

# Alloy+HotCore: a Fast Approximation to Unsat Core

Nicolás D’Ippolito, Marcelo F. Frias, Juan P. Galeotti, Esteban Lanzarotti, and Sergio Mera

Departamento de Computación, UBA, Argentina,  
{ndippolito, mfrias, jgaleotti, elanzarotti, smeraj}@dc.uba.ar

**Abstract.** Identifying a minimal unsatisfiable core in an Alloy model proved to be a very useful feature in many scenarios. We extend this concept to *hot core*, an approximation to unsat core that enables the user to obtain valuable feedback when the Alloy’s sat-solving process is abruptly interrupted. We present some use cases that exemplify this new feature and explain the applied heuristics. The NP-completeness nature of the verification problem makes hot core specially appealing, since it is quite frequent for users of the Alloy Analyzer to stop the analysis when some time threshold is exceeded. We provide experimental results showing very promising outcomes supporting our proposal.

## 1 Introduction

Alloy [1] is a relational modeling language. Its simplicity, object-oriented flavor and automated analysis support have made this formal language appealing to a growing audience. The Alloy Analyzer wisely transforms Alloy models in which domains are bounded to a fixed scope, into a propositional formula that is later fed to a selected SAT-solver (such as MiniSat [2] or SAT4J). Then, given an assertion to be verified in the model, the Alloy Analyzer attempts to produce a counterexample violating the assertion. If no such counterexample is found, it concludes that the analyzed property holds in the model within the given scopes.

A useful feature of Alloy Analyzer is the *unsat core* highlighting [3]. This feature allows an user to see a (possibly minimal) subset of the model constraints from which the assertion follows. As mentioned in [3], this information helps to mitigate a variety of modeling problems, such as overconstraining the model, using a weak assertion, or setting an analysis scope that is too small.

As the problem of knowing if a given propositional formula is satisfiable is NP-complete [4], it is quite frequent to exceed the time the user is willing to spend in analysis even for small scopes. That is why the Alloy Analyzer supports interrupting the SAT-solving process. The drawback is that when the user aborts the current analysis, he or she obtains no feedback on the verification process. This occurs even when the search space is almost covered.

Due to the analysis interruption, no conclusive answer is given by the SAT-solver on the satisfiability of the propositional formula. Nevertheless, the SAT-solver has spent a (possibly large) amount of time analyzing the input formula,

and has gained valuable knowledge about it. Our main contribution consists on using this knowledge to identify a collection of “problematic” or “hard” clauses, and present them to the user as a potentially unsatisfiable set of constraints. We call this set the *hot core* of the analysis. Intuitively, a problematic set of clauses is a set that adds significant computational cost to the SAT-solving process. Since the worst case scenario for a SAT-solver is to work with an unsatisfiable formula, a computationally hard set of clauses seems a reasonable symptom of unsatisfiability.

HotCore, our proposed extension to Alloy, profiles the SAT-solver execution in order to gather the required information as the solving process takes place. When the user interrupts the analysis, HotCore identifies a set of the most problematic clauses, applies the inverse translation (from propositional formulas to Alloy model) to this set, and shows the highlighted constraints to the user. Since Alloy users are used to the unsat core highlighting, HotCore uses this presentation style.

To the best of our knowledge, the idea of profiling an interrupted SAT-solving process in order to provide feedback to the user does not seem to be explored. We could only find some weakly related work in the direction of visualizing a DPLL run [5], and in the line of incomplete solving methods, like GSAT, Walksat and some applications [6–9].

The structure of this article is as follows. In Section 2 we present a methodological approach to HotCore through several user scenarios. In Section 3 we provide a brief theoretical background to SAT-solving, and we give a detailed explanation of the heuristics we used to identify a hot core. In Section 4 we provide a representative collection of chosen Alloy problems, showing a very promising behavior of HotCore. As we will see, on average HotCore covers above 90% of the unsat core within one fifth of the total solving time. Finally, in Section 5 we draw some final conclusions and suggest future lines of work.

## 2 Motivation

In this section we will present some of the benefits of identifying a hot core, and show how HotCore helps to solve some limitations of the current distribution of the Alloy Analyzer. In order to make it more amenable to the reader, we will do it through a running example. We will assume that the reader is familiar with Alloy’s syntax and semantics. Refer to [1] for a more detailed description of Alloy. We present below signature definitions for doubly linked list-like structures.

```
one sig null {
}
sig Object {
  owner: one Object + null
}
sig Node extends Object {
  next: one Node+null,
  previous: one Node+null
}

sig List extends Object {
  head: one Node+null,
  last: one Node+null
}
```

Signatures and fields are interpreted as sets of atoms and relations between atoms, respectively. Singleton signature `null` represents the *null* value. Signature `Object` includes all possible objects. A field `owner` marks the atom's owner (or `null` if no owner exists). Atoms in signature `Node` form a subset of the atoms in signature `Object`, and have two extra fields (`next` and `previous`) which are intended for storing the next and the previous nodes respectively (or the *null* value in case no next/previous node exists). Similarly, atoms in signature `List` have two extra fields (`head` and `last`). These fields are references to the first node and last node respectively. The following facts constrain atoms and relations.

```

fact head_is_null_iff_last_is_null {
  all list: List | list.head=null <=> list.last=null
}
fact head_last_nullity {
  all list: List | {
    list.head!=null implies list.head.previous=null
    list.last!=null implies list.last.next=null }
}
fact next_prev_symm {
  all node: Node | {
    node.next!=null implies node.next.previous=node
    node.previous!=null implies node.previous.next=node }
}

fact no_node_sharing {
  no l1, l2: List |
  some n: Node | {
    n in l1.head.*next
    n in l2.head.*next }
}

```

These facts state that: 1) head points to *null* if and only if last points to *null*, 2) the first node (respectively the last node) previous field (respectively next field) points to *null*, 3) symmetry between next and previous fields is preserved, and 4) no *Node* atom is shared between two lists. We will refer to these axioms as *list structure facts*.

Next, the model is completed with facts to characterize ownership relations among node and list objects. These facts were inspired by the Spec# programming methodology [10].

```

fact owner_of_head {
  all list: List | list.head!=null implies list.head.owner=list
}
fact owner_of_last {
  all list: List | list.last!=null implies list.last.owner=list
}
fact owner_of_next {
  all list: List | all node: list.head.*next - null |
  node.next!=null implies node.next.owner=node.owner
}
fact owner_of_prev {
  all list: List | all node: list.last.*previous - null |
  node.previous!=null implies node.previous.owner=node.owner
}

```

We will refer to these newly added axioms as *ownership facts*. Now, let us consider the scenario where an Alloy user writes an assertion to verify that every node has its container as owner:

```

assert container_is_owner {
  all list: List | all n: list.head.*next - null | n.owner = list }

```

Running the analyzer with the command `check container_is_owner for 3`, it yields as result that the assertion has no counterexamples within that scope.

As it was previously mentioned, some SAT-solvers provide the feature of *unsat core extraction*. Exploiting this facility, the Analyzer manages to highlight the following subset of the Alloy model.

```

sig List extends Object {
  head: one Node+null,
  ...
}
sig Node extends Object {
  next: one Node+null,
  ...
}

fact next_prev_symm {
  all node: Node | {
    node.next!=null implies node.next.prev=node
    node.prev!=null implies node.prev.next=node }
}

fact no_node_sharing {
  no l1,l2: List | some n: Node | {
    n in l1.head.*next
    n in l2.head.*next }
}

assert container_is_owner {
  all list: List | all n: list.head.*next - null | n.owner = list
}

```

These constraints are marked as a (possibly minimal) cause of unsatisfiability. As no ownership fact was highlighted, the user could conclude that they were not needed to prove the validity of the assertion within the scope. Increasing the analysis scope up to 10 yields the same result. Although the scope has grown, the unsat core remains the same.

From the user point of view, ownership rules must play an important part in constraining the `owner` field. Therefore, he or she may find it hard to believe that the validity of the property does not depend on those axioms.

A further inspection of the highlighted constraints exposes a subtle error in the fact `no_node_sharing`. As we can see, the fact states that *no two lists share the same node*. This is stronger than *no two distinct lists share the same node*.

An overconstrained model represents one of the most common modeling mistakes using Alloy. In the absence of a counterexample, the unsat core highlighting helps the user to validate that the assertion does not follow trivially from the model. A revised definition of fact `no_node_sharing` is given below:

```

fact no_node_sharing {
  no disj l1,l2: List | some n: Node | {
    n in l1.head.*next
    n in l2.head.*next }
}

```

Once again, the user checks the assertion `container_is_owner`, but in this case considering the revised model. The Analyzer finds no counterexample within the scope of 3. But, in this case, the Analyzer highlights several ownership facts as part of the unsat core constraints. This means that ownership facts were used during the analysis to prove the assertion is valid. From the user's point of view, this is closer to the expected behavior of the Analyzer.

As we have seen, an Alloy user could profit enormously from inspecting a given unsat core. We now present a collection of use cases that exhibits some practical applications for HotCore.

**Scenario 1: unsat core approximation** Let us consider the following scenario. In order to gain more confidence about the assertion validity, the user starts analyzing the faulty model using a scope of 10 instead of a scope of 3.

The analysis carried out by the Alloy Analyzer relies on solving a SAT problem. A larger scope leads to a formula with more variables to be solved. Since the analysis time grows (in the worst case) exponentially on the number of variables, analysis may take much longer.

As analysis takes a lot more to conclude the assertion validity, it is not rare that the Alloy user may interrupt the analysis once a certain time bound is reached. This bound may range from seconds to hours. Assume that he or she decides to abort the analysis after 5 minutes. Since the Alloy Analyzer failed to exhaust the search space and to produce a counterexample within this time bound, HotCore highlights some constraints as the hot core of the analysis. Among those highlighted constraints is the overconstrained fact `no_node_sharing`.

As the user interprets this hot core as an approximation to a potential unsat core, he or she inspects more deeply the constraints. During this task, the user detects the flaw in fact `no_node_sharing`, and fixes the constraint. Notice that it was not necessary neither to decrease the analysis scope nor wait until an unsat core was identified.

**Scenario 2: constraint optimization** Another possible use of HotCore consists on optimizing a given formula. It is well known that equivalent Alloy formulas may produce different but equivalent CNF formulas. Given two equivalent CNF formulas, the SAT-solving time may vary enormously depending on factors such as clause ordering, number of variables and average length of clauses, just to name a few of them.

In large and complex Alloy models, it is not always a feasible task to re-write every constraint in a more compact, friendly to the Alloy Analyzer, equivalent constraint. But, knowing those constraints that played a key role during analysis, an experienced Alloy user may re-write those constraints in order to reduce the analysis time.

Going back to our revised model, the Alloy Analyzer still exceeds the 5 minutes bound to prove the validity of the assertion `container_is_owner` in a scope of 10. Once again, the user may interrupt the analysis once the time bound is exceeded. As the model was weakened, more constraints are identified as members of the hot core. Not only several ownership facts, but also the fact `next_prev_symm` is marked.

An experienced user may replace fact `next_prev_symm` with a relational formula with no quantifiers, defining `next:>Node=~(previous:>Node)`. It is easy to see that both facts are equivalent. Rewriting this fact leads to a CNF formula with less variables and clauses. Upon this model re-writing, the Alloy Analyzer is now able to end the validity analysis in less than 5 minutes. Observe that this use of HotCore can be always applied, independently of the assertion validity.

**Scenario 3: gaining more confidence in larger scopes** Although the *small scope hypothesis* [11] argues that a high portion of counterexamples can be found by analyzing an assertion within small scopes, the usual practice shows that most Alloy users tend to try larger scopes as long as the time resources are not

exhausted. From the user’s perspective, a larger scope means a better confidence in the validity of an assertion.

As it was mentioned before, the analysis time grows as the scope increases. In fact, analysis time may scale from seconds to hours with a slight increase of scope. Because of this, although the Alloy Analyzer may have proved that an assertion is valid within a given scope, it may be infeasible to prove the same assertion for a larger scope. Notice that for smaller scopes the Alloy Analyzer will return an unsat core, while for the larger scope, our extension will identify a hot core. If a larger-scope hot core matches a smaller-scope unsat core, an Alloy user can conclude that most of the analysis time was spent dealing with a set of constraints which happen to be unsatisfiable within the smaller scope. This may be used as evidence that the assertion follows from the same premises within the larger scope. Of course this cannot be considered a proof of validity of the assertion for the larger scope. This is just a heuristic to gain more confidence when the Analyzer fails to return a conclusive result within the time bound.

As an example, let us consider the case where the user starts the analysis of the assertion `container_is_owner` in a scope of 20 instead of 10. If the user interrupts the analysis after 10 minutes, he or she can confirm that the 20-scope hot core matches the 10-scope unsat core. Under the previous premises, the user has more confidence on the assertion validity. Observe that, without having the HotCore extension, the user has no elements to produce a hypothesis on the assertion validity when the analysis is interrupted.

### 3 Finding minimal sets of hot clauses

In this section we describe the theory behind the procedure we use to compute the hot core. Let’s recall first the general schema used by the Alloy Analyzer to verify an assertion. Given a fixed bound in the size of the models to explore (which is also known as the *scope* of the analysis), the Alloy Analyzer verifies whether a given assertion follows from a specification by translating the problem to a set of propositional clauses  $S$ . Then the tool runs a SAT-solving process over that set: if the SAT-solver determines that  $S$  is unsatisfiable, that means that the property follows from the specification within the specified scope. Otherwise, a satisfiable assignment for  $S$  is found, and a counterexample for the assertion is constructed using that assignment. Finally, in the case  $S$  is unsatisfiable, a minimal unsatisfiable core  $U \subseteq S$  is computed, and then  $U$  is mapped back to the associated constraints. The unsat core is then presented to the user highlighting the appropriate Alloy formulas.

We first give a global description of the process to compute the hot core. The rest of the section will be devoted to explain it in detail. The general idea follows the same steps used to extract the unsat core. The SAT-solving process starts with an initial clause database  $C$  given by  $S$ , and during the execution this set of clauses is augmented with new clauses, all of them being consequences of  $S$ . Since the user may interrupt the process at any time, we monitor the SAT-solving clause database keeping track of the clauses that are “hard” or

“problematic” to solve. We will give a more precise definition of this condition later. When the SAT-solving process is interrupted, we take a set  $H \subseteq C$  of the most problematic clauses and we find a minimal subset  $H' \subseteq S$  such that all the clauses in  $H$  are consequences of  $H'$ . The set  $H'$  represents a minimal subset of  $S$  which is potentially unsatisfiable. We then proceed as it is done with the unsatisfiable core  $U$ , applying the inverse translation over  $H'$ , (from propositional formulas to Alloy) to identify the hot core.

To determine which are the problematic clauses we take advantage of the heuristics already implemented in the SAT-solver. As we said before, we used Minisat to implement the hot core extraction, but this feature can be extended to any DPLL based SAT-solver that provides a way to measure the “hardness” of a clause. The efficiency of a SAT-solving search procedure depends on having well-tuned techniques to identify problematic clauses, so we base our procedure on those techniques. As several of the available SAT-solvers based on DPLL do, Minisat uses a heuristic to measure how actively a clause participates in the search process, in order to determine the next variable assignment. The idea behind that metric is that a clause with a high activity is a sign of unsatisfiability, and Minisat makes use of that knowledge to guide the backtracking process that searches for a satisfiable assignment. In this way, the aims of the SAT-solving optimizations and the identification of a probable unsatisfiable set of clauses coincide. We take advantage of this scenario to use already developed clause identification techniques that showed to enjoy a good empirical behavior.

### 3.1 Overview of the SAT-solving process

We will concentrate here on SAT-solvers based on the DPLL algorithm [12]. The key characteristics of the algorithm are: backtracking by conflict analysis, clause recording (which is also known as *learning*) and boolean constraint propagation (BCP) (see [13, 14]). We start by giving some definitions.

**Definition 1.** *A conjunctive normal form (CNF) formula  $\varphi$  on  $n$  variables  $x_1, \dots, x_n$  is the conjunction of  $m$  clauses  $\omega_1, \dots, \omega_m$ , each of which is the disjunction of one or more literals. A literal is the occurrence of a variable or its negation.*

Most solvers operate with clauses in CNF. Observe that a formula  $\varphi$  can be thought of as an  $n$ -variable boolean function  $f(x_1, \dots, x_n)$ , where each clause of  $\varphi$  is an implicate of  $f$ .

**Definition 2.** *Given an  $n$ -variable propositional formula  $\varphi$ , the satisfiability problem (SAT) consists in finding an assignment to the associated boolean function  $f(x_1, \dots, x_n)$  that makes the function equal to 1, or otherwise proving that the function is the constant function 0.*

Given a  $n$ -variable propositional formula, the backtracking search algorithm for SAT is implemented by a *search process* that explores the space of  $2^n$  possible binary assignments to the variables. The exploration does not always traverse

the whole state space, but, as shown in [4], this may happen in a worst-case scenario.

The SAT-solving process works by extending a partial assignment of the formula variables. A variable whose binary value has already been determined is considered to be *assigned*; otherwise it is *unassigned*. An *assignment* for a formula  $\varphi$  is a set of assigned variables and their corresponding binary values. An assignment is *complete* when all the variables are assigned. A clause is said to be *unit* if the number of its unassigned literals is one.

We now give a general overview of the DPLL algorithm. For further details, see [12]. Starting from an empty truth assignment, the algorithm explores the assignment space and organizes the search for a satisfying assignment through a *decision tree*. In this way, each node of the tree represents the explicit assignment to an unassigned variable. The process iterates through the following steps:

1. **Search.** The current assignment is extended by deciding a binary value for an unassigned variable. This step involves deciding which unassigned variable to pick, and which value to assign. The search process terminates successfully if all the clauses become satisfied. It terminates unsuccessfully if some clauses are unsatisfied and all possible assignments have been tried.
2. **Propagation.** The current assignment is extended by analyzing the logical consequences of the assignments made so far. This step is known as Boolean Constraint Propagation (BCP), and it is based on unit clauses analysis. This analysis may extend the current assignment, and may also lead to the identification of unsatisfiable clauses, implying that the current assignment is not a satisfying assignment. This is known as a *conflict*, which is handled by the following step.
3. **Learning.** Given an assignment that produced a conflict, an analysis is made to determine a set of variable assignments that implied the conflict. From this set of variables a clause prohibiting that particular assignment is built and added to the clause database. Observe that this *learnt clause* is a consequence of the original set of clauses. This new clause should be thought of as a “witness” of the reason for the conflict, and it avoids regenerating the conflicting assignment that led to the current conflict.
4. **Conflict analysis.** This stage undoes the current assignment, if it is conflicting, so that another assignment can be tried. A conflict analysis is performed here, that identifies the point in the decision tree where precisely one of the literals of the learnt clause becomes unassigned. This is usually referred to as *backjumping* or *non-chronological backtracking* [13].

### 3.2 Activity heuristics in the search process

The search procedure of a modern SAT-solver is usually a complex algorithm. Heuristics are needed to pick the next unassigned variable and to decide a value for it. Decision strategies (that range from randomly selecting variables to more sophisticated heuristics like JW-OS and JE-TS [15]) have a relevant impact over the SAT-solver’s performance (see [15]).

Minisat uses a variation of the so called Variable State Independent Decaying Sum (VSIDS) heuristic (which was originally introduced in the Chaff solver [14], and it is also used in BerkMin [16] with some differences in the updating process). Using VSIDS, a value is associated with each literal. When a clause is added to the database, the value associated with each literal in the clause is incremented. Periodically, all the counters are divided by a constant. The selection process in the search stage picks the variable with the highest associated value.

Minisat uses a variation of this technique, but applied to clauses. When a learnt clause is used in the analysis process of a conflict, its activity is incremented. Inactive clauses are periodically removed. This strategy can be viewed as attempting to satisfy the clauses involved in a conflict, but particularly attempting to satisfy the most *recent* clauses involved in a conflict. In fact, the decision heuristic of Minisat involves decaying the activity of clauses more often than the standard VSIDS heuristic. Benchmarks have shown that this schema responds faster to changes and avoid branching on out-dated variables [2].

Using this heuristic, the clauses with the highest activity values represent the clauses most actively involved in recent conflicts. Since a set of unsatisfiable clauses generates many conflicts, and therefore many conflict clauses, the high activity of a clause can be seen as a potential sign of unsatisfiability. The strategy we use to identify the hot core is to keep a set  $H$  (of fixed size) with clauses with the highest activity. HotCore updates this set every time the search process is triggered. When the user interrupts the SAT-solving process, HotCore uses the last updated set of clauses to calculate a hot core. This strategy can be thought of as an attempt to identify the set of clauses with the best chances to belong to an unsat core *given the current state* of the solving process. In Section 4 we discuss the empirical results obtained when choosing an appropriate size for  $H$ .

### 3.3 Extracting the core

Once we have identified a set  $H$  of potentially unsatisfiable clauses, we want to associate  $H$  with its corresponding Alloy constraints. This will allow us to present to the user a highlighted part of the Alloy constraints representing the hot core, in the same way it is done for the unsat core. To implement the unsat core, the Alloy Analyzer applies an inverse translation that maps propositional clauses to Alloy formulas, and then it highlights them. We would like to reuse this feature and feed this translation with  $H$ , but the problem is that this inverse translation needs *original* clauses, that is, clauses that were in the output of the forward translation (from Alloy formulas to propositional clauses). The set  $H$  we have identified does not necessarily fulfill this requirement, since there may be clauses which are *consequences* of the original clause database. Therefore, we should be able to identify a set  $H'$  of clauses such that a)  $H'$  implies  $H$  and b) all the clauses in  $H'$  originally belong to the initial clause database.

To build  $H'$ , we implemented an extension of the algorithm proposed by Zhang and Malik in [17]. Let us briefly describe the original algorithm. Given an unsatisfiable set of Boolean formulas, the algorithm extracts a subset of this

set that is still unsatisfiable, using the unsatisfiability proof found by the SAT-solver. A SAT-solving process applied to an unsatisfiable set of clauses can be regarded as a resolution process that generates the empty clause. This can also be described with a *resolution graph*. Each node of the graph represents a clause, the root nodes are the clauses in the original set, and the internal nodes represent the learned clauses generated during the solving process. The edges in the graph represent resolution steps. An edge from a node  $a$  to a node  $b$  represents that  $a$  is a resolve source of the node  $b$ . This clearly defines a directed acyclic graph, where there is a node  $e$  that represents the empty clause. Observe that all the root nodes that are not in the transitive fan-in cone that has  $e$  as a vertex are not needed to construct that particular proof, and therefore can be ruled out. The root nodes in the cone represent a minimal subset of the original clauses that are needed *for that particular proof*.

The identified set is not a minimal subset in the general sense, but its size could be much smaller than the original set of clauses. The main advantage of this algorithm is that it scales well on very large instances [17]. This is one of the algorithms used by the Alloy Analyzer (among others, see [3, 18]).

In our case, we do not have an unsatisfiability proof, but a set of clauses  $H$  that *could* lead to the empty clause. Looking at the resolution graph, every clause  $c_i \in H$  defines a cone  $C_i$ , taking the transitive fan-in cone that has  $c_i$  as a vertex. Observe that the set of root clauses in  $C_i$  contains the original clauses needed to deduce  $c_i$  for that particular proof. Therefore we can define the set  $H'$  as the set of root nodes in  $\bigcup_i C_i$ . HotCore implements this idea in a relatively straightforward way, using a simple graph traversal algorithm.

## 4 Experimental results

We are interested in measuring the quality of the unsat core approximation performed by HotCore. To achieve that, HotCore has been evaluated on ten unsatisfiable problems from the Alloy Analyzer distribution. The chosen problems come from a variety of domains and exhibit a wide range of behaviors (they are fully described in [1]). Each of them was ran on Alloy+HotCore for satisfiability in several scopes. The Alloy Analyzer offers three different algorithms to perform the unsat core extraction. All the comparisons were done with respect to the algorithm proposed by Zhang and Malik in [17]. We report the obtained results using scopes whose solving time is below a 10 minutes time bound. All experiments were performed on an Intel Core 2 duo 2.4GHz, with 2GB RAM. HotCore implementation was built on top of the Alloy Analyzer release 4.1.9. HotCore can be downloaded from <http://www.dc.uba.ar/hotcore>.

In order to measure the quality of the unsat core approximation, we have defined two different metrics: *Hit* and *Error*. *Hit* intends to represent how well the hot core approximates the unsat core, and *Error* tries to capture the error in that approximation. We provide a formal definition below.

**Definition 3.** *Given an unsat core  $U$  and a hot core  $H$  we define  $\text{Hit} = \#(H \cap U) / \#(U) * 100$  and  $\text{Error} = \#(H - U) / \#(H) * 100$ .*

We are interested in using these metrics in terms of both propositional clauses and Alloy specification language. Therefore, we will add in *Hit* and *Error* the subscript  $p$  when we measure clauses at the level of propositional formulas, and the subscript  $a$  when we measure characters at the level of Alloy specification.

As we mentioned, HotCore shows the hot core to the user by highlighting a set of constraints, following the unsat core presentation schema. Consequently, evaluating HotCore at the Alloy model level seems a rather obvious choice, since the highlighted constraints conform the user expected feedback. On the other hand, due to the nature of the Alloy Analyzer’s translation, each Alloy constraint may result in several distinct propositional clauses. Additionally, some propositional clauses may not even have an Alloy model counterpart (such as the symmetry breaking predicates [19]). Therefore, measuring a hot core at the proposition formula level should lead to finer-grained results.

Problem	Size	Scope	Unsat	Hot	Hit <sub>a</sub>	Error <sub>a</sub>	Hit <sub>p</sub>	Error <sub>p</sub>	Hot Unsat
lists - reflexive	21	10	153	16	50	0	75	3	10%
lists - symmetric	21	8	8	1	44	0	82	8	13%
hotel2	65	15	5	1	100	0	78	4	20%
hotel4	61	17	166	48	100	0	81	2	29%
lights	20	200	8	4	100	0	96	1	50%
addressBook1h	21	30	117	74	100	0	93	93	63%
ringElection2	27	14	5	4	100	0	98	0	80%
sets2	11	36	444	256	100	0	88	1	58%
mediaAssets	61	30	31	19	91	1	90	1	61%
p306-hotel	40	18	43	31	100	0	91	3	72%
<i>Average</i>									45.6%

**Fig. 1.** Results shown with a stopping criterion of 75% *Hit<sub>p</sub>*

In Figure 1 we show the results we have obtained running HotCore on the above-mentioned problems. The first four columns show the problem description in terms of: name of the problem, model size (in lines of code), scope of analysis and time required to compute the unsat core (in seconds). The fifth column records the instant in which the analysis was interrupted (also in seconds), that is to say, the instant when the hot core was identified. The stopping criterion was set to achieve *at least 75% Hit<sub>p</sub>*. Observe that since the set of most active clauses is updated at every search iteration, we do not have a finer grained control over the coverage. This means that the verification was stopped in the first search iteration where the coverage was beyond that threshold. The remaining columns show *Hit* and *Error* for the identified hot core using the two defined metrics and the rate between the times needed to compute the hot core and the unsat core.

We now discuss the results. Observe that in general we reach 75% *Hit<sub>p</sub>* quite fast: on average, HotCore needed less than one half (specifically 45.6%) of the total solving time. Furthermore, these results were accomplished with a very low *Error<sub>p</sub>*. Let’s look now at *Hit<sub>a</sub>* and *Error<sub>a</sub>*. We expect a high correlation between the coverage at the propositional and the Alloy level, and this is confirmed by the results. The example `addressBook1h` has a particular behavior: there is a

significant difference between the two metrics, with 93  $Error_p$  and 0  $Error_a$ . This is a case where the set of identified propositional clauses outside the unsat core has no Alloy model counterpart. We believe that this may be a case where those clauses were introduced for symmetry breaking (or similar) purposes.

Let’s turn now to the end-user perspective and let’s analyze what happens in terms of the Alloy specification language. We want to show the HotCore results on the same ten problems but setting the stop criterion looking at the Alloy level coverage instead of at the propositional level. As we did for  $Hit_p$ , we set the threshold to 75%  $Hit_a$ . These results are shown in Figure 2. Observe that the time needed to reach 75%  $Hit_a$  was, in general, even less than the case when we used  $Hit_p$  as the metric to stop the solving process. On average, HotCore needed 20.6% of the total solving time. The improvement can be explained in terms of the exponential blow-up in the mapping from Alloy constraints to propositional clauses, together with a permissive behavior in Alloy’s highlighting scheme.

Problem	Size	Scope	Unsat	Hot	$Hit_a$	$Error_a$	$Hit_p$	$Error_p$	$\frac{Hot}{Unsat}$
lists - reflexive	21	10	153	52	100	0	91	3	34%
lists - symmetric	21	8	8	5	97	0	96	7	63%
hotel2	65	15	5	1	97	0	28	12	20%
hotel4	61	17	166	7	99	0	44	3	4%
lights	20	200	8	3	100	0	37	2	38%
addressBook1h	21	30	117	23	91	0	4	95	20%
ringElection2	27	14	5	1	94	0	8	14	20%
sets2	11	36	444	1	100	0	3	14	0.23%
mediaAssets	61	30	31	2	88	0	7	0	6%
p306-hotel	40	18	43	1	98	0	19	17	2%
<i>Average</i>									20.6%

**Fig. 2.** Results shown with a stopping criterion of 75%  $Hit_a$

Figure 3 (left) and Figure 3 (right) respectively show how  $Hit_a$  and  $Error_a$  evolve during time for a subset of the studied examples. The X-axis represents time percentage (in a logarithmic scale) while the Y-axis shows the corresponding metric. Let’s take a look at Figure 3 (left). On one hand, the hot core convergence grows exponentially fast. On the other hand, the convergence curve has very few oscillations, showing a quasi-monotone behavior. Let’s analyze now Figure 3 (right). Observe that after 10% of the total time has flown, the error is below 15% for all the cases.

To sum up, the implemented heuristic has shown to behave successfully in many cases. In order to validate the unsat core approximation, we would also like to find some examples that show the weakness of our method. We could not find this behavior among the known available Alloy examples, so we designed some ad hoc cases specifically headed to make the heuristic fail. We combined two independent Alloy models: one hard satisfiable model together with one unsatisfiable. We expected to verify that when the SAT-solver is working with the satisfiable part of the specification, the most active clauses will not be related to the real unsat core. We used two satisfiable Alloy models presented

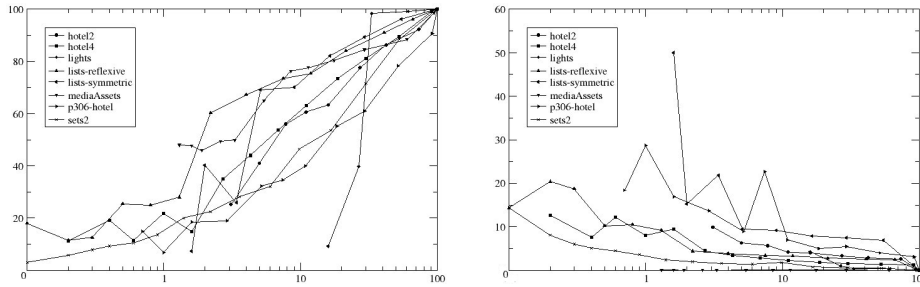


Fig. 3.  $Hit_a$  and  $Error_a$  evolution

in [20] that showed to be computationally hard to solve. We combined them with `RingElection`, known to be unsatisfiable. Our expectations were confirmed, and `HotCore` showed on both examples a  $Hit$  near 0% and a  $Error$  near 100% for almost every search iteration (for both metrics). The hot core coincides with the unsat just in the last search iteration, when around 70% of the total solving time already went by. Nevertheless, the difficulty in finding these examples can be regarded as a good sign of the general behavior of the used heuristic.

To close this section we want to mention some technical details about the implementation. We ran the above mentioned examples varying the size of the set of most active clauses. We have seen that the results do not differ significantly when more than 100 clauses are used. The approximation begins to be less accurate when we use a set with less than 20 clauses. Hence, after several experiments we have chosen 100 to be the default size. This can be seen as a sign that `MiniSat`'s activity heuristic is efficiently tuned, and therefore a relatively small set clauses is enough to be representative. Furthermore, the heuristic we used has almost no computational overhead. The main reason for this is that we build `HotCore` on top of the existing `MiniSat`'s activity heuristic.

## 5 Conclusions and further work

In this article we presented `HotCore`, an extension of the Alloy Analyzer capable of approximating the unsat core of a given set of Alloy constraints. We motivated the use of this tool through several use cases, and we showed that the heuristic we proposed to identify the hot core has a very good empirical behavior. `HotCore` proved to have a nice convergence speed and a low miss rate. Since it is quite frequent to exceed the waiting time bounds during an Alloy analysis, an end-user could obtain a significant profit from using this extension.

Much work rest to be done. On one hand, using a fixed size set of highly active clauses seems a relatively naive approach, and therefore we want to test more refined heuristics. For example, we want to consider the set of clauses that shows a recent activity beyond a given threshold (or a given percentile). On the other hand, the core extraction algorithm we used is quite efficient, but not very accurate in some cases. We would like to evaluate other core extraction algorithms in

the context of HotCore, like the proposed in [19]. We want to investigate other techniques to analyze the information profiled during the solving process. We are also interested in applying the same concept to the Alloy's counterexample generation and visualization. Our idea is to analyze the SAT-solver state when the solving process is interrupted in order to build a *potential* counterexample. For example, the current partial assignment may be a good lead for this purpose.

## References

1. Jackson, D.: Software abstractions: logic, language, and analysis. MIT Press (2006)
2. Een, N., Sorensson, N.: An extensible SAT-solver. Lecture notes in computer science **2919** (2004) 502–518
3. Torlak, E., Chang, F., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. Lecture Notes in Computer Science **5014** (2008) 326
4. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC '71, New York, NY, USA, ACM (1971) 151–158
5. Sinz, C.: Visualizing sat instances and runs of the dpll algorithm. Journal of Automated Reasoning **39**(2) (2007) 219–243
6. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Procs. of the 10th Conf. on Artificial Intelligence. (1992) 440–446
7. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. DIMACS Series in Discrete Mathematics and Theoretical Computer Science (1993)
8. Mazure, B., Saïs, L., Grégoire, É.: A powerful heuristic to locate inconsistent kernels in knowledge-based systems. In: IPMU 96. (1996) 1265–1269
9. Grégoire, E., Mazure, B., Piette, C.: Boosting a complete technique to find mss and mus thanks to a local search oracle. In: Proceedings of IJCAI. (2007) 2300–2305
10. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In Odersky, M., ed.: ECOOP. Volume 3086 of LNCS., Springer (2004) 491–516
11. Andoni, A., Daniliuc, D., Khurshid, S., Marinov, D.: Evaluating the small scope hypothesis (2002) Available at <http://sdg.csail.mit.edu/pubs/2002/SSH.pdf>.
12. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (July 1962) 394–397
13. Silva, J., Sakallah, K.: GRASP – A new search algorithm for satisfiability. In: 1996 IEEE/ACM international conference on Computer-aided design, IEEE Computer Society Washington, DC, USA (1997) 220–227
14. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Design Automation Conference. (2001) 530–535
15. Marques-Silva, J.: The impact of branching heuristics in propositional satisfiability algorithms. Lecture notes in computer science (1999) 62–74
16. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. Discrete Applied Mathematics **155**(12) (2007) 1549–1561
17. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formulas. Proceedings of SAT **3** (2003)
18. Bruni, R., Sassano, A.: Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. ENDM **9** (2001) 162–173
19. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS 2007. 632–647
20. Juan Galeotti, Nicolas Rosner, C.L.P., Frias, M.: Distributed sat-based analysis of object oriented code. In: Proceedings of Symposium on Automatic Program Verification (APV 2009), Rio Cuarto, Argentina, ETH Zurich (February 2009)