Parallel Bounded Verification of Alloy Models by TranScoping*

Nicolás Rosner[1], Carlos Gustavo López Pombo[1,2], Nazareno Aguirre[3,2], Ali Jaoua[4], Ali Mili[5], and Marcelo F. Frias[6,2]

[1] Department of Computer Science, FCEyN, Universidad de Buenos Aires, Argentina
{nrosner,clpombo}@dc.uba.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[3] Department of Computer Science, FCEFQyN,
Universidad Nacional de Río Cuarto, Argentina
naguirre@dc.exa.unrc.edu.ar
[4] Qatar University, Qatar
jaoua@qu.edu.qa
[5] New Jersey Institute of Technology, USA
ali.mili@njit.edu
[6] Department of Software Engineering,
Instituto Tecnológico de Buenos Aires (ITBA), Argentina
mfrias@itba.edu.ar

Abstract. Bounded verification is a technique associated with the Alloy specification language that allows one to analyze Alloy software models by looking for counterexamples of intended properties, under the assumption that data type domains are restricted in size by a provided bound (called the *scope* of the analysis). The absence of errors in the analyzed models is relative to the provided scope, so achieving verifiability in larger scopes is necessary in order to provide higher confidence in model correctness. Unfortunately, analysis time usually grows exponentially as the scope is increased. A technique that helps in scaling up bounded verification is *parallelization*. However, the performance of parallel bounded verification greatly depends on the particular strategy used for partitioning the original analysis problem, which in the context of Alloy is a boolean satisfiability problem. In this article we present a novel technique called *tranScoping*, which aims at improving the scalability of bounded exhaustive analysis by using information mined at smaller scopes to guide decision making at larger ones. In its application to parallel analysis, tranScoping compares different ways to split an Alloy-borne SAT problem at small scopes, and extrapolates this information to select an adequate partitioning criterion for larger scopes. As our experiments show, tranScoping allows us to find suitable criteria that extend the tractability barrier, and in particular leads to successful analysis of models on scopes that have been elusive for years.

* This publication was made possible by NPRP grant NPRP-4-1109-1-174 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

E. Cohen and A. Rybalchenko (Eds.): VSTTE 2013, LNCS 8164, pp. 88–107, 2014.

Keywords: Alloy Analyzer, Parallel analysis, Bounded verification, Parallel SAT-solving.

1 Introduction

Software specification is a crucial activity for software development. It consists of describing software and its intended properties without the operational details of implementations. By specifying software, and especially if one does so prior to implementation, one is able to better understand the software to be developed, and even validate requirements, which would save time and development costs compared to finding flaws in them in later stages of development. The vehicle to specify software is the specification language. Some important characteristics of specification languages are declarativeness, expressiveness and analyzability. Declarativeness and expressiveness allow one to capture requirements more naturally and precisely, while analyzability allows one to better *exploit* specifications by more effectively finding flaws, inconsistencies, etc.

Due to their intrinsic well-defined formal semantics, formal approaches to specification are usually better suited for analysis. Representatives of formal specification languages are, for instance, B [1], Z [9], the Object Constraint Language (OCL), the Java Modeling Language (JML) [3], and Alloy [14]. Some of these languages, B and Alloy in particular, have been designed with *analysis* as a main concern. A main difference between these two languages is that the analysis underlying B's design is *heavyweight* (semi automated theorem proving, essentially), while Alloy favors fully automated analysis. The main analysis technique behind Alloy is *lightweight*, based on boolean satisfiability (SAT). This analysis turned out to be extremely useful in making subtle modeling errors visible, as is evidenced by approaches to the analysis of all the aforementioned specification languages (or, more precisely, fragments thereof) that translate to Alloy in order to profit from the latter's analysis mechanism.

The analysis mechanism implemented by the Alloy Analyzer, the tool associated with Alloy, is *bounded verification*. Bounded verification is a lightweight formal analysis technique that consists of looking for assertion violations of a *model*, under the assumption that the data domains in the model are bounded by a user provided bound (called the *scope* of the analysis). Thus, the absence of errors in the analyzed models is relative to the provided scope, and errors might be exposed in larger scopes. Consequently, confidence in the correctness of models depends on the scope: the larger the scope, the more confident we will be that the specification is correct. That is, achieving verifiability in larger scopes is necessary in order to provide higher confidence on model correctness. Unfortunately, analysis time usually grows exponentially as the scope increases, so approaches to increase the scalability of bounded verification are essential. A technique that helps to increase the scalability of bounded verification is *parallelization*. Essentially, this consists of partitioning the original SAT problem into a number of different independent smaller problems, which can be solved in parallel.

Typically, the speed up obtained by parallelization strongly depends on how the original problem is partitioned. Unfortunately, finding an adequate partition for a problem is difficult; for problems whose sequential analysis takes hundreds of hours, most partitions of the original problem often lead to parallel analyses that still exhaust the available resources (time or memory). In this article, we study the problem of choosing an appropriate partition of a SAT problem, in order to analyze it in parallel. We present a novel technique called *tranScoping*, which consists of examining alternative partitions for small scopes, and extrapolating this information to select an adequate partition for larger scopes. As the experiments presented in Section 5 show, tranScoping indeed allows us to find suitable partitions that make the parallel analysis feasible. Moreover, the experiments in Section 5 deal with problems whose sequential analyses take hundreds of hours, and whose parallel analyses most often timeout as well, but by extrapolating analysis information via tranScoping we can efficiently analyze them. In particular, tranScoping allows us to analyze models on scopes that have been elusive for years. In Section 6 we discuss related work, and finally, in Section 7 we conclude and present some ideas for further work.

2 Bounded Verification: Alloy and the Alloy Analyzer

Alloy is a formal language based on a simple notation, with a simple relational semantics, which resembles the modelling constructs of less formal object oriented notations, and therefore is easier to learn and use for developers without a strong mathematical background. In addition to being a relevant specification language, Alloy has also received attention as an *intermediate* language: there exist many translations from other languages into Alloy. For instance, a translation from JML (a formal language for behavioral specification of Java programs) to Alloy is implemented as part of the TACO tool [11]. A number of tools have also been developed for translating OCL-annotated UML models into Alloy (e.g., [2,15]). Alloy has also been the target of translations from Event-B [17] and Z [16].

There is a good reason for the existence of the above mentioned translations from other languages into Alloy: Alloy offers a completely automated SAT based analysis mechanism, implemented in the Alloy Analyzer [13]. Basically, given a system specification and a statement about it, the Alloy Analyzer exhaustively searches for a counterexample of this statement (under the assumptions of the system description), by reducing the problem to the satisfiability of a propositional formula. Since the Alloy language features quantifiers, the exhaustive search for counterexamples has to be performed up to certain bound in the number of elements in the universe of the interpretations, called the *scope* of the analysis. Thus, this analysis procedure cannot be used in general to guarantee the absence of counterexamples for a model. Nevertheless, it is very useful in practice, since it allows one to discover subtle counterexamples of intended properties, and when none is found, gain confidence in the validity of our specifications. The existence of the many translations from other languages into Alloy provides evidence of the usefulness of the Alloy Analyzer's analysis in practice.

```
module addressBook

abstract sig Target {}
sig Addr extends Target {}
sig Name extends Target {}
sig Book { addr: Name -> Target }

fact Acyclic { all b: Book | no n: Name | n in n.^(b.addr) }

pred add [b, b': Book, n: Name, t: Target] { b'.addr = b.addr + n -> t }

fun lookup [b: Book, n: Name]: set Addr { n.^(b.addr) & Addr }

assert addLocal { all b,b': Book, n,n': Name, t: Target |
                  add [b,b',n,t] and n != n' => lookup [b,n'] = lookup [b',n'] }

// This command should produce a counterexample
check addLocal for 3
```

Fig. 1. An Alloy example: the addressBook sample model from [14, Fig. 5.1]

Let us introduce the Alloy language by means of an example, which will also serve the purpose of explaining how the Alloy Analyzer performs its analyses. Consider the address book example from [14, Fig. 5.1], presented in Fig. 1. In this example, an Alloy model of an address book, consisting of a set of known people and their corresponding addresses, is proposed. Let us go through the elements of an Alloy model. Alloy is a rich declarative language. It allows one to define data domains by means of *signatures*, using the keyword "`sig`". An `abstract signature` is one whose underlying data set contains those objects belonging to *extending* signatures. In the example, the data domain associated with signature `Target` is composed of the union of the (disjoint) domains `Addr` and `Name`. Signatures are, in some sense, similar to classes, and may have fields. For instance, signature `Book` has a field named `addr`, which represents the mapping from names to targets (other names or addresses) that constitutes an address book. According to Alloy semantics, fields are relations. In this case, since "`->`" stands for Cartesian product, `addr` $\subseteq$ `Book` $\times$ `Name` $\times$ `Target`. Axioms are provided in Alloy as *facts*, while predicates (defined using the keyword "`pred`") and functions (defined using the keyword "`fun`"), offer mechanisms for defining parameterized formulas and expressions, respectively. Formulas are defined using a Java-like notation for connectives. Alloy features quantifiers: "`all`" denotes universal quantification, while "`some`" is existential quantification. Terms are built from set-theoretic/relational operators. They include constants (like "`univ`", denoting the set of all objects in the model, or "`none`", which denotes the empty set). Unary relational operators include transposition (which flips tuples from relations) and is denoted by "`~`". Alloy also includes transitive closure (noted by "`^`") and reflexive-transitive closure (noted by "`*`"), which apply to binary relations. Relational union is noted by "`+`", intersection by "`&`", and composition by "`.`".

Fact `Acyclic` in the model specifies that there are no cyclic references in address books (formula "`no n: Name | ...`" is equivalent to "`all n: Name | not ...`"). Predicate `add`, on the other hand, is used to capture an *operation* of the model – the one corresponding to adding a new entry into an address book.

The formula corresponding to this predicate indicates which is the relationship between the pre- and post-states of the address book (referred to as `b` and `b'` in the predicate).

In addition to the described elements, an Alloy model may also have *assertions*. An assertion represents an intended property of a model, i.e., a model that is expected to hold as a consequence of the specification. Assertions can be analyzed, by checking their validity in all possible scenarios within a provided scope. The "`check`" command is used to instruct the Alloy Analyzer on how to analyze an assertion, in particular by specifying the corresponding scope. The Alloy Analyzer translates the model and the assertion of interest to a propositional formula. Notice that the model may include explicit facts (the model axioms), implicit facts (properties that follow from the typing of fields and "subtyping" between signatures), and the assertion to be analyzed. The Analyzer then produces a propositional formula representing the conjunction:

$$\textit{Explicit Facts} \quad \&\& \quad \textit{Implicit Facts} \quad \&\& \quad !\textit{Assert}\,.$$

The translation is made possible due to the finitization information provided by the scopes in the check statement. Notice that if the resulting propositional formula is satisfiable, then the Alloy Analyzer can retrieve a valuation that satisfies the facts, yet violates the assertion (a counterexample showing that the property of interest does not hold in the model). Since the analysis is performed relative to the prescribed scope, a verdict of unsatisfiability only implies that counterexamples do not exist *within the scope*. The assertion under analysis may be false but larger domains may be necessary to exhibit counterexamples.

3 Parallel SAT-Solving

Parallel SAT solving corresponds to the problem of deciding the satisfiability of a propositional formula, by dividing the original problem into smaller instances, and then solving these independently. Parallelization approaches to SAT solving use a divide-and-conquer pattern: problems that are too hard to be tackled directly are split into several (hopefully easier) subproblems, by choosing n propositional variables, and splitting the problem into the 2^n disjoint smaller subproblems, where the chosen propositional variables are instantiated with all possible combinations of boolean values. As we will see, how many non trivial subproblems are obtained, whether they are in fact easier, or how much easier than the parent problem these turn out to be, all strongly depend on the branching variables chosen to partition the search space into disjoint subproblems.

In our case, the splitting process is achieved by means of a mechanism similar to guiding paths [25], with some differences that are worth noting. While one could simply choose n branching variables to split a problem into 2^n disjoint smaller ones, our experience working with CNF formulas arising from the translation of Alloy specifications suggests that the actual number of nontrivial subproblems is usually small compared to the number of subproblems, and often significantly smaller. It is worth it to try and filter out subproblems that can

easily be shown to be trivially unsatisfiable during the splitting process, without ever producing or enqueueing them. For instance, if the n branching variables happen to be part of the same "row" within the representation of a functional Alloy relation, a quick round of boolean constraint propagation will easily discard most combinations, and only the $n + 1$ subproblems where at most one of the variables is true will "pass the filter" and become new subproblems. This is the approach we follow.

Two separate parameters control how problems are split. One of them is a *source* of branching variables, i.e., a criterion determining which sequence of decision variables should be considered (but not how many). The second one is a limit on the number of subproblems to be spawned, i.e., how many new tasks the system is willing to accept. The actual number of nontrivial subproblems may greatly vary depending on which variables are chosen. So, an *a priori* limit on the number of variables to branch is hard to determine. We therefore generate subproblems and solve the trivial ones as part of the same process. The following pseudocode illustrates our resulting approach to splitting a satisfiability problem into subproblems:

```
children = [[]]

while varSource.hasMore() and len(children) < children_limit:

    var = varSource.next()
    newchildren = []

    for litlist in children:
        for newlit in (-var, +var):
            newlitlist = litlist + [newlit]
            if not trivially UNSAT(newlitlist):
                newchildren.append(newlitlist)

    children = newchildren
```

The above described approach to parallel SAT solving is implemented in our prototype distributed solving tool ParAlloy. The parallel analysis experiments featured in this article were run using the latest prototype of ParAlloy, which runs on any cluster of independent commodity PCs. Its main system requirements are a working MPI [6] implementation, a C++ compiler and a Python interpreter. The latest version of the Minisat [5] solver is used at the core of each worker process. Python and mpi4py [7] are used to glue the dynamic aspects of the system together.

The implementation constantly monitors the subproblem solving rate, i.e., the average number of tasks that are proved UNSAT (thus closing a branch of the search space) per unit of time. At regular intervals, said rate is inspected and compared with a threshold, in order to take action if not enough progress is taking place. If the rate is below the threshold, the oldest worker process (whichever has been solving its subproblem for the longest amount of time) is instructed to split that subproblem. In order to keep efficiency rates high, this is also done if the UNSAT rate is above the threshold but there are idle workers

(which implies that all pending task queues are empty). In the current version of the ParAlloy tool, inspection of the UNSAT rate (and possibly corrective action) takes place every 5 seconds, and the UNSAT rate threshold is set at 0.15 per second per worker. For the 68-worker setup used in the parallel analysis experiments shown in Section 5, this means that a progress threshold of 10.2 UNSATs per second is enforced.

4 TranScoping

In this section we present *tranScoping*, the main contribution of the article. *TranScoping* is a new technique for improving the scalability of bounded exhaustive analysis by using information mined at smaller scopes to guide decision making at larger ones. This exploits the regularity often observed across scopes during analysis of an Alloy model.

In this paper we focus on one particular application – that of parallelizing the analysis. For the problem of parallel bounded exhaustive analysis, transCoping compares the performance of different alternative ways of splitting a SAT problem for small scopes, and extrapolates this information to select an adequate splitting approach to be used with larger scopes.

Let us start by introducing the notion of *splitter*, corresponding to a criterion for selecting propositional variables to split a propositional satisfiability problem.

Definition 1. *Given an Alloy model A whose translation to conjunctive normal form (CNF) is a propositional formula P, and a bound b on the number of new subproblems, a* splitter *is an algorithm for selecting propositional variables from P in such a way that the number n of produced subproblems satisfies* $n \leq b$.

Not every variable-selecting algorithm is an appropriate splitter. We require a splitter S to satisfy the following properties:

- *tranScopability*: it must be possible for S to extrapolate how to partition a problem at a larger scope, based on how the problem was partitioned by S at a smaller scope.
- *predictability in a class C of splitters*: if S is the best splitter in C for scope k (the one yielding the partition that can be solved the fastest in parallel), then there exists a scope i ($i < k$), such that S is the best splitter in C for all scopes j such that $i \leq j < k$.

While tranScopability is in general easy to guarantee (we will discuss this property later on, when the splitters are presented), predictability may, on the other hand, be more intricate. In order to understand why, consider, as an example, the model of the mark and sweep garbage collection algorithm provided as part of the Alloy Analyzer's distribution, and the assertion *Soundness2* in it. The sequential analysis times (in seconds) for this assertion are 1, 23, 217 and 2855, for scopes 7, 8, 9 and 10, respectively. Notice that for scope 7 the sequential analysis takes only 1 second. Therefore, all the splitters will generate partitions whose

problems in general will have a very low analysis time, which prevents us from perceiving a clear order if one exists. So, we must consider larger scopes in which the differences between analysis times are easier to perceive. Unfortunately, the analysis time grows quite fast. Already for scope 10, applying all the available splitters (that will be presented in Section 4.1) and analyzing the generated subproblems in order to define an adequate ordering, is too costly. Therefore, we will be limited to the conclusions that we can reach by mining the data obtained for the smallest scopes that are large enough to allow us to differenciate splitters (e.g., for *Soundness2*, scopes 8 and 9). As we will show in Section 5, in the case of *Soundness2*, this is enough to arrive at valuable conclusions.

4.1 A Portfolio of Splitters

Let us now describe an initial collection of splitters, that we assume that satisfy tranScopability and predictability. We will present evidence to this effect when the tranScoping technique is evaluated, in Section 5.

The VSIDS Splitter. VSIDS is a particular decision heuristic that many modern SAT-solvers (including MiniSat) use in order to select the next variable to *decide*, i.e., to be used for splitting (by instantiating it with true and false). The heuristic keeps track of the number of occurrences of a given literal in the formula under analysis, a value that is incremented by a fixed amount whenever new clauses containing the literal are learnt. When a new variable is selected to be decided, the one with the largest VSIDS ranking is chosen. Given k, the maximum number of subproblems to be generated, the VSIDS splitter is defined as follows:

> *Once the underlying SAT-solver is interrupted, select branching variables by considering the ranking of the variable activity score in the solving process, until the number of nontrivial subproblems reaches k.*

For the evaluation in Section 5, in order to compute the VSIDS rank we will analyze the problem sequentially and use the ranking resulting at the end of the sequential analysis. This forces us to use small scopes during the mining phase (otherwise the complete sequential analysis becomes infeasible). Alternatively, we could use an intermediate ranking (for example, the ranking obtained after 10 seconds of analysis), but that would add another dimension to the evaluation, making it too complex for our purposes. For those scopes in which the complete sequential analysis is infeasible, we will use the ranking produced after 5 seconds of SAT-solving (this is the case when analyzing a problem for large scopes after the mining phase). As we will see in Section 5, this limitation does not affect the quality of the analysis of the presented examples (or any other example we used for assessment).

TranScopability is clearly satisfied by VSIDS, since lifting variables from the VSIDS ranking is algorithmic. Notice that there is no direct relationship between the variables selected using VSIDS in a small scope, and the variables selected in larger scopes. As the experiments in Section 5 show, predictability is achieved just by using the same technique.

The "Field" Family of Splitters. Alloy models include *signature fields*. During the process of translating a model to a propositional formula, fields are modeled as matrices of propositional variables. Matrix dimensions are determined by the field typing and the analysis scopes. As an example, consider an Alloy specification containing the following signature declaration:

```
sig Source {
   field : Target
}
```

Suppose we want to analyze the command `check assertion for k but 4 Source, 5 Target`. If the assertion has counterexamples, each counterexample must provide domains $S = \{S_0, S_1, S_2, S_3\}$ and $T = \{T_0, T_1, T_2, T_3, T_4\}$ for signatures `Source` and `Target`, respectively, as well as a binary relation *field* $\subseteq S \times T$, that make the formula corresponding to the assertion satisfiable. The relation *field* is characterized by the following matrix:

$$M_{field} := \begin{array}{|c|c|c|c|c|} \hline p_{S_0,T_0} & p_{S_0,T_1} & p_{S_0,T_2} & p_{S_0,T_3} & p_{S_0,T_4} \\ \hline p_{S_1,T_0} & p_{S_1,T_1} & p_{S_1,T_2} & p_{S_1,T_3} & p_{S_1,T_4} \\ \hline p_{S_2,T_0} & p_{S_2,T_1} & p_{S_2,T_2} & p_{S_2,T_3} & p_{S_2,T_4} \\ \hline p_{S_3,T_0} & p_{S_3,T_1} & p_{S_3,T_2} & p_{S_3,T_3} & p_{S_3,T_4} \\ \hline \end{array}$$

whose entries are propositional variables, and where $p_{S_i,T_j} = \mathit{true} \iff \langle S_i, T_j \rangle \in \mathit{field}$. Different fields have different degrees of relevance on a satisfiability problem, depending on how the fields are involved in the model. So one may consider different fields, to choose variables from these fields' representations in order to partition the SAT problem. Each model field f gives rise to a different splitter. The "Field" family of splitters is defined as follows:

> *select variables from those in matrix M_f, from the bottom-right entry, and towards the top-left, while the number of subproblems does not surpass the given bound k.*

For the above matrix, the order in which variables would be selected is:

$$p_{S_3,T_4}, p_{S_3,T_3}, p_{S_3,T_2}, \ldots, p_{S_0,T_1}, p_{S_0,T_0}.$$

Other Candidate Splitters. Various other splitters have been devised. However, for the case studies assessed so far, the Field family and VSIDS are the most promising ones. The parallel SAT-solver PMSat [12] uses as its variable-selecting heuristic those variables that occur in more clauses. This could give origin to a new splitter by selecting those variables that are more frequently found in the formula. Similarly, one can determine, for a given variable, which are the variables whose decision propagate the value of more literals. A splitter is then defined by selecting those variables that propagate the most.

4.2 Selecting the Right Splitter

Given an Alloy model containing an assertion A to be checked, a splitter S and a bound b on the number of subproblems to generate, S provides an algorithm to select variables to be used in an initial splitting of the (CNF translation of the) model. The splitting produces CNF subproblems $sp_1, \ldots, sp_k$, with $k \leq b$, which can be SAT-solved sequentially ($sp_1; \cdots ; sp_k$), or in parallel ($sp_1 || \cdots || sp_k$).

Once all the splitters are run on scopes $i, i+1, \ldots, j$, we must decide which splitter is going to be used in scopes larger than j. In order to make an informed decision we will store, for each splitter S and scope l ($i \leq l \leq j$), the following information. Given a problem on scope l,

NUM$_{S,l}$ is the number of subproblems generated by splitter S.
MAX$_{S,l}$ is the maximum analysis time incurred by any of the subproblems generated by splitter S.
AVG$_{S,l}$ is the average time required by subproblems generated by S.
SUM$_{S,l}$ is the sum over the analysis times of the subproblems generated by S.
DEV$_{S,l}$ is the standard deviation of the analysis times of the subproblems generated by splitter S.
MED$_{S,l}$ is the median of the analysis times of subproblems generated by S.

Our goal is to convey our insight on how the information about how splitters behave for small scopes has to be interpreted in order to decide which splitter to use for larger scopes (as opposed to defining a *unique* mechanism for ranking splitters based on this information). As we will see in Section 5, based on this information it is often possible to choose a good splitter.

Of the above listed parameters, **MAX** is the most important. A high value of **MAX** (close to the time required to analyze the source problem before being splitted), shows that a child subproblem (the one that has **MAX** as its analysis time) is likely to be nearly as hard to be analyzed as its parent, deeming the splitting performed not useful. On occasion, **MAX** alone is not enough in order to appropriately comparing splitters. This can be observed in Table 1 (see Section 5), where splitters `VSIDS` and `Domain2.dstBinding` alternate their order with respect to **MAX**, as the scope is increased. By looking at the value of the **SUM** parameter in scope 8, one can see that `VSIDS` has a much lower value than `Domain2.dstBinding` (218.29" versus 1678.88"), allowing us to decide between these splitters. A high sum (compared to the other splitters) usually indicates a bad splitting, where subproblems share complex portions of the SAT-solving search space. Therefore, splitters with a high sum are usually demoted to lower positions in the ordering. The most appropriate ordering in this case would then be `VSIDS` < `Domain2.dstBinding`.

It is important to remark that the heuristics just presented allow us to predict the best splitter (within the available set) for each of the case studies to be discussed in Section 5. Moreover, computing the parameters **MAX**, **SUM**, etc. for each splitter in a small scope is inexpensive. We have both a sequential prototype and parallel prototype that can be used interchangeably depending on the availability of the cluster infrastructure, in order to compute these values.

An alternative to the use of the above heuristics for ordering the splitters is to carry out the actual parallel analysis in smaller scopes. This would allow us to rank the splitters according to the parallel analysis times they induce, yielding an ordering that is usually more precise. We will nevertheless stick to the heuristics presented, resorting to parallel analysis in small scopes only if required. Although the latter will not be necessary in this article for detecting the best splitter, we will show in Section 5.5 that performing the parallel analyses yields a better ordering on the whole set of splitters.

5 Experimental Results

In this section we evaluate the heuristics for choosing an appropriate splitter for larger scopes, by analyzing the performance of splitters for smaller scopes. Our evaluation is performed for a number of case studies. For each case study we discuss how the VSIDS and the Field splitters can be ordered, and show that by using the best splitter according to the defined ordering we achieve analyzability in larger scopes. In Section 5.1 we describe the computing infrastructure used in the evaluation; in Sections 5.2–5.5 we present our case studies, and in Section 5.6 we discuss some possible threats to the validity of our experimental results. Since the parallel analysis times depend on the actual scheduling of the queued jobs, we run each experiments 3 times and report the average analysis time. All the times are given in seconds. In all the experiments we set the maximum number of generated subproblems to 256. For each experiment we will report the time required for computing the tranScoping data. This time is almost negligible when compared to the analysis time in the largest scopes. In all the reported experiments we were able to analyze assertions in scopes that were infeasible (analysis would invariably diverge) without tranScoping.

5.1 The Computing Infrastructure

All experiments were run on the CeCAR [26] cluster, which consists of 17 identical quad-core PCs, each featuring two Intel Dual Core Xeon 2.67 GHz processors with 2 MB of L2 cache per core and 2 GB main memory per host. Parallel analyses were run as 17x4 jobs, i.e., 17 nodes running one process per core (1 master + 68 workers). Sequential analyses were run on a single dedicated CeCAR node.

5.2 A Model of Routing in Heterogeneous Networks

In [24], a model of routing in heterogeneous networks is presented. A companion Alloy model can be downloaded from the author's web page. This model is equipped with an assertion, shown in Fig. 2, that could not be checked for some relatively small scopes. As explained before, it is important to analyze model properties on larger scopes, since the larger the analyzed scope, the greater our confidence will be in the validity of the model. This model is very difficult to analyze; its sequential analysis time grows very steeply, from 308 seconds in scope 8

```
assert StructureSufficientForPairReturnability {
   all g: Agent, a1, a2: Address, d1, d2: Domain3 |
      StructuredDomain[d1] &&
      MobileAgentMove[g,a1,a2,d1,d2]
      => ReturnableDomainPair[d1,d2]
}
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 4 Agent, 7 Identifier -- checked
check StructureSufficientForPairReturnability for 2 but
   2 Domain, 2 Path, 3 Agent, 8 Identifier -- checked
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]
```

Fig. 2. Assertion `StructureSufficientForPairReturnability` and its companion checks

to over 15 days in scope 10 (cf. Table 4). Problems like this one require strategies for scaling up bounded analysis, and parallelization could be a valuable tool for it. Still, the parallel analysis technique presented in Section 3 only allowed us to complete the analysis for scopes 1 to 10. In fact, before tranScoping, our repeated attempts to analyze this assertion for scope 11 were unsuccessful. As shown in Table 4, tranScoping allowed us to select splitter `Domain3.srcBinding`, and to analyze successfully the assertion using this splitter.

In order to evaluate which splitter to choose, we started by mining information about the performance of all splitters, for scopes 6 to 8, shown in Table 1. Using this information, we discarded for scope 9 those splitters that stand no chance of becoming best candidates. The possibility of separating viable from inviable splitters is a good quality of tranScoping, since it allows us to reduce the time invested in the data computing phase. It took 868.27 seconds to compute this table. We start by sorting splitters according to **MAX**, as shown in Table 1. This is insufficient to decide an adequate splitter. In particular, observe the ordering between splitters `Domain2.dstBinding` and `VSIDS` (the same applies to the ordering between splitters `Domain2.dstBinding` and `Domain.routing`). For scope 8, `Domain2.dstBinding` < `VSIDS` with respect to **MAX**, but by looking at value **SUM**, we see that `Domain2.dstBinding` has a **SUM** that is 7.7 times larger than `VSIDS`' **SUM**. The difference is large enough to justify promoting `VSIDS` above `Domain2.dstBinding`. This decision is backed up by Table 2, which shows the performance of each of the splitters in the parallel analysis of the assertion. A timeout (TO) was set at 600 seconds. Notice that the best two splitters (according to tranScoping) performed better than the others. At first sight the two best splitters seem to have performed similarly. In fact, `Domain3.srcBinding` performed better than `Domain3.BdstBinding`, as we expected. Not because the former took 1 second less to finish the analysis (that difference might even be reverted if more analyses were made before averaging the results), but because the number of subproblems that it had to generate (see the **UNSATs** column in Table 2) is definitely smaller than the number of subproblems generated by the latter. This has a direct correlation with the **MAX** value: a larger **MAX**

Table 1. Routing: mined tranScoping information, scopes 6 to 9, sorted by **MAX**

Scope	Splitter	NUM	MAX	AVG	SUM	DEV	MED
6	Domain3.srcBinding	77	0.08	0.02	1.45	0.02	0.01
	Domain3.BdstBinding	77	0.09	0.02	1.50	0.02	0.01
	Domain2.dstBinding	192	0.18	0.04	7.67	0.03	0.03
	Domain.routing	102	0.21	0.02	1.98	0.03	0.01
	VSIDS	228	0.49	0.01	3.55	0.05	0.00
	Domain3.AdstBinding	192	1.22	0.05	9.78	0.10	0.02
	Identifier_remainder	64	2.31	0.73	46.94	0.52	0.59
7	Domain3.BdstBinding	136	0.84	0.12	17.00	0.18	0.08
	Domain3.srcBinding	141	0.90	0.10	14.60	0.19	0.06
	VSIDS	140	3.39	0.13	19.18	0.38	0.01
	Domain2.dstBinding	192	3.71	0.49	94.74	0.42	0.32
	Domain.routing	192	4.46	0.14	27.51	0.40	0.04
	Domain3.AdstBinding	192	13.05	0.53	101.28	1.04	0.23
	Identifier_remainder	128	25.97	7.45	953.82	6.41	4.93
8	Domain3.srcBinding	136	8.09	1.13	154.17	1.37	0.51
	Domain3.BdstBinding	136	18.06	1.28	173.48	1.95	0.72
	Domain2.dstBinding	192	36.25	8.74	1678.88	7.45	5.78
	VSIDS	174	63.62	1.25	218.29	6.27	0.05
	Domain.routing	192	89.41	2.18	418.04	7.66	0.39
	Domain3.AdstBinding	192	288.79	10.18	1954.07	22.36	2.46
	Identifier_remainder	256	376.70	86.03	22024.53	81.98	56.98
9	Domain3.srcBinding	365	7.57	163.47	2764.89	15.53	3.68
	Domain3.BdstBinding	272	13.25	360.04	3603.38	27.38	5.89

Table 2. Routing: parallel analysis time, scope 9, all splitters. Timeout (TO) set to 600 seconds.

Splitter	Time	Pending	UNSATs
Domain3.srcBinding	171.30	0	1562
Domain3.BdstBinding	172.23	0	2117
VSIDS	350.39	0	5974
Domain.routing	562.74	0	4534
Domain2.dstBinding	TO	11709	735
Domain3.AdstBinding	TO	17268	475
Identifier_remainder	TO	7682	32

value implies that there are some subproblems that are more complex and have to be split more times (thus causing a larger number of UNSATs) in order to be tamed. In this case this is not reflected in the analysis times because the hardware available was able to cope with the number of subproblems generated by both splitters. Table 3 reports the parallel analysis times for these two splitters in scope 10, where the better performance of `Domain3.srcBinding` can be clearly appreciated.

By using tranScoping we are able to analyze the assertion for scopes 1 through 11, as Table 4 shows. We set a timeout (indicated as TO when reached) of 15 days. The sequential analysis for scope 10 did not finish in 15 days. Looking at the progression of sequential values, it is clear that the sequential analysis for scope 11 may take most probably over a year. Therefore, we use the notation $\gg$ to indicate that the actual speed-up is most probably much larger than the indicated speed up. We do not report parallel analysis times for scopes 6 and 7 because the sequential time is too small and the problem is solved before even being split.

Table 3. Routing: comparing splitters `Domain3.srcBinding` and `Domain3.BdstBinding` during parallel analysis, scope 10

Splitter	Time	Pending	UNSATs
Domain3.srcBinding	1053.48	0	10231
Domain3.BdstBinding	1129.49	0	10884

Table 4. Sequential versus parallel analysis time, and speed-up obtained by using the best tranScoped splitter: Domain3.srcBinding. Timeout (TO) = 15 days.

Scope	6	7	8	9	10	11
Sequential time	1.60	18.34	308.26	76168.16	TO	TO
Parallel time	-	-	26.55	171.30	1053.48	10949.72
Speed-up			11X	444X	>1230X	$\gg$ 118X

5.3 A Model of the Mark and Sweep Garbage Collection Algorithm

Mark and Sweep is a garbage collection algorithm that, as its name conveys, traverses the memory marking those objects reachable from the memory heap, and then sweeping those objects that are no longer reachable. An Alloy model of the mark and sweep algorithm comes as a sample model with the Alloy Analyzer's distribution. Among the assertions to be checked we have `Soundness2`. Unlike assertion `Soudness1` in the same model (whose analysis time grows slowly as the scope increases), assertion `Soundness2` is hard to analyze (Table 7 shows a growth in the analysis time of at least 10 times from a scope to the next).

We also start with this case study by mining information about the performance of all splitters, for scopes 7 to 9, ordered by **MAX**, and reported in Table 5. It took 1007.41 seconds to compute this table. While splitter `VSIDS` appears to be the best option in scope 7, splitter `HeapState.marked` takes a clear lead in scopes 8 and 9. Moreover, as shown in Table 6, the information mined extrapolates to the parallel analysis: `HeapState.marked` is the best splitter and `VSIDS` comes in second place. Table 7 shows that, resorting to the tranScoped splitter `HeapState.marked`, we are able to analyze assertion `Soundness2` for scopes 1 to 10, obtaining significant speed-ups.

5.4 A Model of the *Mondex* Electronic Purse

Mondex is a smart card electronic cash system owned by Master Card. A Mondex smart card allows its owner to perform secure commercial transactions and offers features similar to those provided by ATM machines (albeit with greater mobility). An Alloy model of the Mondex electronic purse is provided and analyzed in [19]. Among the many assertions to be verified, there is assertion `Rab_archive`. Table 8 displays the tranScoping information for this assertion. It took 1145.74 seconds to compute this table. The sequential time required to analyze the assertion in scope 4 is 3.62 seconds. Such short time compresses all the information for the different splitters, preventing us from ordering the splitters precisely. Still, we can at least separate those splitters whose application is bound to be

Table 5. Mark&Sweep: mined tranScoping information, scopes 7 to 9, sorted by **MAX**

Scope	Splitter	NUM	MAX	AVG	SUM	DEV	MED
7	VSIDS	154	0.12	0.03	4.14	0.02	0.02
	HeapState.marked	252	1.75	0.03	8.50	0.11	0.03
	HeapState.left	192	3.36	0.43	82.97	0.41	0.31
	HeapState.freeList	164	4.39	1.49	245.29	0.57	1.38
	HeapState.right	192	4.44	0.46	87.94	0.49	0.32
8	HeapState.marked	254	0.30	0.07	17.67	0.06	0.05
	VSIDS	200	2.32	0.19	38.90	0.25	0.12
	HeapState.right	162	34.54	5.65	914.84	7.13	2.78
	HeapState.left	162	45.38	5.42	877.34	7.17	2.73
	HeapState.freeList	146	50.06	24.45	3570.58	8.32	22.68
9	HeapState.marked	254	1.73	0.21	54.65	0.28	0.12
	VSIDS	181	7.78	0.85	154.07	1.06	0.41
	HeapState.freeList	182	260.93	131.26	23890.37	42.94	131.95
	HeapState.right	200	272.34	32.42	6483.97	42.99	14.96
	HeapState.left	200	301.02	31.43	6285.75	42.22	15.32

Table 6. Mark&Sweep: parallel analysis time, scope 9, all splitters. Timeout (TO) set to 600 seconds.

Splitter	Time	Pending	UNSATs
HeapState.marked	9.95	0	128
VSIDS	184.88	0	2472
HeapState.left	TO	16491	726
HeapState.right	TO	17064	699
HeapState.freeList	TO	7201	1575

expensive. For instance, out of the 16 splitters in Table 8, only 5 seem to have a chance of producing good parallel analyses. The tranScoping data collected for these 5 splitters in scopes 5 and 6, allows us to conclude that the best candidate to use in larger scopes is `VSIDS`. In effect, in scope 6 `VSIDS` has a substantially lower **SUM** than the other 4 splitters, while having a comparable (even smaller) **MAX** as well. The results in Table 9 confirm our prediction, by showing that for scope 6 `VSIDS` produces a better parallel analysis. Table 10 shows that, resorting to the tranScoped splitter `VSIDS`, we are able to analyze assertion `Rab_archive` for scopes 1 to 8. Notice that while the speed-up obtained is modest, it is the best speed-up that can be obtained with these splitters. Better analyses are perhaps possible, but they require to devise new splitters that perform better than `VSIDS`.

5.5 An Alloy Specification of the XPath Data Model

XPath [23] is a language for querying XML documents. In [22], an Alloy model for the XPath 1.0 data model is presented. Subelements inside an XML element cannot be duplicated. As part of the model, assertion `nodup_injective`, states the equivalence between two distinct ways of expressing this fact.

Table 11 reports the values computed for the different parameters in scopes 6 and 7, for the XPath case study. It took 609.02 seconds to compute this data. Based on the retrieved information, some of the splitters can be immediately

Table 7. Mark&Sweep: parallel analysis time and speed-up obtained by using the best tranScoped splitter, HeapState.marked

Scope	6	7	8	9	10
Sequential time	0.25	1.37	22.98	217.31	2855.30
Parallel time	-	-	10.13	9.95	28.35
Speed-up			2X	21X	100X

Table 8. Mondex: mined tranScoping information, scopes 4 to 6, sorted by **MAX**

Scope	**Splitter**	**NUM**	**MAX**	**AVG**	**SUM**	**DEV**	**MED**
4	common/TransferDetails.from	149	1.20	0.38	56.58	0.26	0.31
	common/TransferDetails.to	149	1.82	0.84	124.58	0.41	0.85
	a/AbPurse.abLost	256	2.80	0.35	88.82	0.27	0.29
	common/TransferDetails.value	256	2.84	1.86	475.69	0.41	1.92
	c/ConPurse.status	256	3.04	0.69	176.87	0.93	0.17
	cw/ConWorld.archive	256	3.19	0.12	30.81	0.31	0.02
	c/ConPurse.nextSeqNo	256	4.00	0.69	177.98	1.00	0.16
	cw/ConWorld.ether	128	4.21	0.91	117.11	1.00	0.52
	c/PayDetails.toSeqNo	149	4.39	1.44	215.03	1.08	1.33
	c/PayDetails.fromSeqNo	149	4.46	1.72	255.81	1.14	1.62
	c/ConPurse.pdAuth	256	4.55	2.15	549.94	0.38	2.08
	a/AbPurse.abBalance	256	4.61	0.56	144.19	0.61	0.42
	VSIDS	184	4.84	0.14	25.50	0.43	0.01
	cw/ConWorld.conAuthPurse	224	5.57	0.28	63.38	0.60	0.05
	c/ConPurse.exLog	256	6.16	0.80	204.89	1.02	0.35
	c/ConPurse.balance	256	9.92	1.34	342.87	1.06	1.18
5	common/TransferDetails.from	131	16.05	7.52	984.80	3.46	7.04
	VSIDS	138	28.08	1.77	244.26	4.44	0.09
	cw/ConWorld.conAuthPurse	200	36.92	1.94	388.25	5.46	0.12
	a/AbPurse.abLost	256	39.81	2.90	742.30	5.66	1.50
	cw/ConWorld.archive	256	49.69	2.72	696.12	5.39	0.79
6	VSIDS	176	202.18	2.83	498.34	21.12	0.048
	common/TransferDetails.from	151	206.73	89.23	13473.59	38.86	90.19
	a/AbPurse.abLost	256	423.67	20.37	5215.74	62.26	5.02
	cw/ConWorld.conAuthPurse	164	506.25	12.34	2024.09	51.73	0.35
	cw/ConWorld.archive	256	559.32	40.79	10442.80	66.75	16.36

ruled out as best candidates in larger scopes. This is the case for instance for splitters `Name.NSName`, `Node.stringvalue`, `Name.Localname`, `PI.expanded_name` and `PI.target`, whose **SUM** value is much larger than those for the other splitters. The remaining splitters (those that were not discarded) are listed in Table 12, and their parallel analysis times are reported along other useful information. In this table, splitters are listed in the order inferred from Table 11, following the heuristics discussed in Section 4.2. Notice that the ordering thus determined is flawed; splitter `VSIDS` appears in a better place than it should. At the end of Section 4.2 we proposed to perform the parallel analysis in a small scope in order to tranScope the ordering more accurately. We performed the corresponding analyses for scope 7, and `VSIDS` now falls behind splitter `NodeWithChildren.chseq`, which is consistent with the ordering expected from observing the results reported in Table 12. The results obtained with the selected splitter, and the corresponding speed-up with respect to sequential analysis, are reported in Table 13.

Table 9. Mondex: parallel analysis time, scope 6. Timeout (TO) = 600 seconds.

Splitter	Time	Pending	UNSATs
VSIDS	170.18	0	2185
cw/ConWorld.conAuthPurse	TO	5551	4385
common/TransferDetails.from	TO	5499	4619
cw/ConWorld.archive	TO	13160	2233
a/AbPurse.abLost	TO	9627	2576

Table 10. Mondex: parallel analysis time and speed-up obtained by using the best tranScoped splitter: `VSIDS`)

Scope	6	7	8
Sequential time	456.33	8111.65	149678.26
Parallel time	170.18	1643.91	78685.75
Speed-up	2X	5X	2X

5.6 Threats to Validity

TranScoping is a heuristic for deciding which splitter to use along the analysis of an assertion in a large scope. While we perceive the technique as a breakthrough that allowed us to analyze assertions in scopes in which the analysis (even the parallel one) was previously infeasible, tranScoping is so far only supported experimentally. As such, it requires more experiments. We tried tranScoping in the assertions packed within the sample problems distributed with the Alloy Analyzer as well as in selected interesting models downloaded from the Internet. For assertions whose analyses in large scopes are beyond the capabilities of the Alloy Analyzer, tranScoping gave us useful insights into how to choose a splitter, usually leading to parallel analyzability in larger scopes.

The information compiled in Tables 1, 5, 8 and 11 is based on splitting the root problem just once (with each splitter). Our hypothesis is that a good initial splitting propagates its advantages to the rest of the parallel analysis (or, conversely put, that a bad initial splitting will ruin the parallel analysis altogether). This is confirmed in our case studies, since we were always able to predict the best splitter amongst the ones available in each experiment. But, as discussed in Section 5.5, a more accurate ordering (one not just focusing on the best splitter) is obtained if the complete parallel analysis is performed on the smaller scopes.

The variables selected by the `VSIDS` splitter strongly depend on how long is the analysis allowed to run before observing the ranking. Therefore, different query times may produce quite distinct sequences of variables. This did not prevent tranScoping from predicting the best splitter in the case studies in this article and other examples we ran. Yet we noticed that the different runs of the `VSIDS` splitter (whose times are averaged when reported in the tables), yielded analysis times with significant variation.

Finally, we are presenting a very limited, albeit useful, set of general purpose splitters. Further research has to be conducted in order to identify other general purpose splitters, or new domain-specific ones.

Table 11. XPath: mined tranScoping information, scopes 6 and 7, sorted by **MAX**

Scope	Splitter	NUM	MAX	AVG	SUM	DEV	MED
6	Node.parent	150	0.56	0.18	26.42	0.09	0.17
	VSIDS	166	1.42	0.04	8.02	0.19	0.01
	NodeWithChildren.ch	144	2.99	0.13	18.94	0.28	0.06
	NodeWithChildren.chseq	129	4.12	0.05	6.75	0.36	0.01
	Attribute.name	98	4.51	0.12	11.51	0.48	0.01
	Element.nss	134	4.68	0.10	14.01	0.42	0.02
	PI.expanded_name	135	4.83	0.64	87.65	0.55	0.45
	Element.gi	133	5.24	4.12	548.67	0.83	4.28
	PI.target	135	5.44	0.69	93.61	0.56	0.51
	Name.Localname	150	5.84	2.77	416.77	2.20	4.19
	Node.stringvalue	150	5.88	4.72	708.71	1.03	4.95
	Name.NSName	147	6.22	5.01	735.98	0.38	4.94
7	Node.parent	155	8.51	1.43	222.15	1.38	1.11
	VSIDS	168	53.34	0.84	141.39	4.23	0.02
	NodeWithChildren.ch	192	67.32	0.95	182.98	5.00	0.15
	Attribute.name	99	92.43	1.38	136.37	9.33	0.05
	PI.target	178	109.93	7.40	1317.73	8.70	5.25
	NodeWithChildren.chseq	171	129.51	0.80	137.17	9.90	0.03
	PI.expanded_name	178	134.24	8.24	1466.69	10.21	6.36
	Element.nss	140	201.73	1.73	241.64	17.05	0.02
	Name.Localname	153	235.16	83.57	12786.90	65.44	110.10

Table 12. XPath: parallel analysis time, scope 8, only splitters that are viable candidates according to tranScoping. Timeout (TO) set to 600 seconds.

Splitter	Time	Pending	UNSATs
Node.parent	98.61	0	1231
VSIDS	TO	13160	5698
NodeWithChildren.ch	227.09	0	4456
Attribute.name	286.32	0	1384
NodeWithChildren.chseq	548.66	0	7947
Element.nss	419.45	0	1926

6 Related Work

Parallel bounded verification has been used mainly in the context of program static analysis. For example, [21] proposes to split the program control flow graph and use JForge [10] (a tool for program bounded verification) to analyze each slice. An approach to parallelizing scope-bounded program analysis based on data-flow analysis was presented in [20].

An alternative to tranScoping is the use of a large-scale parallel SAT-solver. Unfortunately, while multi-core tools are starting to take off, *distributed* parallel SAT-solvers are still scarce. CryptoMiniSat2 [8] is an award-winning open source solver with sequential and parallel operation modes. The author also mentions distributed solving among its long-term goals. No public release or other news about this have been announced. GrADSAT [4] reported experiments showing an average 3.27X and a maximum 19.9X speed-up using various numbers of workers ranging between 1 and 34. C-sat [18] is a SAT-solver for clusters. It reports linear speed-ups, but the tool is not available for experimentation. PMSat [12], an MPI-based, cluster-oriented SAT-solver is indeed available for experimentation, but reports generally small speed-ups.

Table 13. XPath: parallel analysis time and speed-up obtained by using the best tranScoped splitter: `Node.parent`

Scope	6	7	8	9
Sequential time	5.15	140.90	2560.17	19559.49
Parallel time	–	23.95	98.61	1473.32
Speed-up		6X	26X	13X

7 Conclusions and Further Work

We presented *TranScoping*, a technique for principled selection of splitting heuristics in parallel bounded verification. This approach exploits information from simple analyses in small scopes of a model under analysis, in order to give the user of the technique the insight necessary to infer an adequate splitter for larger scopes. We evaluated this approach on a number of case studies, showing that by tranScoping we are able to analyze assertions in scopes where we failed before many times. As these experiments show, for many problems the enormous growth of the analysis times causes them to have a bad initial splitting, resulting in diverging analysis. We believe tranScoping is a useful tool, that helps us make an informed decision about the most critical point in the parallel SAT solving analysis process.

TranScoping opens a new research line, namely, the search for new splitters that may produce better speed-ups than the general purpose splitters we presented in this article. Also, it may be possible to find splitters tailored to specific domains (SAT based program analysis, parallel test generation using SAT, etc.). We plan to work on defining and evaluatiing such new splitters.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1), 69–86 (2010)
3. Chalin, P., Kiniry, J.R., Leavens, G.T., Poll, E.: Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 342–363. Springer, Heidelberg (2006)
4. Chrabakh, W., Wolski, R.: GrADSAT: A Parallel SAT Solver for the Grid. In: UCSB Computer Science Technical Report Number 2003-05
5. 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)
6. MPI2: A Message Passing Interface Standard. Message Passing Interface Forum, High Performance Computing Applications 12, 1–2, 1–299 (1998)
7. Dalcin, L., Paz, R., Storti, M., D'Elia, J.: MPI for Python: Performance improvements and MPI-2 extensions. J. Parallel Distrib. Comput. 68(5), 655–662
8. `http://www.msoos.org/cryptominisat2`

9. Davies, J., Woodcock, J.: Using Z: Specification, Refinement and Proof. International Series in Computer Science. Prentice Hall (1996)
10. Dennis, G., Chang, F., Jackson, D.: Modular Verification of Code with SAT. In: ISSTA 2006, pp. 109–120 (2006)
11. Galeotti, J.P., Rosner, N., Pombo, C.L., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: ISSTA 2010, pp. 25–36 (2010)
12. Gil, L., Flores, P., Silveira, L.M.: PMSat: a parallel version of MiniSAT. Journal on Satisfiability, Boolean Modeling and Computation 6, 71–98 (2008)
13. Jackson, D., Schechter, I., Shlyakhter, I.: Alcoa: the alloy constraint analyzer. In: Proceedings of ICSE 2000, Limerick, Ireland (2000)
14. Jackson, D.: Software Abstractions. MIT Press (2006)
15. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 592–607. Springer, Heidelberg (2011)
16. Malik, P., Groves, L., Lenihan, C.: Translating Z to Alloy. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 377–390. Springer, Heidelberg (2010)
17. Matos, P.J., Marques-Silva, J.: Model Checking Event-B by Encoding into Alloy. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 346–346. Springer, Heidelberg (2008)
18. Ohmura, K., Ueda, K.: c-sat: A Parallel SAT Solver for Clusters. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 524–537. Springer, Heidelberg (2009)
19. Ramananandro, T.: Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. Formal Aspects of Computing 20(1), 21–39 (2008)
20. Shao, D., Gopinath, D., Khurshid, S., Perry, D.: Optimizing Incremental Scope-Bounded Checking with Data-Flow Analysis. In: ISSRE 2010, pp. 408–417 (2010)
21. Shao, D., Khurshid, S., Perry, D.: An Incremental Approach to Scope-Bounded Checking Using a Lightweight Formal Method. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 757–772. Springer, Heidelberg (2009)
22. Sperberg-McQueen, C.M.: Alloy version of XPath 1.0 data model, `http://www.blackmesatech.com/2010/01/xpath10.als`
23. World Wide Web Consortium (W3C), XML Path Language (XPath) Version 1.0, W3C Recommendation (November 16, 1999)
24. Zave, P.: Compositional binding in network domains. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 332–347. Springer, Heidelberg (2006)
25. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. 21, 4–6 (1996)
26. `http://cecar.fcen.uba.ar/`