Dynamite 2.0: New Features Based on UnSAT-Core Extraction to Improve Verification of Software Requirements

Mariano M. Moscato[1] and Carlos G. López Pombo[1] and Marcelo F. Frias[2]

[1] Department of Computer Science, FCEyN, Universidad de Buenos Aires and CONICET. e–mail:{`mmoscato,clpombo`}@dc.uba.ar.
[2] Department of Computer Engineering, Technological Institute of Buenos Aires (ITBA) and CONICET. e–mail:`mfrias`@itba.edu.ar.

Abstract. According to the Verified Software Initiative manifesto,

"*Lightweight techniques and tools have been remarkably successful in finding bugs and problems in software. However, their success must not stop the pursuit of this projects long-term scientific ideals*".

The Dynamite Proving System (DPS) blends the good qualities of the lightweight formal method Alloy with the certainty provided by the theorem prover PVS. Using the Alloy Analyzer during the proving process improves the PVS theorem proving experience by reducing the number of errors introduced along creative proof steps. Therefore, rather than becoming an obstacle to the goals of the Initiative, inside DPS Alloy becomes an aid. In this article we introduce new features of DPS based on the novel use of unsat cores to guide the proving process by pruning unnecessary information. We illustrate these new features using a non-trivial case-study coming from the networking domain.

1 Introduction

The Dynamite Proving System (DPS) was presented in [7]. The rationale behind DPS is that automated analysis, albeit incomplete, should support formal verification processes based on theorem proving. DPS has Alloy [9] as its specification language. Alloy's syntax includes constructs ubiquitous in modern object-oriented languages. The Alloy Analyzer [10], an analysis tool that provides automated (partial) analysis of Alloy specifications, makes Alloy a lightweight formal method with increasing adoption in academy and industry. The Alloy language (an extension of first-order logic with reflexive-transitive closure) is quite appropriate for modeling critical systems. According to the VSI manifesto [8],

"*Even though software requirements cannot be verified against a customers informal needs and desires, a great deal of clarity, insight, and precision can be gained by formalizing these requirements as a more precise specification. Once this is done, verification technology can be applied* *to the resulting formal specification, to investigate its consistency and to see if it captures important system properties such as safety or security.*"

The partial analysis offered by the Alloy Analyzer assumes data domains have a user-bounded size, called *scope*. Moreover, the analysis technique employed by the Alloy Analyzer (based on SAT-solving), does not scale well enough when domain sizes are increased past a (specification dependent) threshold. This should not be considered a shortcoming. Alloy's use is targeted at model debugging, and therefore small domain sizes are many times sufficient to uncover errors in specifications. Unfortunately, this kind of analysis only allows us to conclude that a system model is consistent, safe or secure up to a given size for data domains.

The Dynamite Proving System was developed with the intention of providing a tool for the verification (in the sense of the VSI Manifesto) of Alloy models. In order to accomplish this task, DPS extends the PVS [12] semi-automated theorem prover with a complete calculus for Alloy. We also integrated the Alloy Analyzer into DPS in order to automatically detect bugs introduced during creative proof steps (introduction of lemmas, new hypotheses, etc.) Including a pretty printer that exhibits sequents using Alloy notation, DPS provides the clerical Alloy user a more amenable and less error-prone theorem proving experience.

In order to better convey the contributions of this article, we will briefly discuss the proving process within PVS and DPS. DPS provides a complete calculus for Alloy, implemented on top of the higher-order calculus provided by PVS. In order to prove that a set of formulas $\Delta = \{\delta_1, \ldots, \delta_m\}$ follows from a set of hypotheses $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$, one begins with the *sequent* $\Gamma \vdash \Delta$. Applying *inference rules*, from $\Gamma \vdash \Delta$ one must reach other sequents that can be recognized as *valid* (for example, sequents of the form $\alpha \vdash \alpha$). The informal understanding of the sequent $\Gamma \vdash \Delta$ is that from the conjunction of the formulas in Γ, the disjunction of the formulas in Δ must follow. The formulas in Γ (Δ) are called the *antecedent* (*consequent*) of the sequent.

The application of an inference rule results in one or more new sequents whose proofs provide a proof of the original sequent. Therefore, proofs in this kind of calculi are usually seen as *trees* in which the root is the sequent $\Gamma \vdash \Delta$. When all the leaves of the proof tree are *valid* sequents (in the sense mentioned before) the tree is considered *closed* and the proof is finished. In Fig. 1 we present, as examples, proof rules in order to deal with conjunctions in the antecedent and the consequent, respectively.

$$\frac{\alpha, \beta, \Gamma \vdash \Delta}{\alpha \wedge \beta, \Gamma \vdash \Delta} \wedge\vdash \qquad\qquad \frac{\Gamma \vdash \Delta, \alpha \qquad \Gamma \vdash \Delta, \beta}{\Gamma \vdash \alpha \wedge \beta, \Delta} \vdash\wedge$$

Fig. 1. Proof rules for conjunction.

On start of a proof of an Alloy assertion α, DPS presents sequent $\vdash \alpha$. A proof must then be derived using the inference rules. Whenever the application of an inference rule introduces new goals (sequents) to be proved, some of the antecedents and consequents inherited by the new sequents may be unnecessary to close the branch that initiates in that sequent. Our experience using DPS in our case-study is that along a proof of a given assertion the number of antecedents and consequents in intermediate sequents tends to grow. This leads many times to formulas that are not necessary in order to prove the sequents. These formulas make the identification of new proof steps more complex. PVS provides a command (`hide`) for hiding hypotheses and conclusions in sequents, yet its use is error-prone: removing necessary antecedents or consequents makes the proof infeasible.

In this article we will use an Alloy UnSAT-core [16] in order to remove formulas from sequents and from the underlying theories. An Alloy UnSAT-core is a subset of formulas (and even parts of formulas) from an inconsistent (up-to the selected scopes) Alloy theory that is itself inconsistent. How is an inconsistent theory obtained at a given point in the proving process? Notice that proving a sequent $\Gamma \vdash \Delta$ in a theory Ω (where $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$ and $\Delta = \{ \delta_1, \ldots, \delta_m \}$), is equivalent to proving in theory Ω the sequent

$$\vdash \left(\bigwedge_{1 \leq i \leq k} \gamma_i \right) \Rightarrow \left(\bigvee_{1 \leq j \leq n} \delta_j \right) .$$

Elemental logic reasoning allows us to conclude that the former sequent is derivable if and only if the theory

$$\Omega \cup \left\{ \neg \left(\left(\bigwedge_{1 \leq i \leq k} \gamma_i \right) \Rightarrow \left(\bigvee_{1 \leq j \leq n} \delta_j \right) \right) \right\}$$

is inconsistent.

Notice that since we are not requesting a *minimal* UnSAT-core, the UnSAT-core might be the whole theory. Fortunately, this is most times not the case.

The contributions of this article are summarized as follows:

1. We release DPS 2.0, downloadable from `http://www.dc.uba.ar/dynamite`.
2. We present a novel heuristic to reduce the proof search space based on the use of UnSAT cores to remove possibly unnecessary antecedents and consequents in a sequent. The technique also allows us to remove formulas from the underlying theories.
3. We discuss the presented heuristic focusing on its (un)soundness and/or (in)completeness.
4. We discuss the applicability of the heuristic using as a reference a nontrivial case-study.
5. We present some experimental results obtained from extensively using this heuristic along the proving process in our case-study.

The article is organized as follows. In Section 2 we present our running example. In Section 3 we describe the proving process within Dynamite, including a discussion on how lightweight and heavyweight formal methods can be combined in a synergic way. In Section 4, after a short introduction to UnSat-cores, we present our heuristic for proof space reduction and discuss some experiences learnt using the technique. In Section 5 we discuss related work. Finally, in Section 6 we present our conclusions and proposals for further work.

2 Compositional Binding in Network Domains

In order to test the usefulness of the techniques we will present in this article, we worked on an Alloy model presented by Zave in [21]. There are good reasons for choosing this model. The first one is that although the model is not extremely complex, its analysis using the Alloy Analyzer does not scale for some properties even for small scopes. Notice that "small scope" is a subjective notion that strongly depends on the user knowledge about the model. It is seldom the case that Alloy models include information on the willingness of the user to analyze the model for scopes that surpass the possibilities of the Alloy Analyzer. This model is particular in that it thoroughly documents the intentions of the user:

```
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 9 Identifier -- this one is too big also
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 11 Identifier
-- attempted but not completed at MIT; formula is not that large; results
-- suggest that the problem is very hard, and that the formula is almost
-- certain unsatisfiable [which means that the assertion holds]
```

Notice that the modeler was concerned enough about the validity of the model assertions that she requested assistance from the developers of the Alloy Analyzer. The limitations of the automated techniques open the possibility to use verification in order to determine the validity (or not) of the model assertions. Of course there are other Alloy models that are also good candidates to be verified using DPS. Among these, we want to mention the Mondex electronic purse presented in [14], or the Flash filesystem presented in [11]. Since these problems have become sort of benchmarks for different analysis and verification techniques, it was our intention to leave them as interesting case-studies that might attract new users of DPS.

Zave formalizes a mechanism for binding of identifiers in the delivery of messages in computer networks. Thus, the model is not a specification of an isolated software or hardware artifact but rather the specification of network services whose implementation may involve several software and hardware artifacts. This model is mainly about communication in computer networks, and, more specifically, about how communicating agent identifiers are bound so that the messages reach their correct destination. Properties about the possibility of reaching an agent, determinism in the delivery of messages, existence of cycles in the routing of messages and the possibility of constructing a return path for a message are formally specified in the model. In particular, the model studies how these properties are affected by the addition of new bindings of identifiers.

Communicating artifacts in these kinds of networks may be software systems or hardware devices. As this distinction is not important for the specification, all the communicating artifacts are called *agents*. Thus, the communications are established between agents and take place over network domains.

An agent g is considered *reachable* in a domain d from an identifier i if:

- i is connected to an address a in the reflexive and transitive closure of the binary relation formed by all the bindings corresponding to d,
- a cannot be bound to another identifier in d, and
- a can route messages to g in d.

Figure 2 shows an Alloy assertion `BindingPreservesReachability`. This assertion states that if an agent is reachable in a domain d, it is also reachable in the domain resulting from adding a new binding to d, provided that the newly bound identifiers are not used in d. This latter condition is formalized by a predicate `IdentifiersUnused`.

```
assert BindingPreservesReachability {
   all d, d': Domain, newBinding: Identifier -> Identifier |
   IdentifiersUnused(d,newBinding.Identifier) &&
   AddBinding(d,d',newBinding)
   => (all i: Identifier, g: Agent |
         ReachableInDomain(d,i,g) => ReachableInDomain(d',i,g) ) }
```

Fig. 2. One proved property: `BindingPreservesReachability`.

A domain is called *deterministic* if each identifier is associated to at most one agent. One of the properties to be analyzed for this model states that

> *whenever a new binding for an unused identifier is added to a deterministic domain, it remains deterministic.*

A domain is considered *non-looping* if the transitive closure of the bindings for that domain has no cycles. A second assertion then states that

> *the addition of a new binding to a non-looping domain does not change this condition whenever the transitive closure of the new binding does not have cycles.*

Also a notion of *structured* domain is introduced.

In [21], Zave used the Alloy Analyzer to analyze this model and concluded that these five properties hold for Alloy domains containing at most two network domains and four elements in each set (such as identifiers, agents, etc).

Using DPS we have proved that the following assertions hold despite their domain bounds:

- `BindingPreservesReachability`,
- `BindingPreservesDeterminism`,
- `BindingPreservesNonLooping`,
- `ABindingPreservesStructure`, and
- `BBindingPreserverStructure`.

Notice that these assertions suffer the similar limitations, regarding their analyzability, with assertion `StructureSufficientForPairReturnability`.

3 An Introduction to the Dynamite Proving System

The Dynamite Proving System is an extension of the PVS theorem prover [12] that interacts with the Alloy Analyzer. Alloy is a formal modeling language well suited for modeling of critical systems. Its simple semantics based on relations and the automated analysis provided by the Alloy Analyzer make Alloy an increasingly accepted lightweight formal method. The analysis provided by the Alloy Analyzer assumes domains sizes are user-bounded, and is therefore partial. This makes the Alloy Analyzer unsuitable for verification of critical models. An alternative would be the use of a theorem prover in order to verify Alloy assertions. Unfortunately, no complete calculus for Alloy was known. In [7] we presented such complete calculus, and extended PVS in order to include the calculus. An appropriate pretty-printer allowed us to present formulas using Alloy notation.

While PVS automatically detects syntactic errors and uses proof techniques in order to (try to) automatize parts of the proofs, some errors many times cannot be detected. We refer to the errors that occur when:

1. An invalid sequent has to be proved.
2. An invalid lemma is introduced.
3. A new hypothesis (which does not follow from the axioms in the current model or the antecedents of the sequent being proved) is added.
4. A necessary formula is incorrectly hidden from a sequent.

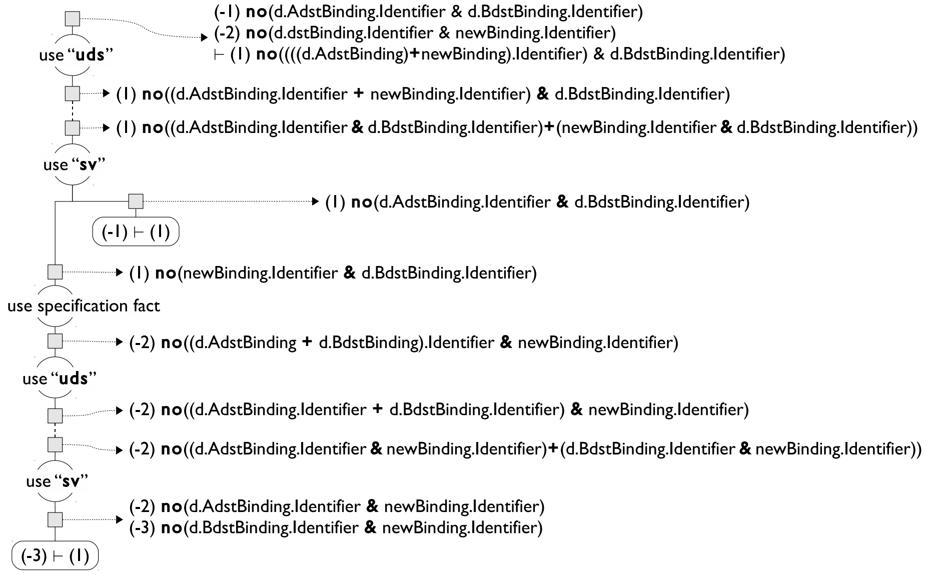
In [7] we deal with the first three situations. In order to reduce the chances of introducing erroneous lemmas or hypotheses, DPS resorts to the Alloy Analyzer. Let us assume we are proving a sequent $\gamma_1, \ldots, \gamma_k \vdash \delta_1, \ldots, \delta_n$, and a new hypothesis φ is introduced using the PVS command `(case varphi)`. According to PVS, we are now left with two sequents to prove, namely,

$$\gamma_1, \ldots, \gamma_k, \varphi \vdash \delta_1, \ldots, \delta_n \quad \text{and} \quad \gamma_1, \ldots, \gamma_k \vdash \delta_1, \ldots, \delta_n, \varphi.$$

It might be the case that φ is overly strong, i.e., it simplifies proving sequent $\gamma_1, \ldots, \gamma_k, \varphi \vdash \delta_1, \ldots, \delta_n$, but sequent $\gamma_1, \ldots, \gamma_k \vdash \varphi$ (which allows us to discharge the newly added hypothesis) is not valid. In order to detect such situations, an Alloy model is automatically created. The model contains, as an assertion to be checked using the Alloy Analyzer, the formula

$$\left(\bigwedge_{1 \leq i \leq k} \gamma_i \right) \Rightarrow \varphi.$$

If a counterexample is returned by the Alloy Analyzer, then it is automatically reported by DPS. The existence of a counterexample means that the sequent is not valid and therefore formula φ is too restrictive. In Fig. 3 we show a proof fragment from our case-study where this happens. We have a sequent S of the form $\gamma_1, \gamma_2 \vdash \delta_1$. We then introduce a new hypothesis h, and obtain new goals $\gamma_1, \gamma_2, h \vdash \delta_1$ and $\gamma_1, \gamma_2 \vdash \delta_1, h$. The proof structure for S is:



```
(-1) no(d.AdstBinding.Identifier & d.BdstBinding.Identifier)
(-2) no(d.dstBinding.Identifier & newBinding.Identifier)
⊢ (1) no((((d.AdstBinding)+newBinding).Identifier) & d.BdstBinding.Identifier)
use "uds"
(1) no((d.AdstBinding.Identifier + newBinding.Identifier) & d.BdstBinding.Identifier)
(1) no((d.AdstBinding.Identifier & d.BdstBinding.Identifier)+(newBinding.Identifier & d.BdstBinding.Identifier))
use "sv"
(1) no(d.AdstBinding.Identifier & d.BdstBinding.Identifier)
(-1) ⊢ (1)
(1) no(newBinding.Identifier & d.BdstBinding.Identifier)
use specification fact
(-2) no((d.AdstBinding + d.BdstBinding).Identifier & newBinding.Identifier)
use "uds"
(-2) no((d.AdstBinding.Identifier + d.BdstBinding.Identifier) & newBinding.Identifier)
(-2) no((d.AdstBinding.Identifier & newBinding.Identifier)+(d.BdstBinding.Identifier & newBinding.Identifier))
use "sv"
(-2) no(d.AdstBinding.Identifier & newBinding.Identifier)
(-3) no(d.BdstBinding.Identifier & newBinding.Identifier)
(-3) ⊢ (1)
```

Lemma "uds" : **all** A: set univ, R,S: A**->**A | R.A **+** S.A **=** (R**+**S).A

Lemma "sv" : **all** A, B: set univ | **no** A+B **<=>** **no** A **&&** **no** B

Specification fact: **all** d: Domain | d.dstBinding = d.AdstBinding + d.BdstBinding

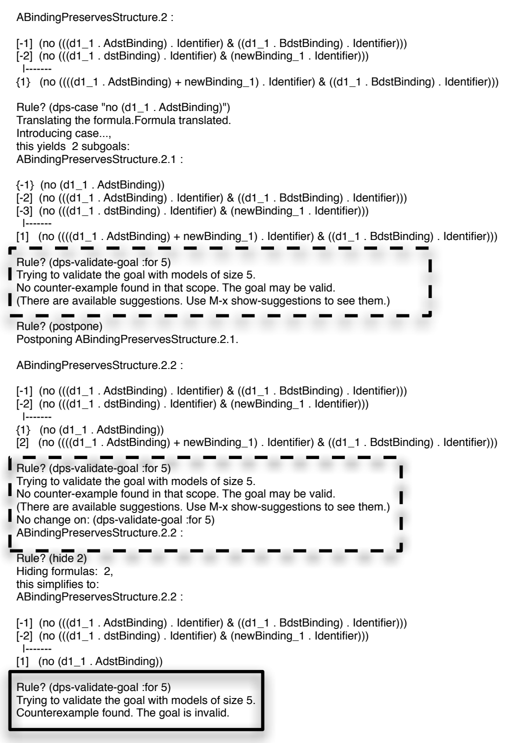
Notice the following:

– Goals `ABindingPreservesStructure.2.1` and `2.2` are validated in Fig. 3 using the Alloy Analyzer. Notice that no counterexamples are found (as reported inside the dashed boxes), and therefore the goals may be correct.
– When goal `ABindingPreservesStructure.2.2` is validated *after* formula 2 is hidden, inside the solid square a counterexample is reported. Notice that hiding formula 2 is a reasonable decision, since we are trying to verify that the introduced hypothesis indeed follows from the sequent antecedents.
– Although not related to the technique, we want to stress the fact that formulas in sequents are actual Alloy formulas.

The counterexample can be used in order to gain a better understanding of the model.

4 Reducing the Proof Search Space Using UnSAT-Cores

In this section we will discuss two techniques to reduce the proof search space during the theorem proving process. The first technique uses an iterative proce-



```
ABindingPreservesStructure.2 :

[-1]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-2]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
{1}   (no ((((d1_1 . AdstBinding) + newBinding_1) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))

Rule? (dps-case "no (d1_1 . AdstBinding)")
Translating the formula.Formula translated.
Introducing case...,
this yields  2 subgoals:
ABindingPreservesStructure.2.1 :

{-1}  (no (d1_1 . AdstBinding))
[-2]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-3]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
[1]   (no ((((d1_1 . AdstBinding) + newBinding_1) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))

Rule? (dps-validate-goal :for 5)
Trying to validate the goal with models of size 5.
No counter-example found in that scope. The goal may be valid.
(There are available suggestions. Use M-x show-suggestions to see them.)
Rule? (postpone)
Postponing ABindingPreservesStructure.2.1.

ABindingPreservesStructure.2.2 :

[-1]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-2]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
{1}   (no (d1_1 . AdstBinding))
[2]   (no ((((d1_1 . AdstBinding) + newBinding_1) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))

Rule? (dps-validate-goal :for 5)
Trying to validate the goal with models of size 5.
No counter-example found in that scope. The goal may be valid.
(There are available suggestions. Use M-x show-suggestions to see them.)
No change on: (dps-validate-goal :for 5)
ABindingPreservesStructure.2.2 :
Rule? (hide 2)
Hiding formulas:  2,
this simplifies to:
ABindingPreservesStructure.2.2 :

[-1]  (no (((d1_1 . AdstBinding) . Identifier) & ((d1_1 . BdstBinding) . Identifier)))
[-2]  (no (((d1_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier)))
  |-------
[1]   (no (d1_1 . AdstBinding))

Rule? (dps-validate-goal :for 5)
Trying to validate the goal with models of size 5.
Counterexample found. The goal is invalid.
```

Fig. 3. A proof fragment where an overly restrictive hypothesis is introduced.

dure in order to remove formulas from sequents. The second technique uses an UnSAT-core in order to determine which formulas can be removed. In Section 4.1 we discuss the iterative technique. In Section 4.2 we present the technique based on UnSAT-core extraction and compare it with the iterative technique. Finally, in Section 4.3 we present some experimental results about the usefulness of the proposed techniques.

Along this section we will assume we are willing to prove a sequent $\Gamma \vdash \Delta$ (where $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$ and $\Delta = \{ \delta_1, \ldots, \delta_m \}$) from a theory Ω containing axioms $\omega_1, \ldots, \omega_n$. In order to reduce the proof search space we will try to remove formulas from Γ, Δ and Ω. Notice that having fewer formulas actually reduces the proof search space. Many proof steps that could depend on the removed formulas (rules for instantiation, rewriting, or applying strategies) are now avoided. This reduces the number of instantiations of inference rules that the theorem prover has to consider, as well as helps the user stay focused on the relevant parts of the sequent.

4.1 An Iterative Technique to Reduce the Proof Search Space

The algorithm in Fig. 4 allows us to determine a set of formulas candidate to be removed. The algorithm attempts to remove each formula φ, and analyzes (using the Alloy Analyzer) whether the sequent obtained *after* formula φ has been removed is valid or not. If the sequent is valid, then φ can be (safely?) removed.

```
algorithm iterative_remove(Gamma, Delta, Omega)
// let Gamma = {g1,...,gk},
// let Delta = {d1,...,dm},
// let Omega = {o1,...,on}.
for i=1 to k do
   if proves(Gamma - gi, Delta, Omega) then
      Gamma = Gamma - gi
   fi
od
for i=1 to m do
   if proves(Gamma, Delta - di, Omega) then
      Delta = Delta - di
   fi
od
for i=1 to n do
   if proves(Gamma, Delta, Omega - oi) then
      Omega = Omega - oi
   fi
od
```

Fig. 4. The iterative algorithm.

Procedure "proves(A, B, C)" (for $\texttt{A} = \{ a_1, \ldots, a_{k_1} \}$, $\texttt{B} = \{ b_1, \ldots, b_{m_1} \}$ and $\texttt{C} = \{ c_1, \ldots, c_{n_1} \}$) checks, using the Alloy Analyzer, whether sequent $\texttt{A} \vdash \texttt{B}$ holds in theory C. In Alloy terms, this amounts to checking, having as facts formulas $c_1, \ldots, c_{n_1}$, the assertion

$$\left(\bigwedge_{1 \leq i \leq k_1} a_i \right) \Rightarrow \left(\bigvee_{1 \leq j \leq m_1} b_j \right). \tag{1}$$

Procedure `proves` returns true whenever the Alloy analysis does not produce a counterexample.

The previous Alloy analysis requires providing a scope for data domains. Therefore, it might be the case that the analysis of formula (1) does not return a counterexample, yet the formula indeed has counterexamples in larger scopes. This shows that this technique is not complete, since a necessary formula might be removed (this explains the question mark on "safely" above) and a valid sequent may no longer be derivable. This is not a problem in itself. Hiding formulas based on the user's intuition is not complete either. Since removing formulas does not allow us to prove previously underivable sequents, refining sequents and theories as explained is a sound rule. In Section 4.3 we will discuss experimental results in order to determine the utility of the technique.

4.2 Using the UnSAT-Core Extraction Feature to Remove Formulas

Some SAT-solvers, such as MiniSat [6] among the ones provided by the Alloy Analyzer, allow one to obtain upon completion of the analysis of an inconsistent propositional theory, an UnSAT-core. An UnSAT-core is a subset of clauses from the original inconsistent theory that is also inconsistent. The UnSAT-core extraction algorithm implemented in MiniSat produces many times small UnSAT-cores. The Alloy Analyzer converts the propositional UnSAT-core into an Alloy UnSAT-core [16] (i.e., a subset of the Alloy model that is also inconsistent if the source model was inconsistent). Notice that the algorithm in Fig. 4 actually computes an Alloy UnSAT-core. Moreover, it computes a *minimal* Alloy UnSAT-core.

Our proposal in order to remove unnecessary formulas when proving a sequent $\Gamma \vdash \Delta$ in a theory Ω (where $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$, $\Delta = \{ \delta_1, \ldots, \delta_m \}$ and $\Omega = \{ \omega_1, \ldots, \omega_n \}$) consists on requesting the Alloy Analyzer an UnSAT-core of the Alloy model whose set of facts is Ω, and the assertion to be checked is

$$\left(\bigwedge_{1 \leq i \leq k} \gamma_i \right) \Rightarrow \left(\bigvee_{1 \leq j \leq n} \delta_j \right) .$$

Upon extraction of the UnSAT-core, the Alloy Analyzer highlights those formulas from $\Gamma \cup \Delta \cup \Omega$ that are part of the UnSAT-core. We propose (as a strategy to use in the proving process) to remove those formulas that are not highlighted. Figure 5 shows a sequent from the running case-study where some of the formulas are highlighted (those that are part of the UnSAT-core). Notice that upon application of rule `dps-hide` (our new proof rule that allows to hide non-highlighted formulas) the formulas that were not highlighted are hidden. In this example (an actual sequent from the case-study) only 5 out of 23 formulas are kept in the sequent upon application of rule `dps-hide`. As with the iterative technique, addition of rule `dps-hide` makes the logic incomplete, but still sound.

Since the technique presented in Section 4.1 ends up computing an Alloy UnSat-core, a comparison to the technique presented in this section is mandatory. Notice first that the iterative technique guarantees a minimal Alloy UnSAT-core, while the UnSAT-core extraction presented in this section does not guarantee minimality. This implies that use of the UnSAT-core extraction feature provided by the Alloy Analyzer may include formulas that could be removed. An essential aspect that moved us towards adopting the technique based on UnSAT-cores is the overhead imposed on the theorem proving process. Given a sequent $\Gamma \vdash \Delta$ to be proved in a theory Ω such that $|\Gamma| = k$, $|\Delta| = m$ and $|\Omega| = n$, the iterative algorithm requires $k+m+n$ calls to the SAT-solver. On the other hand, obtaining the UnSAT-core requires a single call to the SAT-solver. Ideally, the (sequent and theory) refinement process must be applied at each proof step where the

```
[-1]  NonloopingDomain[d1]
[-2]  no (d1.AdstBinding.Identifier & d1.BdstBinding.Identifier)
[-3]  no (d1.AdstBinding.Identifier & d1.routing.Agent)
[-4]  no (d1.BdstBinding.Identifier & d1.routing.Agent)
[-5]  all i : Identifier | lone (i.(d1.BdstBinding))
[-6]  all i : Identifier | lone (i.(d1.routing))
[-7]  all i : Identifier | lone ((d1.BdstBinding).i)
[-8]  d1.srcBinding = ~ d1.BdstBinding
[-9]  no (Identifier.(d1.BdstBinding) & d1.AdstBinding.Identifier)
[-10] no (d1.routing.Agent & newBinding.Identifier)
[-11] no (d1.dstBinding.Identifier & newBinding.Identifier)
[-12] no (Identifier.(d1.dstBinding) & newBinding.Identifier)
[-13] no (Identifier.(d1.BdstBinding) & newBinding.Identifier)
[-14] no (Identifier.newBinding & newBinding.Identifier)
[-15] all i : Identifier | i in newBinding.Identifier =>
          ((i in Address => i in d.space) && (i in AddressPair => i.addr in d.space))
[-16] d2.endpoints = d1.endpoints
[-17] d2.space = d1.space
[-18] d2.routing = d1.routing
[-19] d2.dstBinding = d1.dstBinding + newBinding
[-20] d2.AdstBinding = d1.AdstBinding + newBinding
[-21] d2.BdstBinding = d1.BdstBinding
[-22] d2.srcBinding = d1.srcBinding
  |-------
{1}   no (d2.AdstBinding.Identifier & d2.BdstBinding.Identifier)
```

Fig. 5. A sequent with a highlighted UnSAT-core.

sequent under analysis has new or fewer formulas. In Section 4.3 we will show experimental data supporting the election of this technique.

4.3 Experimental Results

In this section we present some experimental results we have obtained while applying the techniques presented in Sections 4.1 and 4.2. We begin by presenting some statistics about the model being verified. We have verified the model in three different ways, namely:

– Without using any technique for refining the sequents and theories. This corresponds to verification using Dynamite 1.0, as described in [7] (noted as `NoRec` – for *no recommendation* – in Table 1).
– Using the iterative algorithm in order to refine sequents (see Section 4.1). In Table 1 we note this technique as `IterRec` (for *iterative recommendation*).
– Using the UnSAT-core extraction technique presented in Section 4.2. This technique will be noted in Table 1 as `UnsatRec`.

In Table 1 we measure for each technique:

– Length of proofs (measured as the number or rule applications).
– Average number of formulas per sequent.
– Sum of occurrences of formulas in proof sequents.
– At each proof step PVS must consider sentences from the current sequent as well as the sentences from the underlying theory. We then measure the average number of such formulas over the different proof steps.

– Sum (over the proof steps) of occurrences of formulas in proof sequents or from the underlying theories.
– Number of SAT-solver calls for the iterative and the UnSAT-core-based techniques.
– Number of times the UnSAT-core obtained missed a formula necessary for closing a proof branch.
– Number of times the UnSAT-core allowed us to remove formulas that were used in the original proof because of an unnecessary detour.

In order to focus on the most relevant data we are ignoring proof steps where we prove Type Check Constraints (TCCs), which in general can be proved in a direct way. Also, we only applied the techniques (either the iterative or the UnSAT-core-based) on 69 proof steps where it was considered relevant to apply the rules. Systematic application of the iterative technique (for instance each time a new proof goal was presented by PVS) would have required in the order of 25000 calls to the SAT-solver. As a general heuristic, we set the scope for all domains (in the calls to the Alloy Analyzer) to 3.

	`NoRec`	`IterRec`	`UnsatRec`
Proofs' length	969	597	573
Average # of formulas per sequent	5.89	6.01	6.20
Occurrences of formulas in proofs (no theories)	5706	3590	3215
Average # of formulas in sequents or theories	34.89	35.01	7.02
Occurrences of formulas in proofs (with theories)	33807	20903	4023
# SAT-solver calls	N/A	770	69
# times UnSAT-core missed formulas	N/A	N/A	1
# times UnSAT-core avoided detour	N/A	N/A	2

Table 1. Measures of attributes for the employed techniques (N/A = not applicable).

Notice that proofs carried out using any of the techniques for sequent and/or theory refinement are about 40% shorter than the original proof.

In the original proof, as a means to cope with sequents' complexity, formulas that were presumed unnecessary were systematically hidden. While the average number of formulas per sequent is smaller for the original proof, having half the proof steps shows that the automated techniques are better focused on the more complex parts of proofs. This is supported by the analysis of the total number of formulas that occur in sequents. The UnSAT-core-based technique uses 56% of the formulas used in the original proof, while the iterative technique uses 63% of the formulas.

Since the underlying theory in the case-study has 29 formulas, the overhead in applying the iterative technique to formulas in the theory was too high. Therefore, the iterative technique was only applied to formulas occurring in the sequents being verified along a proof (we believe this will be the case most times). On the other hand, the UnSAT-core extraction receives the current sequent plus the underlying theory, and automatically refines *also* the theory. This explains the big difference between the average number of formulas involved in proofs (both in sequents and in the supporting theory) using the iterative technique and the UnSAT-core-based technique. Notice that this implies that in each proof step PVS had to consider significantly fewer formulas in order to suggest further proof steps.

Since proofs are shorter and each sequent contains possibly fewer formulas, the total number of formulas occurring in proofs using UnSAT-cores reduces from the original proofs in about 88% (recall that hiding was also used in the original proofs but not in an automated way, and that formulas from the underlying theory were not hidden). For the iterative technique, the number of formulas reduces in about 40%.

While using UnSAT-cores required only 69 calls to the SAT-solver, the corresponding proof steps using the iterative algorithm required 770 calls to the SAT-solver (without making calls for formulas occurring in the underlying theory). Thus, the UnSAT-core-based technique requires under 10% of the calls required by the iterative technique.

Often during the original proof necessary formulas were incorrectly hidden. We do not have precise records of the number of times this happened because those erroneous proof steps (which at the time were not considered important) were most times undone. We only kept track of 9 cases where the `reveal` command was used in order to exhibit a previously hidden formula, but these were just a few of the cases. It is worth comparing with the single case where the UnSAT-core-based technique missed a formula. This missed formula is recovered if instead of using a scope of 3 in calls to the Alloy Analyzer, scope 5 is used.

Recalling that we have proved 5 Alloy assertions, the ones corresponding to assertions `BindingPreservesDeterminism` and `BindingPreservesNonLooping` required fewer formulas during the proof based on UnSAT-cores. This shows that the original proof used unnecessary formulas that were removed using rule `dps-hide`.

A more qualitative analysis of the techniques allows us to conclude that refining sequents and theories using UnSAT-cores leads to a shift in the way the user faces the proving process. Looking at the (usually few) remaining formulas after `dps-hide` is applied helped the user gain a better understanding on the property to be proved.

5 Related Work

In this section we discuss work related to Dynamite on the combination of SAT-solving with theorem proving, and more specific work on the applications of UnSAT-cores. Using SAT-solving in the context of first-order theorem proving is not new. The closest works to Dynamite 1.0 are the thesis [20] and the article [5]. They follow the idea of our 2007 article [7] of using a model generator to look for counterexamples of formulas being proved by a theorem prover. Previous articles such as [19] only focus on using the SAT-solver to prove propositional tautologies and use the resolution proofs provided by the SAT-solver to guide the theorem prover proofs. This is more restricted than Dynamite 1.0 in that Dynamite is not constrained to propositional formulas. The 2009 article [4] introduces Nitpick, which is based (as Dynamite 1.0) on Kodkod [17]. Nitpick, as Dynamite 1.0, helps during the theorem proving process by detecting that a non-theorem is being proved. It is worth emphasizing that none of these articles make use of UnSAT-cores during the proving process. Reducing the number of sentences in sequents has been acknowledged as an important problem by the Automated Theorem Proving community. The tool MaLARea [18] reduces sets of hypotheses using machine learning techniques. Sledgehammer [3], uses automated theorem provers to select axioms during interactive theorem proving. The iterative technique presented in Section 4.1 shows resemblance with [13], but [13] uses the Darwin model finder tool to convert first-order sentences into function-free clause sets. No notion of UnSAT-cores is provided or used. The SRASS system [15] uses the ideas presented in [13] and complements them with a notion of syntactic relevance, but does not make use of UnSAT-cores. Last, theorem proving of Alloy assertions was first considered in [2]. The theorem prover Prioni translated Alloy sentences to first-order logic sentences, and the theorem prover Athena [1] was used on the resulting formula. Notice that the translation removes the relational flavor of Alloy, and therefore Alloy users are confronted with an unfamiliar formalism. While Prioni is a theorem prover for the Alloy language, it does not make use of the Alloy Analyzer to contribute to the proving process.

6 Conclusions and Further Work

In this article we have presented two techniques for the elimination of superfluous formulas in sequents and theories. The iterative technique allows us to remove formulas but is not appropriate in the context of sequents or theories containing many formulas because it requires many calls to the SAT-solver. It is appropriate if we restrict the application of the technique to formulas occurring in sequents and forget about formulas in the supporting theories. To the best of our knowledge, the idea of refining sequents and theories using UnSAT-cores is novel and shows (on the experiments reported) to contribute to produce shorter and more focused proofs.

This article is part of a more ambitious project on using the unsatisfiability proofs produced by the SAT-solver in order to suggest proof steps, but making special emphasis on proof steps that use quantifier-related proof rules. We plan to continue working in this direction. The current DPS 2.0 interface is based on EMACS. We are developing a new interface that shows closer resemblance to the Alloy Analyzer's interface (including exhibiting counterexamples using the graphic capabilities provided by the Alloy Analyzer).

References

1. Arkoudas K., *Type-ω DPLs*, MIT AI Memo 2001-27, 2001.
2. Arkoudas K., Khurshid S., Marinov D. and Rinard M., *Integrating Model Checking and Theorem Proving for Relational Reasoning*, in Proceedings of RelMiCS'03, LNCS, Springer, 2003.
3. Böhme S. and Nipkow T., *Sledgehammer: Judgement Day.* IJCAR 2010. To appear.
4. Blanchette J.C. and Nipkow T., *Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder*, in TAP 2009.
5. Dunets A., Schellhorn G. and Reif W., *Automated Flaw Detection in Algebraic Specifications*, Journal of Automated Reasoning, 2010.
6. Eén N. and Sörensson N., *MiniSat-p-v1.14. A proof-logging version of MiniSat*, September 2006.
7. Frias M.F., López Pombo C.G., Moscato M.M., *Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications.* TACAS 2007: 587-601.
8. Hoare C.A.R., Leavens G.T., Misra J. and Shankar N., *The Verified Software Initiative: A Manifesto*, 2007.
9. Jackson, D.: *Alloy: a lightweight object modelling notation.* ACM Transactions on Software Engineering and Methodology **11** (2002) 256–290.
10. Jackson D., Schechter I. and Shlyakhter I., *Alcoa: the Alloy Constraint Analyzer*, ICSE 2000, 730–733.
11. Kang E., Jackson D., *Formal Modeling and Analysis of a Flash Filesystem in Alloy.* ABZ 2008: 294-308.
12. Owre S., Rushby J.M., Shankar N., *PVS: A prototype verification system*, CADE'92, LNAI 607, Springer-Verlag (1992), 148–752.
13. Pudlák P., *Semantic Selection of Premisses for Automated Theorem Proving*, in Proceedings of ESARLT 2007, pp. 27–44.
14. Ramananandro T., *Mondex , an electronic purse: specification and refinement checks with the Alloy model-finding method*, Formal Aspects of Computing, 20(1), January 2008, pp. 21–39.
15. Sutcliffe G. and Puzis Y., *SRASS a semantic relevance axiom selection system*, 2007. `http://www.cs.miami.edu/~tptp/ATPSystems/SRASS/`.
16. Torlak E., Chang F., Jackson D., *Finding Minimal Unsatisfiable Cores of Declarative Specifications.* FM 2008: 326-341.
17. Torlak E. and Jackson D., *Kodkod: A Relational Model Finder.* TACAS 2007:632-647.
18. Urban J., *MaLARea: a Metasystem for Automated Reasoning in Large Theories*, in Proceedings of ESARLT 2007, pp. 45–58.
19. Weber T., *Integrating a SAT Solver with an LCF-style Theorem Prover*, in Proceedings of PDPAR 2005, ENTCS 144(2), pp. 67-78.
20. Weber T., *SAT-based Finite Model Generation for Higher-Order Logic*, Ph.D. Thesis, TUM, 2008.
21. Zave P., *Compositional binding in network domains*, FM'06, LNCS 4085, 2006, 332–347.