hardware Design and Verification Methods, pp. 82–96 (1999)
3. Shlyakhter, I., Seater, R., Jackson, D., Sridharan, M., Taghdiri, M.: Debugging overconstrained declarative models using unsatisfiable cores. In: ASE 2003 (2003)
4. Shlyakhter, I.: Declarative Symbolic Pure Logic Model Checking. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (2005)
5. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT Press, Cambridge (2006)
6. Chang, F.: Alloy analyzer 4.0 (2007), `http://alloy.mit.edu/alloy4/`
7. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (1965)
8. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
9. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT solver. In: DATE 2002, pp. 142–149 (2002)
10. Mahajan, Y.S., Fu, Z., Malik, S.: zchaff2004: An efficient SAT solver. In: SAT (Selected Papers), pp. 360–375 (2004)
11. Kalman, J.A.: Automated Reasoning with Otter. Rinton Press (2001)
12. Riazanov, A.: Implementing an Efficient Theorem Prover. PhD Thesis, The University of Manchester, Manchester (2003)
13. Weidenbach, C.: Combining superposition, sorts and splitting. In: Handbook of automated reasoning, pp. 1965–2013 (2001)
14. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
15. Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model finding. In: CADE-19 Workshop on Model Computation, Miami, FL (2003)
16. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. In: Journal of Applied Logic (2007)
17. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)
18. Chinneck, J.W., Dravnieks, E.W.: Locating minimal infeasible constraint sets in linear programs. ORSA Journal of Computing 3(2), 157–158 (1991)

19. Dershowitz, N., Hanna, Z., Nadel, A.: A Scalable Algorithm for Minimal Unsatisfiable Core Extraction. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
20. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE 2003 (2003)
21. Liffiton, M.H., Sakallah, K.A.: On Finding All Minimally Unsatisfiable Subformulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 173–186. Springer, Heidelberg (2005)
22. Lynce, I.: On computing minimum unsatisfiable cores. In: SAT 2004 (2004)
23. Mneimneh, M., Lynce, I., Andraus, Z.: A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 467–474. Springer, Heidelberg (2005)
24. Oh, Y., Mneimneh, M., Andraus, Z., Sakallah, K., Markov, I.: Amuse: A minimally-unsatisfiable subformula extractor. In: DAC, ACM/IEEE, pp. 518–523 (2004)
25. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formula. In: SAT 2003 (2003)
26. Grégoire, E., Mazure, B., Piette, C.: Extracting MUSes. In: ECAI 2006, Trento, Italy, pp. 387–391 (2006)
27. Grégoire, E., Mazure, B., Piette, C.: Local-search extraction of MUSes. Constraints Journal 12(3), 324–344 (2007)
28. Grégoire, E., Mazure, B., Piette, C.: Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In: IJCAI 2007, Hyderabad, India, vol. 2, pp. 2300–2305 (2007)
29. Dennis, G., Chang, F., Jackson, D.: Modular verification of code. In: ISSTA 2006, Portland, Maine (2006)
30. Dolby, J., Vaziri, M., Tip, F.: Finding bugs efficiently with a sat solver. In: ESEC-FSE 2007, pp. 195–204. ACM, New York (2007)
31. Taghdiri, M.: Automating Modular Program Verification by Refining Specifications. PhD thesis, Massachusetts Institute of Technology (2007)
32. Yeung, V.: Declarative configuration applied to course scheduling. Master's thesis, Massachusetts Institute of Technology, Cambridge (2006)