Ranger: Parallel Analysis of Alloy Models by Range Partitioning

Nicolás Rosner
Department of Computer Science,
FCEyN, UBA,
Argentina,
nrosner@dc.uba.ar

Junaid H. Siddiqui
Department of Computer Science,
LUMS School of Science and Engineering,
Pakistan,
junaid.siddiqui@lums.edu.pk

Nazareno Aguirre
Department of Computer Science,
FCEFQyN, UNRC,
Argentina,
naguirre@dc.exa.unrc.edu.ar

Sarfraz Khurshid
Department of Electrical and Computer Engineering,
The University of Texas at Austin,
USA,
khurshid@ece.utexas.edu

Marcelo F. Frias
Department of Software Engineering,
Instituto Tecnológico de Buenos Aires,
Argentina,
mfrias@itba.edu.ar

***Abstract*—We present a novel approach for parallel analysis of models written in Alloy, a declarative extension of first-order logic based on relations. The Alloy language is supported by the fully automatic Alloy Analyzer, which translates models into propositional formulas and uses off-the-shelf SAT technology to solve them. Our key insight is that the underlying constraint satisfaction problem can be split into subproblems of lesser complexity by using *ranges* of candidate solutions, which partition the space of all candidate solutions. Conceptually, we define a total ordering among the candidate solutions, split this space of candidates into ranges, and let independent SAT searches take place within these ranges' endpoints. Our tool, Ranger, embodies our insight. Experimental evaluation shows that Ranger provides substantial speedups (in several cases, superlinear ones) for a variety of hard-to-solve Alloy models, and that adding more hardware reduces analysis costs almost linearly.**

***Index Terms*—Static analysis, Alloy, Parallel analysis, SAT.**

I. Introduction

Declarative formal models of software are valuable at a number of different stages of software development. They are particularly useful during requirements elicitation, as a means to express requirements in a language that is precise and expressive enough to document the needs of stakeholders. Moreover, declarative models allow us to document rationales behind design decisions, and even to analyze properties of software designs prior to implementation. During the implementation phase, declarative models allow programmers to document expected properties of their classes and methods, e.g., using class invariants, method contracts, and loop invariants, which can also be exploited for different kinds of analyses.

A number of modeling languages today allow writing formal, declarative models [18], [35], [26], [21], [2]. Our specific focus is Alloy [18], a declarative extension of first-order logic based on relations. Alloy's concise yet expressive notation, together with its fully automated, SAT-based Alloy Analyzer tool [3], make the language particularly appealing for modeling and analysis. Indeed, Alloy has already been used effectively in requirements [19], [40], design [25], [20], testing [23], [1], and as an intermediate language in static program analysis [10], [8], [13], [14], [28]. Section II provides further details regarding Alloy and its analysis tool.

Alloy's analysis technique, known as *scope-bounded checking*, analyzes a model's correctness with respect to a bounded universe of discourse, by searching for violations of assertions that the user may expect to hold in the model. The assertions on a model are evaluated on model instances whose domains are bounded in size. The bound on the size of model instances is termed the *scope*, and is given by the user. Clearly, assertions that pass the analysis are not necessarily valid in general – they are valid for the given scope. Thus, to enhance their confidence in the correctness of their models, Alloy users must run their analyses for larger scopes. However, the cost of the SAT-based analysis underlying Alloy is exponential in those bounds, so, in many cases, the analysis is limited to small scopes. This might not be an issue if Alloy is used just as a convenient declarative language, with easy-to-use automated analysis, to quickly check the validity of intended properties on small model instances. But the versatility of the language and the significant advances on SAT technology are causing a shift from the above use of the tool to its current use as an expressive specification language with a powerful underlying analysis technique. Thus, Alloy users are continuously demanding more efficiency from the tool, as well as scalability (the possibility of running analyses for larger scopes), without having to resign the declarativity of the language. This is evidenced by the existence of a variety of tools that use Alloy as a backend for sophisticated analyses which push the limits of the Alloy Analyzer, and by the increasing concern on employing suitable Alloy "idioms" in modeling, that allow for more efficient analysis.

Khurshid's work was funded in part by the US NSF grant #CCF-0845628.

Our work in this paper is motivated by the aforementioned demand on scalability of Alloy analysis, and is driven by our desire to effectively leverage the increasing availability of commodity hardware for effective parallelization schemes. We present Ranger, a novel parallel analysis technique for Alloy models, based on what we call *range partitioning*. Essentially, the technique relies on the definition of a linear ordering on the state space of an Alloy model. A partition of the state space can then be defined by splitting the linear ordering into non-overlapping intervals, called *ranges*. Each restriction of the original problem to a particular range thus becomes an independent subproblem, which can be analyzed by a separate processor on a cluster of computers. Further details are presented in Section III. We also discuss some more technical, implementation-related issues in Section IV.

We perform an experimental evaluation of our approach for range partitioning using a benchmark consisting of 10 hard-to-analyze properties from 7 different Alloy models (Section V). The benchmark includes unsatisfiable and satisfiable problems from a variety of problem domains, from protocol specification to complex test input generation. We show that, for 64 workers, the average speedup over the hardest scopes that were still tractable by the Alloy Analyzer within 10 hours was 41.76x, and the maximum such speedup was 205.35x. More importantly, in all cases Ranger was able to push the tractability barrier, successfully handling the assertions for scopes that exceed the capabilities of the Alloy Analyzer; in some cases, it was able to do so for scopes that stand no chance whatsoever of being tractable by the Analyzer.

In Section VI we discuss existing techniques aimed at improving the scalability of Alloy, including parallel analysis tools and techniques related or applicable to Alloy. Finally, our conclusions and some proposals for further work are presented in Section VII.

In summary, this paper makes the following contributions:

- **Range partitioning.** We introduce the idea of distributing an Alloy problem into several subproblems of lesser complexity by defining *ranges* of candidate solutions;
- **Parallel analysis for Alloy.** We present Ranger, our technique for effective parallelization of Alloy problems using dynamic work stealing;
- **Experimental evaluation.** We embody Ranger into a prototype implementation and present experimental results that show the effectiveness of Ranger in analyzing a variety of Alloy models.

II. Alloy and the Alloy Analyzer

Alloy is a declarative modeling language whose syntax incorporates features that are ubiquitous in object orientation. This amenable syntax has a relational semantics whose comprehension requires elementary concepts from discrete mathematics. Formally, Alloy's relational logic is an extension of first-order logic with reflexive-transitive closure.

Let us introduce Alloy's syntax and semantics through an example, corresponding to an Alloy model for a heap allocated binary tree data structure, shown in Figure 1. Data domains are

```
one sig null {}

abstract sig Object {}

sig BinTree extends Object { root : Node + null }

sig Node extends Object { left, right : Node + null }

pred Acyclic[t : BinTree] {
   all n : t.root.*(left + right) |
      n !in n.^(left + right) &&
      (n.(left) & n.(right)) in null &&
      (n != null => (lone n.~(left + right))) }

pred NumNodesEqualsNumEdgesPlusOne[t: BinTree] {
   t.root != null =>
      #(t.root.*(left+right)-null) =
         #(left.Node)+#(right.Node)+1 }

pred NoUnreachableNodes[t : BinTree] {
   t.root.*(left+right) = (Node + null) }

fact { all t : BinTree | NoUnreachableNodes[t] }

check { all t : BinTree |
     Acyclic[t] <=> NumNodesEqualsNumEdgesPlusOne[t]
} for 0 but 1 BinTree, exactly 5 Node
```

Fig. 1. A sample Alloy model for binary trees.

defined using *signatures* (denoted by the keyword `sig`), which are represented as sets. Signature `Node`, for instance, declares a set of node objects. Akin to classes in object oriented languages, signatures may extend other signatures, in which case domains defined by the extending signatures are subsets of the domains defined by the extended ones. A signature may be `abstract`, meaning that its domain only contains elements that belong to its extending signatures. Like classes, signatures may contain fields, which are captured by relations. For example, field `root` denotes a (total and functional) binary relation contained in `BinTree` $\times$ (`Node` $\cup$ `null`). It is worth emphasizing that Alloy fields may also denote relations of arity greater than 2. Predicates allow us to name properties, while functions name terms. They may be combined to write axioms, which are called *facts* in Alloy.

Alloy expressions are built using set-theoretical and relational operators and constants. Constants `univ`, `iden` and `none` denote the set containing all elements, the identity binary relation on such set, and the empty set, respectively. Operations $+$, $-$ and & denote set union, difference and intersection, respectively. Relational operators include composition (called *navigation* in Alloy), transpose, and (reflexive-)transitive closure. They are defined as follows:

$$R.S = \{\langle a_1, \ldots, a_{n-1}, b_2, \ldots, b_m \rangle \ : \ a_n = b_1 \wedge \langle a_1, \ldots, a_n \rangle \in R \wedge \langle b_1, \ldots, b_m \rangle \in S\} \text{ (navigation)}$$

$$\sim R = \{\langle b, a \rangle \ : \ \langle a, b \rangle \in R\} \text{ (transpose)}$$

$$\hat{\ } R = \bigcup_{i>0} R^i \text{ (transitive closure)}$$

$$*R = \bigcup_{i \geq 0} R^i \text{ (reflexive-transitive closure)}$$

We define $R^0 = \texttt{iden}$ and $R^i = \overbrace{R.\cdots.R}^{i\text{ times}}$. Transpose and closures are only defined for binary relations in Alloy.

Function `#` computes the cardinality of a relation. For example, term `#(t.root.*(left+right)−null)` in predicate `NumNodesEqualsNumEdgesPlusOne` denotes the number of nodes reachable from the root of tree `t` by traversing `t` along fields `left` and `right`. Expression `lone` r requires relation r to have at most one element.

Alloy formulas are built from the atomic predicate `in` (inclusion), using standard connectives from first-order logic. Java notation is used for propositional connectives. Quantifiers are denoted by `all` (universal) and `some` (existential).

Our sample model also includes an assertion – a property that is expected to hold in valid model instances (i.e., those satisfying the structural constraints imposed by signature definitions, and facts). As explained in Section I, assertions are a means to verify model correctness, and are analyzed using the Alloy Analyzer within user-prescribed bounds. Check commands are issued in the model, and set the bounds for data domains. In this example, the Analyzer will analyze all configurations with at most one tree and exactly 5 nodes (and zero elements for other domains, except `null`, which is constrained in the model to have exactly one element).

III. Range Partitioning

In this section we present *range partitioning*, our new technique for parallel analysis of Alloy models. As we will show later on, this technique provides a substantial improvement to the scalability of the SAT-based analysis, compared to the sequential Alloy Analyzer.

The technique essentially consists of the following stages:

- Given an Alloy model, all candidate configurations that would be explored by the Analyzer are linearly ordered. This step establishes a linear ordering $C_1, C_2, \ldots, C_n$ (notice that the number of configurations, although usually very large, is finite due to the imposed bounds).
- Some arbitrary configurations $C_{j_1}, C_{j_2}, \ldots, C_{j_i}$ are selected, and the aforementioned ordering is split into ranges using those arbitrary configurations as partition points (note that we do not demand these configurations to satisfy the model axioms). We then obtain ranges $[C_1, C_2, \ldots, C_{j_1})[C_{j_1}, \ldots, C_{j_2}), \ldots, [C_{j_i}, \ldots, C_n]$.
- The Alloy model is constrained, yielding models that correspond to the different ranges. These models are distributed to different processors and analyzed in parallel.

The above described process requires addressing the following technical challenges:

- Define a linear ordering on the set of candidate configurations.
- Provide an algorithm for selecting appropriate configurations as partition points (ideally, we want ranges to contain roughly the same number of configurations).
- Solve the distribution problem in an efficient way.

We will deal with each one of these challenges in Sections III-A–III-C.

A. A Linear Ordering on Configurations

In order to explain how a linear ordering on configurations can be defined, let us first describe how configurations are internally handled by the Alloy Analyzer. This tool translates models to Kodkod's [37] language. From the scopes in the check command, Kodkod generates a uniform naming for domain elements, or atoms, as these are called. For the example in Fig. 1, Kodkod produces the naming in Table I.

TABLE I
Naming table for the model from Fig. 1.

Sig	scope	naming
`Object`	1	`Object$0`
`null`	1	`null$0`
`Node`	5	`Node$0`,...,`Node$4`

From the naming and other information (see Section IV-A), Ranger builds a vector specification, i.e., a mapping

$$vecSpec : AtomNames \times RelNames \rightarrow \mathcal{P}(AtomNames)$$

that, given an atom name n and a relation name R, retrieves the atom names that may be considered as the result of computing $n.R$. Hence, a vecSpec is used to capture the state space of configurations. For the sake of simplifying the presentation, we will restrict ourselves to total and functional signature fields. Notice that, in the example from Fig. 1, all fields satisfy this constraint. Figure 2 provides a graphical representation of the vecSpec associated with the model from Fig. 1.

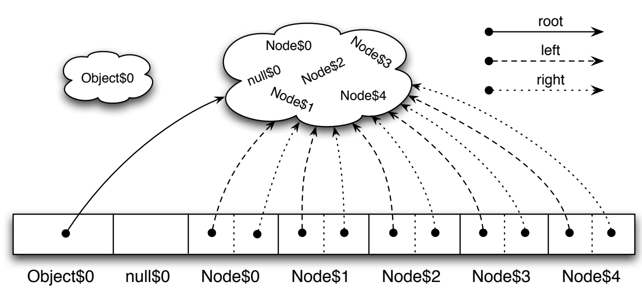

Fig. 2. Graphical representation of a vecSpec.

In the figure, `Object$0` may relate to `null$0`, `Node$0`, ..., `Node$4` via relation `root`. No element relates (via any relation) to `Object$0`. While a vecSpec describes the state space, a *configuration* is a particular state. Configurations can be described by choosing, for each entry in the vecSpec, one of the possible values. In Fig. 3 we show a configuration vector as well as the binary tree described by the configuration.

We take the ordering in which Kodkod lists atom names as the strict linear ordering on atom names. We will denote this ordering on atom names by $<_K$. This ordering can be extended to a lexicographical ordering between configurations (denoted by $<_{LC}$) as follows (notice that all vector configurations have the same size, which we denote by s):

$$C <_{LC} C' \iff$$
$$(\exists i, 0 \leq i < s)(C[0, i-1] = C'[0, i-1] \wedge C[i] <_K C'[i])\ .$$

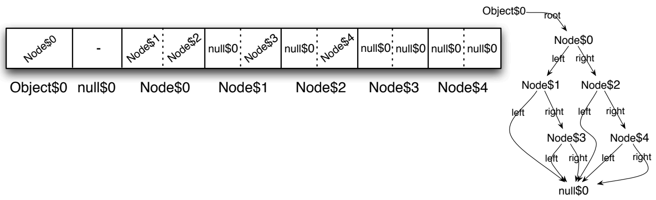

Fig. 3. Sample configuration for the model in Fig. 1.

THEOREM 1. *Relation $<_{LC}$ is a strict total order on the set of configurations.*

Proof. We must prove that $<_{LC}$ is irreflexive, transitive and total. Irreflexivity and totality follow from the irreflexivity and totality of $<_K$, respectively. Let us focus, then, on transitivity. Let $C_1 <_{LC} C_2$ and $C_2 <_{LC} C_3$. Let i_0, i_1 be the values such that $C_1[0, i_0 - 1] = C_2[0, i_0 - 1] \ \wedge \ C_1[i_0] <_K C_2[i_0]$ and $C_2[0, i_1 - 1] = C_3[0, i_1 - 1] \ \wedge \ C_2[i_1] <_K C_3[i_1]$. Let $I = min(i_0, i_1)$. Notice that $C_1[0, I - 1] = C_3[0, I - 1]$. If $I = i_0 < i_1$, $C_1[I] = C_1[i_0] <_K C_2[i_0] = C_3[i_0]$. If $I = i_1 < i_0$, $C_1[I] = C_1[i_1] = C_2[i_1] <_K C_3[i_1]$. If $I = i_0 = i_1$, then $C_1[I] = C_1[i_0] <_K C_2[i_0] = C_2[i_1] <_K C_3[i_1]$. Thus, since $<_K$ is transitive, $C_1[I] <_K C_3[I]$, which implies $C_1 <_{LC} C_3$. □

Theorem 1 allows us to adopt $<_{LC}$ as the strict linear ordering on configurations.

B. Selection of the Partition Points

Partition points are configurations that serve as boundaries between ranges. In this section we show how, given a range $R = [C_1, C_2]$ (with $C_1 <_{LC} C_2$) and a positive number n, we can select configurations $X_1, \ldots, X_{n-1} \in R$ such that ranges $[C_1, X_1]$, $[nextConf(X_1), X_2]$, $\ldots$, $[nextConf(X_{n-1}), C_2]$, where $nextConf(X_i)$ is the next configuration after X_i with respect to $<_{LC}$, are all contained in $[C_1, C_2]$ and are balanced with respect to the number of configurations they contain. Alg. 1 presents the pseudocode for partitioning a range into two subranges. It consists of finding an appropriate "mid point" between the two end points of the range. Lines 1–15 describe the most frequent case, where the first position in which the vectors corresponding to the end points of the range disagree contains elements that are far apart enough that a middle element can readily be found. The scenario, as well as the result returned by Alg. 1, are illustrated in Fig. 4.

Notice that once a partition point has been found, the source range $[C_1, C_2]$ is split into the ranges $[C_1, C]$ and $[nextConf(C), C_2]$. Pseudocode for method $nextConf$, which retrieves the next configuration according to ordering $<_{LC}$, is presented in Alg. 2. Intuitively, the behavior of Alg. 2 is quite similar to adding 1 in elementary arithmetic – yet, instead of using digits 0 through 9 (or those of any other fixed, uniform base), the algorithm uses, for each cell of a vector, the sorted list of options available for that cell according to the vecSpec. The reader should keep in mind that the options for each cell can be an arbitrary subset of the set of all atoms.

```
1  Config binRangePartition(lv, rv : Config, vs : vecSpec)
2      i = 0;                                    // Skip common prefix
3      while lv[i] == rv[i] do
4          i = i + 1;
5      end
6      fdl, fdr = lv[i], rv[i];        // First values that differ
7      options = [x ∈ vs[i] : fdl <_K x <_K fdr];  // Sorted list
8      if options != ∅ then
9          midOption = options[(len(options) − 1)/2];
10         output[0, i − 1] = lv[0, i − 1];
11         output[i] = midOption;
12         for i < j < len(vs) do
13             output[j] = max(vs[j]);
14         end
15         return output;
16     else
           // Values at pos i differ by 1
           // e.g., lv=[3,8,5,...], rv=[3,8,6,...]
17         if i = len(vs) then
18             return lv;
19         end
20         i = i + 1;
21         while lv[i] == max(vs[i]) && rv[i] == min(vs[i]) do
22             i = i + 1;
23             if i == len(vs) then
                   // e.g. lv=[3,8,5,9,9,9]
                   //  and rv=[3,8,6,0,0,0]
24                 return lv;
25             end
26         end
27         if lv[i] != max(vs[i]) then
               // e.g. lv=[3,8,5,9,9,7,...]
               //  and rv=[3,8,6,0,0,x,...]
28             output[0, i − 1] = lv[0, i − 1];
29             for i ≤ j < len(vs) do
30                 output[j] = max(vs[j]);
31             end
32             return output;
33         else
               // e.g. lv=[3,8,5,9,9,9,...]
               //  and rv=[3,8,6,0,0,3,...]
34             output[0, i − 1] = rv[0, i − 1];
35             for i ≤ j < len(vs) do
36                 output[j] = min(vs[j]);
37             end
38             return output;
39         end
40     end
41 end
```

Algorithm 1: The binary partitioning algorithm.

```
Config nextConf(c : Config, vs : vecSpec)
    // c must not be the last config.
    i = len(vs) − 1;
    output = c;
    while true do
        options = [x ∈ vs[i]];                    // Sorted list
        pos = position of options[i] in options;
        if pos < len(options) − 1 then
            output[i] = options[pos + 1];
            return output;
        else
            output[i] = options[0];
            i = i − 1;
        end
    end
end
```

Algorithm 2: Pseudocode for method *nextConf*.

Let us now focus on the generation of range partitions in the general case. Algorithm 3 presents the pseudocode for the actual partition algorithm. The algorithm starts from an input

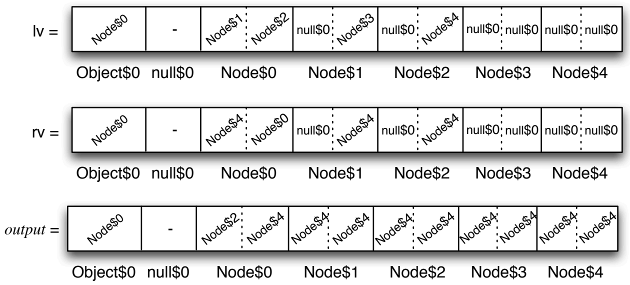

Fig. 4. Application of partitioning (standard case).

```
List[Range] rangePartitioning(R : Range, n : int, vs: vecSpec)
  L = addToEmptyList(R);                          // L = [R]
  i = 0;              // index for traversing the list
  while len(L) < n && hasSplittableRanges(L) do
    if splittable(L[i]) then
      Config C1 = leftEndpoint(L[i]);
      Config C2 = rightEndpoint(L[i]);
      Config C3 = binRangePartition(C1, C2, vs);
      removeRangeAtPos(L, i);
      addRangeAtPos(L, i, [C1, C3]);
      addRangeAtPos(L, i + 1, [nextConf(C3, vs), C2]);
      if i < len(L) − 2 then
        i = i + 2;
      else
        i = 0;
      end
    else
      if i < len(L) − 1 then
        i = i + 1;
      else
        i = 0;
      end
    end
  end
end
```

Algorithm 3: The range partitioning algorithm.

range, and iterates over a list of already generated ranges as long as the requested number of ranges has not been reached and there are still some ranges left to be split (ranges of the form $[C, C]$ cannot be split any further).

C. Parallel Range Analysis

In Section III-B we proposed a method to partition a range into sub-ranges; we now explain how to use the aforementioned in order to parallelize the analysis of an Alloy model.

Our scenario for parallel analysis makes use of a cluster of computers. Worker processes execute commands sent by a master process, which runs on a dedicated processor. Actions that workers can perform include: solving a task sequentially, aborting the ongoing analysis, splitting the current task into a given number of ranges (and locally enqueueing the resulting subtasks), fetching a new task from the local queue, and moving tasks between local and remote queues (requesting and obtaining tasks from other workers). Further details on Ranger's implementation will be provided in Section IV.

In the remaining parts of this section we present two alternatives for parallelization, both based on range partitioning. Before doing so, we discuss the impact of Alloy's symmetry-breaking predicates on the ranges generated by partitioning.

1) Range Partitioning and Symmetry Breaking: Atom names are irrelevant when building an Alloy configuration: given a configuration that satisfies (or not) the facts of an Alloy model, any other configuration obtained by mere permutation of atom names (while, of course, preserving typing constraints) will behave in the same way. Symmetry-breaking axioms are introduced by Kodkod [37] during the translation of the Alloy model to a propositional formula, and greatly improve the performance of the underlying SAT-solver by avoiding the exploration of many such superfluous isomorphisms. Therefore, symmetry breaking has a direct impact on what non-superfluous configurations will look like. For example, for the model in Fig. 1, field `root` can only relate to atom names `null$0` or `Node$4`. Hence, any ranges in which `root` points to nodes `Node$0`, `Node$1`, `Node$2` or `Node$3` contain configurations that cannot satisfy the propositional formula generated by Kodkod. For our sample Alloy model from Fig. 1 (but using a scope of 10 `Node`, since 5 `Node` is too easy), Table II shows how partitioning the full range into increasingly large numbers of ranges yields very small numbers of nontrivial subproblems. The remaining ranges can all be proved unsatisfiable in under a millisecond each. As we will see in Sections III-C2 and III-C3, the fact that a significant number of the tasks resulting from a partition may become trivial can lead to hardware being severely underused. In order to measure the actual utilization of the assigned hardware during parallel analysis, we define the metric

$$\text{H}\mathit{ardware}\ \text{U}\mathit{se}\ \text{E}\mathit{fficiency} = \frac{\text{Total non-idle seconds}}{\text{Number of workers} \times t},$$

where t is the wallclock runtime in seconds taken by the whole parallel analysis (as perceived by the end user), and the numerator is the sum, over all workers, of the number of seconds during which some task was actually being analyzed.

TABLE II
RANGE PARTITIONING: NUMBER AND PERCENTAGE OF NONTRIVIAL SUBPROBLEMS (OF THOSE GENERATED FOR BINTREES WITH 10 NODE).

Num. Ranges	512	1024	2048	4096	8192	16384
Nontrivial	10	20	35	56	66	90
Nontrivial %	0.019	0.019	0.017	0.013	0.008	0.005

2) Flat Range Partitioning: One way of parallelizing the analysis based on range partitioning consists in determining a large enough value of n (called the *fan-out* of the analysis), and having a worker partition the full range into n ranges using algorithm *rangePartitioning*. The n newly generated tasks are then solved in parallel by the available workers. Each worker receives a task and solves it sequentially until a SAT/UNSAT verdict is obtained. Unfortunately, this approach seldom performs as well as expected. In most cases the solving time variance (hardest vs. easiest subproblems obtained after initial partitioning) is very high. Thus, if the initial range is only partitioned once, eventually only a few active workers will remain, while (almost all) other workers will be idle until the end of the run. Table III reports the HUE for the analysis of the model in Fig. 1 when flat range partitioning is used.

TABLE III
EFFICIENCY RATES OBTAINED FOR FLAT RANGE PARTITIONING OF BINTREE EXAMPLE, WITH 10 NODE, USING 64 WORKERS.

Num. Ranges	512	1024	2048	4096	8192	16384
HUE		0.04	0.04	0.11	0.11	0.12

3) Recursive Range Partitioning: The main drawback of flat range partitioning is its static nature. Trying to balance the number of candidate configurations in each range is indeed a reasonable starting point, but we cannot predict, in the general case, where the harder subranges might lie. A more dynamic approach to determining the location of nontrivial subranges is therefore desirable. In recursive range partitioning, the oldest active subproblem (i.e., the oldest among those that are still being SAT-solved) can be re-partitioned by its assigned worker. This yields sub-subproblems, and so on, recursively. Recursive partitioning of a range may occur under two circumstances:

- the UNSAT frequency, i.e., the number of UNSAT verdicts per second, falls below a user-defined threshold, or
- there are idle workers.

The first condition aims at achieving progress during analysis by avoiding wasting time analyzing tasks that are still too hard to be solved sequentially by a worker. The second condition targets the HUE metric and strives to make good use of resources by avoiding idle workers.

Unlike flat partitioning, where the fan-out must be large and fixed beforehand, in recursive partitioning we use a small fan-out; its value is set to the number of workers. For instance, in the experiments reported in Section V, the fan-out is 64, since that is the total number of worker cores in the cluster. In this way, whenever many of the 64 tasks turn out to be trivial (or shortly after idle workers start to abound) the recursive partitioning process will react by "zooming in" (i.e., focusing the computational effort) on the remaining nontrivial tasks.

Using recursive range partitioning, the HUE value for our example (for the same 10-Node scope) becomes 0.84, which is about 8 times higher than with flat range partitioning. Even if we only count the time invested in *successful* solving attempts as non-idle time (i.e., if partial solving attempts before re-splitting were to be considered completely wasted effort), the HUE value for this run would be 0.59 – still a significant improvement over the low efficiency of flat partitioning.

The following theorem discusses the correctness of recursive range partitioning. A detailed proof is omitted due to space constraints.

THEOREM 2. *Recursive range partitioning is sound and complete, i.e., an Alloy model has a satisfying configuration if and only if one can be found using recursive range partitioning.*

Proof sketch: Recursive range partitioning splits ranges whenever tasks are aborted. Proving the theorem then requires showing that each time a range is partitioned according to Alg. 3, no configurations are lost. Algorithm 3 iteratively splits a list of ranges using the binary split implemented in Alg. 1. Then, given a range $[C_1, C_2]$ visited by Alg. 3, it suffices to show, according to program lines 10 and 11, that $[C_1, C_2] = [C_1, C_3] \cup [\mathit{nextConf}(C_3, \mathrm{vs}), C_2]$ (where vs is the global vecSpec and C_3 is the configuration returned by Alg. 1). We now need to prove that $C_3 \in [C_1, C_2]$ and that $\mathit{nextConf}(C_3, \mathrm{vs})$ indeed returns the next configuration. The first proof is completed by considering the alternatives provided by the guards in the algorithm. For instance, if the set $\mathit{options}$ is nonempty, the proof is immediate (see lines 9–15). Regarding the second property, we must prove that Alg. 2 terminates, and that when it terminates it produces the next configuration. Termination is guaranteed because index i iterates from the back of the array until it finds a position in the configuration in which the stored atom name is not the maximum possible. Such a position must exist because the input configuration is required not to be the largest possible one. Proving that the configuration produced by Alg. 2 is the next one according to ordering $<_{LC}$ reduces to showing that if a configuration C exists such that $C_3 <_{LC} C$ and $C \leq_{LC} \mathit{nextConf}(C_3, \mathrm{vs})$, then $C = \mathit{nextConf}(C_3, \mathrm{vs})$.

IV. IMPLEMENTATION DETAILS

A. Initial Model Translation and VecSpec Construction

Given a user-provided Alloy model, as a first step, Ranger interfaces with the Alloy Analyzer to obtain a list of suitable fields for range partitioning (currently, all functional binary relations are used; see Section V-E) and to have the model translated to CNF. During the translation, it also interacts with Kodkod in order to obtain the necessary information to build the vecSpec: a copy of the atom universe, details on which atoms appear in each relevant relation's domain and range, and on which propositional variable is being used to represent presence or absence of each tuple in each relevant relation.

The model is only translated once. The resulting CNF file is broadcast by the master to all workers, along with a description of the vecSpec, just before the distributed analysis phase starts. All further range-related restrictions are to be injected by the workers, directly at the clausal level, every time they load a new task into their local sequential solver. This eliminates the cost of re-running the Alloy translation toolchain, and allows for subproblems to be very lightweight objects: each pending task is represented by a pair of vectors (which require less than a few hundred bytes each, even for the largest scopes and models in our benchmark).

B. Clauses Added to Enforce SAT-Solving Within Range

In Alg. 4 we show pseudocode illustrating what each worker does when loading a new subproblem. Three groups of clauses are injected. The first group limits the search to candidate configurations that are no smaller (as per $<_{LC}$) than the left endpoint of the range, whereas the second group requires that they be no larger than the right endpoint. For the third group, both vectors are scanned from left to right until the end of their common prefix (if any) is found. A unit clause is added for every cell that could only have one possible value (i.e., within the common prefix). At the first cell where left and right values differ, an all-positive clause is generated.

The third group does not really add any new constraints: its clauses could also be derived (with some effort) from the first two groups and the rest of the translated model. However, empirical evidence suggests that the presence of this positive formulation often promotes faster propagation.

Let l be the vector length, and c_i (with $0 \leq i < l$) the number of choices for the i-th cell according to a vecSpec. Then $\sum_i (c_i - 1)$ is a worst-case upper bound for the number of clauses added by either of the first two groups; as for the third group, it cannot add more than l clauses. In practice, we have not seen any case where the total number of added clauses reached 1% of the number of clauses in the CNF.

```
addRangeClauses(ss: SATSolver, R : Range, vs: vecSpec)
    // Forbid anything smaller than left endpoint
    Config lvec = leftEndpoint(R);
    antecedent = [];                                // Empty list
    for ith, atom in enumerate(lvec) do
        options = vs.atoms[ith];                    // Sorted list
        position = options.indexOf(atom);
        forbidden = options[0, position − 1];
        for f_ith, f_atom in enumerate(forbidden) do
            f_pvar = vs.pvars[ith][f_ith];
            ss.addClause(antecedent ++ [−f_pvar]);
        end
        atom_pvar = vs.pvars[ith][position];
        antecedent.append(−atom_pvar);
    end
    // Forbid anything greater than right endpoint
    Config rvec = rightEndpoint(R);
    antecedent = [];                                // Empty list
    for ith, atom in enumerate(lvec) do
        options = vs.atoms[ith];                    // Sorted list
        position = options.indexOf(atom);
        forbidden = options[position + 1, len(options) − 1];
        for f_ith, f_atom in enumerate(forbidden) do
            f_pvar = vs.pvars[ith][f_ith + position + 1];
            ss.addClause(antecedent ++ [−f_pvar]);
        end
        atom_pvar = vs.pvars[ith][position];
        antecedent.append(−atom_pvar);
    end
    // Add a unit clause per common prefix cell
    // and a clause for the first differing cell
    atompairs = zip(lvec, rvec);
    for ith, (latom, ratom) in enumerate(atompairs) do
        options = vs.atoms[ith];
        lpos = options.indexOf(latom);
        rpos = options.indexOf(ratom);
        if latom == ratom then
            // still within common prefix
            ss.addClause([ vs.pvars[ith][lpos] ]);
        else
            // first cell where values differ
            ss.addClause(vs.pvars[ith][lpos, rpos]);
            break;
        end
    end
end
```

Algorithm 4: Adding clauses to enforce ranged analysis.

V. Experimental Results

In this section we first describe the hardware and software setup (V-A). We then evaluate Ranger on a benchmark of models that includes valid (V-B) and invalid (V-C) assertions. In V-D we evaluate how the speedup achieved by Ranger evolves as the amount of hardware used for the analysis varies. Finally, in V-E we discuss some possible threats to the validity of the presented experimental results.

A. *Setup and Conventions*

Ranger is a distributed application based on the MPI standard. Each of its worker threads runs the Minisat [11] solver, version 2.2.0. All experiments were run in a cluster of 16 commodity PCs, each featuring an Intel Core i7-2600 4-core, 8-thread processor with a 3.40 GHz clock speed and 8 GB DDR3 RAM, running Linux 3.2.0. All Ranger experiments were run on 8x8 (8 nodes, each running 8 worker threads) except where otherwise indicated. Each experiment was run 3 times; the reported timing is the average thereof.

All times are given in wallclock seconds. “TO” (timeout) means failure to complete within 36,000 seconds (10 hours) except where otherwise indicated. “OofM” (out of memory) means failure to complete due to exhausting 8 GB of main memory. “AA” (Alloy Analyzer) means that the same sequential SAT-solver used by Ranger (Minisat 2.2.0) was run on the unmodified CNF translation of the source Alloy model.

B. *UNSAT Cases (Valid Properties)*

BinaryTrees is the model that we introduced as a running example in Section II. Property TwoDefsEquivalent asserts the equivalence of two different characterizations of the binary tree structure. As shown in Table IV, its difficulty curve is particularly steep: although the property can be proven for scope 9 in under a minute, scope 10 requires over 7 hours. Ranger can prove the latter in under 4 minutes – a 119x speedup. It can also prove the property for scope 11 in under an hour, whereas the Alloy Analyzer fails to prove it within the 10-hour timeout. Note that in this case the speedup is conservatively reported as being “>10.83”, since we do not know how much longer than 10 hours the Analyzer would need. The actual speedup is likely to be much higher.

LinkedLists is a model involving singly linked lists. In this case the goal is to verify that 3 different definitions are equivalent. The model includes two separate properties to that effect: Pairwise, which asserts that $D_1 \Leftrightarrow D_2 \wedge D_2 \Leftrightarrow D_3$, and Circular (i.e., $D_1 \Rightarrow D_2 \wedge D_2 \Rightarrow D_3 \wedge D_3 \Rightarrow D_1$). Tables V and VI show similar behaviors for both properties, with Ranger obtaining about 14x speedup on the largest scope that the Alloy Analyzer can handle (within 10 hours), and then being able to prove the properties for 2 additional scopes.

Chord is a model of the Chord [36] distributed hash table lookup protocol. It is one of the case studies bundled with the Alloy distribution. The model contains one property, called FindSuccessorWorks, that is particularly hard to prove. Table VII shows a speedup of at least 103x (again, merely a floor value) for the first scope that is not tractable sequentially. It also shows that distributed analysis pushes the tractability barrier another 2 scopes for this property.

STABLEMUTEXRING, another Alloy-bundled example, is a model of Dijkstra's K-state mutual exclusion algorithm for a ring [9]. There are two hard-to-prove properties in this model. Both use the notion of a "bad tick" – an instant in time where two or more distinct processes try to run their critical sections simultaneously. NOBADSAFETYTRACE asserts that it is impossible to find a trace with a loop containing a bad tick (such that the algorithm would never stabilize). CLOSURE asserts that there can be no bad ticks if the first tick is "good". As seen in Tables VIII and IX, Ranger pushes the ten-hour tractability limit 10 scopes (from 12 to 22) for the former, and 4 scopes (from 13 to 17) for the latter. At the last AA-tractable scopes (12 and 13, respectively), the speedups exceed 200x for NOBADSAFETYTRACE and 40x for CLOSURE.

FIREWIRE describes the behavior of the leader election protocol used in the IEEE 1394 [17] standard for connecting consumer electronic devices. This is another case study included with the Alloy distribution. The hardest property in the model is ATMOSTONEELECTED, which asserts that two or more devices cannot be elected as leader in the same state. As shown in Table X, distributed analysis yields nearly 20x speedup for scope 5. For scope 6, where the Alloy Analyzer fails to yield a result within 10 hours, Ranger proves the property in under 4 hours.

C. SAT Cases (Invalid Properties / Instance Generation)

Many Alloy-borne SAT cases are easy; typically, when the translation of an Alloy formula results in a satisfiable CNF, finding a satisfying valuation is a quick and simple matter. However, hard SAT problems do come up in practice, and can be very challenging indeed. Therefore, we also evaluate Ranger on some difficult SAT instances.

BINOMIALHEAP is the translation to Alloy of a Java binomial heap class implementation, taken from [39]. One of its methods, `extractMin()`, contains a bug that can only be detected for some sufficiently large input structures. Property EXTRACTMINCORRECT asserts the correctness of said method. Its translation to CNF yields UNSAT problems up to scope 12, but nontrivial SAT problems for scopes 13 and above. Although the speedups obtained were modest (around 3x, on average), it was important for us to confirm that Ranger did not miss the counterexample whenever one existed.

AVLTREES was originally written for automated test input generation. The goal, for scope n, is to find some configuration that represents a valid AVL tree of size n. An easy task for small n, this becomes much harder as n grows. Table XII shows that it took the Analyzer over one hour to produce an AVL tree of size 19, while Ranger achieved the same in 138 seconds – a 27x speedup. At scope 22, sequential analysis exhausted 8 GB of memory, whereas distributed analysis succeeded in producing AVLs of sizes 22 and 23.

The FIREWIRE model also includes NOREPEATS, an instance generation command. This is an auxiliary property: the author of the model suggests running it repeatedly, increasing the number of states, until no counterexample is found, to determine how many states suffice for a certain scope. For

TABLE IV
BINARYTREES: TWODEFSEQUIVALENT

Scope	AA	Ranger	Speedup
8	6.00	5.49	1.09
9	43.69	16.58	2.63
10	25,552.44	215.22	118.72
11	TO	3,324.20	$>$ 10.83
12	TO	TO	

TABLE V
LINKEDLISTS: THREEDEFSEQUIVALENT (PAIRWISE)

Scope	AA	Ranger	Speedup
13	24.25	14.56	1.67
14	86.15	37.57	2.29
15	346.48	91.09	3.80
16	1,862.96	197.98	9.41
17	11,580.27	819.81	14.13
18	TO	4,107.56	$>$ 8.76
19	TO	22,845.55	$\gg$ 1.58
20	TO	TO	

TABLE VI
LINKEDLISTS: THREEDEFSEQUIVALENT (CIRCULAR)

Scope	AA	Ranger	Speedup
13	22.17	14.73	1.50
14	74.94	35.61	2.10
15	360.01	85.65	4.20
16	1,602.04	189.79	8.44
17	11,484.27	859.62	13.36
18	TO	4,299.79	$>$ 8.37
19	TO	24,077.44	$\gg$ 1.50
20	TO	TO	

TABLE VII
CHORD: FINDSUCCESSORWORKS

Scope	AA	Ranger	Speedup
6	94.90	23.95	3.96
7	1,447.98	67.86	21.34
8	TO	349.76	$>$ 102.93
9	TO	3,569.07	$\gg$ 10.09
10	TO	TO	

TABLE VIII
STABLEMUTEXRING: NOBADSAFETYTRACE

Scope	AA	Ranger	Speedup
10	322.39	35.79	9.01
11	1,326.09	51.10	25.95
12	24,239.91	118.04	205.35
13	TO	330.74	$>$ 108.85
14	TO	850.46	$\gg$ 42.33
15	TO	1,672.21	$\ggg$ 21.53
16	TO	3,802.20	$\gg\gg$ 9.47
17	TO	5,263.09	$\ggg\gg$ 6.84
18	TO	7,400.67	$\ggg\ggg$ 4.86
19	TO	10,859.77	$\ggg\gg\gg$ 3.31
20	TO	16,404.06	$\gg\gg\gg\gg$ 2.19
21	TO	23,982.52	$\ggg\ggg\ggg$ 1.50
22	TO	29,705.61	$\gg\gg\gg\gg\gg$ 1.21
23	TO	TO	

TABLE IX
STABLEMUTEXRING: CLOSURE

Scope	AA	Ranger	Speedup
10	343.50	33.03	10.40
11	924.96	57.08	16.20
12	2,835.47	111.72	25.38
13	9,459.15	231.50	40.86
14	TO	707.88	$>$ 50.86
15	TO	2,427.98	$\gg$ 14.83
16	TO	9,771.50	$\ggg$ 3.68
17	TO	30,607.91	$\gg\gg$ 1.18
18	TO	TO	

TABLE X
FIREWIRE: ATMOSTONEELECTED

Scope	AA	Ranger	Speedup
3	3.98	3.16	1.26
4	141.04	23.67	5.96
5	6,269.58	319.87	19.60
6	TO	14,297.74	$>$ 2.52
7	TO	TO	

large scopes, finding such intermediate SAT instances becomes a hard problem in its own right. We ran these analyses sequentially for up to 16 states per scope, and for each scope, re-ran the most demanding analysis using Ranger. As shown in Table XIII, the distributed approach yielded over 40x speedup for the last sequentially-tractable scope (22), and was able to raise the tractability limit from 22 to 34.

TABLE XI
BINOMIALHEAP: EXTRACTMINCORRECT

Scope	AA	Ranger	Speedup
8	102.10	40.56	2.52
9	185.05	106.64	1.74
10	243.12	132.81	1.83
11	563.47	196.93	2.86
12	700.69	239.04	2.93
13	80.41	31.20	2.58
14	122.87	60.84	2.02
15	251.49	80.29	3.13
16	349.60	187.09	1.87
17	847.28	270.93	3.13
18	483.16	116.55	4.15
19	542.40	381.70	1.42
20	1,022.42	149.97	6.82

TABLE XII
AVLTREES: GENERATEINSTANCE

Scope	AA	Ranger	Speedup
15	47.09	13.87	3.40
16	121.76	51.12	2.38
17	195.37	81.05	2.41
18	1,703.20	125.95	13.52
19	3,715.74	137.69	26.99
20	3,839.97	241.02	15.93
21	17,588.57	1,422.13	12.37
22	OofM	4,993.58	∞
23	OofM	13,654.49	∞
24	OofM	TO	

TABLE XIII
FIREWIRE: NOREPEATS

Scope	AA	Ranger	Speedup
16	148.97	17.01	8.76
18	334.40	28.48	11.74
20	497.93	41.12	12.11
22	765.93	18.21	42.06
24	OofM	46.20	∞
26	OofM	63.52	∞
28	OofM	100.62	∞
30	OofM	89.99	∞
32	OofM	95.79	∞
34	OofM	171.11	∞
36	OofM	OofM	

D. Adding More Hardware

For each of the aforementioned series, we took the hardest scope that was tractable by Ranger on 8x8 and re-ran it using half as much hardware, 50% more hardware, and twice as much hardware (i.e., on 4x8, 12x8, and the full 16x8 capacity of the cluster). The results are reported in Table XIV. In all UNSAT cases, the actual runtimes were close to the linear extrapolation (200%, 100%, 66%, 50%) of the 8x8 timing.

SAT cases are less predictable since, rather than exhausting the search space, success depends on quickly finding the first needle in the haystack. While AVL instance generation scaled even better than expected, the other two cases did not do as well, and SAT runs on 4x8 performed poorly in general.

TABLE XIV
ADDING MORE HARDWARE

Model/Property	Scope	4x8	8x8	12x8	16x8
LinkedLists: Equiv. Pairwise	19	TO	22,846	14,446	10,197
			100%	63%	45%
LinkedLists: Equiv. Circular	19	TO	24,077	14,368	10,059
			100%	60%	42%
BinTrees: Equivalence	11	6,584	3,324	2,273	1,688
		198%	100%	68%	51%
Chord: FindSuccWorks	9	7,041	3,569	2,270	1,807
		197%	100%	64%	51%
SMRing: Closure	17	TO	30,608	18,806	12,848
			100%	61%	42%
SMRing: BadSafetyTrace	22	TO	29,706	20,526	17,354
			100%	69%	58%
FireWire: AtMostOneElected	6	27,680	14,298	8,843	6,707
		194%	100%	62%	47%
BHeap: ExtractMinCorrect	19	1,989	382	322	272
		521%	100%	84%	71%
AVL: instance generation	19	TO	13,654	5,516	4,958
			100%	40%	36%
Firewire: NoRepeats	34	19,949	171	177	195
		11,659%	100%	104%	114%

E. Threats to Validity

When building a vecSpec from an Alloy model, the current Ranger implementation considers functional binary relations. Note that this does not restrict input models to those that only use such relations. Ranger can analyze Alloy models with relations of arbitrary type and arity; it simply ignores nonfunctional and/or ternary relations for the purposes of building the vecSpec, and therefore, for those of range partitioning. So far, this does not seem to be an impediment: most of the models in the benchmark use nonfunctional and/or ternary relations; some even use many of them, and comparatively few functional binary ones. However, this does imply that if a model uses no functional binary relations at all, it would not be splittable by the current version of Ranger, as the vecSpec would be empty. A binary nonfunctional relation $R \subseteq A \times B$ can be seen as a ternary relation $T_R \subseteq A \times B \times \{true, false\}$, where $(a, b) \in R \Leftrightarrow (a, b, true) \in T_R$. Therefore, the problem reduces to handling ternary relations. We can see a ternary relation $T \subseteq A \times B \times C$ as a total function $F_T : A \to \mathcal{P}(B \times C)$, where $F_T(a) = \{(b, c) \in B \times C \mid (a, b, c) \in T\}$. Note that such functions would introduce a large number of options in the vecSpec, but that each split reduces such options by half.

VI. RELATED WORK

As stated in Section I, scaling Alloy analysis to larger scopes is necessary to improve the confidence levels attainable by users when analyzing models. An important step in this direction is the inclusion of symmetry-breaking predicates during the translation to propositional logic, significantly enhancing analysis capabilities [30], [37]. Surprisingly, developments on parallel and/or distributed analysis of Alloy models are scarce.

The first option to consider is using a parallel SAT-solver. Multi-core SAT-solver research has gained a lot of momentum. ManySAT [16] and plingeling [4] are award-winning multithreaded SAT solvers. As shown on Table XV, Ranger frequently outperforms both of them even when running on a single machine (1x8), possibly due to the synergy between

TABLE XV
RANGER ON 1X8 VS. MULTITHREADED SAT-SOLVERS. (TO=30 MIN)

Model/Property	Scope	Ranger (1x8)	plingeling v578f (1x8)	ManySAT v2.0 (1x8)
LinkedLists: Equiv. Pairwise	15	321.70	657.90	545.53
	16	1,169.76	TO	TO
	17	TO	TO	TO
BinTrees: Equivalence	9	23.20	97.80	15.57
	10	1,467.65	TO	TO
	11	TO	TO	TO
Chord: FindSuccWorks	6	33.87	173.80	51.34
	7	269.88	1,514.10	606.86
	8	TO	TO	TO
SMRing: Closure	11	221.37	837.50	361.60
	12	639.62	1,657.60	1,039.99
	13	TO	TO	TO
SMRing: BadSafetyTrace	11	180.05	TO	426.72
	12	695.54	TO	TO
	13	TO	TO	TO
FireWire: AtMostOneElected	4	51.85	14.10	35.02
	5	TO	71.80	1,529.32
BHeap: ExtractMinCorrect	14	65.15	435.50	12.89
	15	445.18	506.70	139.19
	16	TO	501.40	65.22
AVL: instance generation	15	20.07	8.10	15.95
	16	175.42	12.90	31.65
	17	TO	18.00	78.93
Firewire: NoRepeats	22	19.76	178.10	201.56
	24	741.24	293.70	199.97
	26	TO	502.50	450.80

range partitioning and Alloy's symmetry breaking. But another important advantage of Ranger is its distributed nature, which makes it possible to add more machines and combine their computational power. Multithreaded solvers heavily depend on shared memory and are thus confined to a single computer.

Usable distributed SAT-solvers are hard to come by. PMSat [15], a cluster-oriented version of Minisat, is available but reports generally small speed-ups. GrADSAT [7] 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 [27] is a SAT-solver for clusters. It reports linear speed-ups, but the tool is not available for experimentation. Also, relying on a parallel SAT-solver prevents making use of Alloy-level information that may contribute to better analyses.

In [29], the notion of transcoping is introduced as an aid to improve parallel analysis of Alloy models. Since Alloy analyses occur within given bounds, transcoping proposes to explore small scopes first in order to extrapolate the best way to distribute the analysis of larger scopes. Ranger may contribute to the development of transcoping, given that it introduces a new technique for distributing the analysis.

Although little research has been done on parallelizing its analysis, Alloy has been used as an intermediate language by different tools that parallelize code analysis. In [34], parallel analysis of code is performed by splitting the program control flow graph and using JForge [8] (which relies on Kodkod) to analyze each slice. Note that, as in [33], parallelization occurs at the code level, not at the intermediate Alloy representation level. In [28], parallel analysis of Java code is performed by translating complete methods to Alloy. The partitions needed to parallelize the analysis are obtained from the intermediate Alloy representation. Unfortunately, the efficiency of the technique depends on the presence of class invariants or the lack of aliasing, concepts usually absent in more general Alloy models such as the ones considered in this article.

The vector-based representation of Alloy configurations is adopted from the *candidate vectors* of the Korat tool [5] that performs a backtracking search for test generation using imperative predicates. Two techniques implement Korat in parallel – one technique [31] uses executions of the imperative predicate to distribute the search during backtracking by creating work items for parallel workers and the other technique [24] fast-forwards the search to create ranges for parallel exploration without work stealing. The problems addressed by Korat and Ranger (testing of imperative code and analysis of Alloy models, respectively) as well as the respective partition techniques are quite different.

Ranging techniques for symbolic execution [32] and explicit state model checking [12] of imperative programs were introduced recently in the context of the KLEE symbolic execution tool for C [6] and the JPF model checker for Java [38], respectively. Ranging to analyze declarative models in Alloy is very different from ranging to analyze imperative programs in C or Java. More precisely, ranges in symbolic execution and model checking are based on program execution paths, specifically sequences of control-flow branches. Such paths do not exist in declarative models. Our technique for ranging for Alloy defines a novel form of ranges – at the black-box input space level, not white-box control-flow level.

VII. CONCLUSIONS AND FURTHER WORK

This paper introduced a novel technique for scaling Alloy's SAT-based analysis using ranging. Experiments using a variety of hard-to-solve Alloy formulas showed that the technique is very effective, especially for valid assertions, where the search space needs to be exhausted. When dealing with difficult invalid assertions, except for some situations with particularly low quantities of available hardware, counterexamples were always found in a timely fashion.

Our work opens a new direction in scaling the analysis of declarative models. With the increasing availability of multi-core and multi-processor systems, such parallel techniques have a vital role to play in substantially enhacing our ability to develop more reliable software. Our next step is to update the implementation as discussed in Section V-E, so that models can be range-partitioned on a wider class of relations. Also, the fact that superlinear speedups were obtained implies that some of the gain cannot stem from parallelism, but rather from the partitioning itself – in some cases, merely splitting a problem and solving the resulting subproblems *sequentially* would have yielded some speedup. This surprising phenomenon deserves further analysis. We also plan to explore the application of ranging to other declarative domains, such as SMT solving as well as deep static checking where the program and its specification are represented together using a formula, which captures a violation of the specification by the program for goal-directed counterexample generation.

REFERENCES

[1] Abad P., Aguirre N., Bengolea V., Ciolek D., Frias M.F., Galeotti J., Maibaum T., Moscato M., Rosner N., Vissani I., *Tight Bounds + Incremental SAT = Better Test Generation under Rich Contracts*, in Proceedings of Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST) 2013.

[2] Abrial J. R.,. *The B-Book: Assigning Programs to Meanings*. Cambridge, UK, Cambridge University Press, 1996.

[3] Alloy Analyzer, available at http://alloy.mit.edu/alloy/download.html.

[4] Biere A., *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*, in Solver description, Special Track 1 (Parallel CNF), SAT-Race 2010, available at http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_1+2+3+6.pdf.

[5] Boyapati C., Khurshid S., Marinov D., *Korat: automated testing based on Java predicates.* ISSTA 2002: 123-133.

[6] Cadar C., Dunbar D., and Engler D. R. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. In Proc. 8^{th} Symposium on Operating Systems Design and Implementation (OSDI), pages 209–224, 2008.

[7] Chrabakh W., and Wolski R., *GrADSAT: A Parallel SAT Solver for the Grid*, in UCSB Computer Science Technical Report Number 2003-05.

[8] Dennis, G., Chang, F., Jackson, D., *Modular Verification of Code with SAT*. in ISSTA'06, pp. 109–120, 2006.

[9] Dijkstra E. W., *A belated proof of self-stabilization*, Distributed Computing, Vol. 1, Issue 1, pp.5–6, 1986.

[10] Dolby J., Vaziri M., Tip F., *Finding Bugs Efficiently with a SAT Solver*, in ESEC/FSE'07, pp. 195–204, ACM Press, 2007.

[11] Een N., and Sörensson N., *An Extensible SAT-Solver*. In SAT 2003.

[12] Funes D., Siddiqui J. H., and Khurshid S. *Ranged model checking*. In Proc. Java PathFinder Workshop (JPF), 2012.

[13] Galeotti J. P., Rosner N., López Pombo C., Frias M. F., *Analysis of invariants for efficient bounded verification*. In proceedings of ISSTA 2010, pp. 25-36, 2010.

[14] Galeotti J. P., Rosner N., López Pombo C., Frias M. F., *TACO: Efficient SAT-Based Bounded Verification Using Symmetry Breaking and Tight Bounds*. IEEE Transactions on Software Engineering, to appear.

[15] Gil L., Flores P., and Silveira L. M., *PMSat: a parallel version of MiniSAT*, Journal on Satisfiability, Boolean Modeling and Computation 6 (2008) 71-98.

[16] Hamadi Y., Jabbour S., and Sais L, *ManySAT: a Parallel SAT Solver*, International Journal on Satisfiability, Boolean Modeling and Computation (JSAT), Volume 6, Special Issue on Parallel SAT, IOS Press, 2009.

[17] *IEEE Standard for a High-Performance Serial Bus*, available ar http://ieeexplore.ieee.org/servlet/opac?punumber=4659231

[18] Jackson, D., *Software Abstractions*. MIT Press, 2006.

[19] Kang E., Jackson D., *Formal Modeling and Analysis of a Flash Filesystem in Alloy*. in Proceedings of ABZ 2008, LNCS 5238, Springer, 294–308.

[20] Kim J. S., and Garlan D., *Analyzing Architectural Styles*, Journal of Systems and Software, Vol. 83, Issue 7, Elsevier, 1216-1235.

[21] Leavens G.T.,, Baker A.L., and Ruby C. *JML: a notation for detailed design*. In Behavioral Specifications of Businesses and Systems, Chapter 12, pp. 175-188, Amsterdam, Kluwer, 1999.

[22] Leino K. R. M., Mülcer P., *Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs*, Manuscript KRML 189, 17 September 2009, Available at http://specsharp.codeplex.com/wikipage?title=Tutorial

[23] Marinov D., and Khurshid S. *TestEra: A Novel Framework for Automated Testing of Java Programs*. In *Proc. 16th IEEE Conference on Automated Software Engineering (ASE)*, 2001.

[24] Misailovic S., Milicevic A., Petrovic N., Khurshid S., and Marinov D. *Parallel Test Generation and Execution with Korat* In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2007.

[25] Maoz S., Ringert J.O., and Rumpe B.. *CD2Alloy: Class Diagrams Analysis Using Alloy Revisited*. In Proceedings of MODELS 2011, LNCS 6981, Springer, 592-607.

[26] Object Management Group. *OCL Specification V. 2.3.1*. January 1st, 2012. Available at http://www.omg.org/spec/OCL/2.3.1/PDF/.

[27] Ohmura K., and Ueda K., *c-sat: A Parallel SAT Solver for Clusters*, in SAT 2009, LNCS 5585, 2009.

[28] Rosner N., Galeotti J. P., Bermúdez S., Marucci Blas G., Perez De Rosso S., Pizzagalli L., Zemín L., and Frias M. F., *Parallel Bounded Analysis in Code with Rich Invariants by Refinement of Field Bounds* to appear in Proceedings of ISSTA 2013, pp. 23-33, 2013.

[29] Rosner N., López Pombo C. G., Aguirre N., Jaoua A., Mili A., and Frias M. F., *Parallel Bounded Verification of Alloy Models by TranScoping*, in Proceedings of VSTTE 2013, to appear.

[30] Shlyakhter I., *Generating effective symmetry-breaking predicates for search problems*. In Proceedings of LICS 2001 Workshop on Theory and Applications of Satisfiability Testing, June 2001, Boston, MA. Henry Kautz and Bart Selman (eds.), Electronic Notes in Discrete Mathematics, Vol. 9, 2001.

[31] Siddiqui J. H., and Khurshid S., *PKorat: Parallel generation of structurally complex test inputs.* 2nd International Conference on Software Testing, Verification, and Validation (ICST 2009). Denver, CO. Apr 2009.

[32] Siddiqui J. H., and Khurshid S. *Scaling symbolic execution using ranged analysis*. In *Proc. 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2012.

[33] Shao D., Gopinath D., Khurshid S., Perry D. E., *Optimizing Incremental Scope-Bounded Checking with Data-Flow Analysis*. ISSRE 2010: 408–417.

[34] Shao D., Khurshid S., Perry D. E., *An Incremental Approach to Scope-Bounded Checking Using a Lightweight Formal Method*. FM 2009: 757–772.

[35] Spivey J. M., *The Z Notation: A Reference Manual*, 2nd ed. Upper Saddle River, NJ, Prentice Hall, 1992.

[36] Stoica I., and Morris R., and Liben-Nowell D., Karge D., and Kaashoek M. F., and Balakrishnan H, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, IEEE Transactions on Networking, vol. 11, 2003.

[37] Torlak E., Jackson, D., *Kodkod: A Relational Model Finder*. in TACAS '07, LNCS 4425, pp. 632–647.

[38] Visser W., Havelund K., Brat G., and Park S. *Model checking programs*. In *Proc. 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[39] Visser W., Păsăreanu C. S., Pelánek R., *Test Input Generation for Java Containers using State Matching*, in ISSTA 2006, pp. 37–48, 2006.

[40] Zave, P., *Compositional binding in network domains*. In Proceedings of FM 2006. LNCS 4085, Springer, 332-347.