Exploiting Partial Knowledge for Efficient Model Analysis

Nuno Macedo, Alcino Cunha, and Eduardo Pessoa

INESC TEC & Universidade do Minho, Portugal

Abstract. The advancement of constraint solvers and model checkers has enabled the effective analysis of high-level formal specification languages. However, these typically handle a specification in an opaque manner, amalgamating all its constraints in a single monolithic verification task, which often proves to be a performance bottleneck.
This paper addresses this issue by proposing a solving strategy that exploits user-provided partial knowledge, namely by assigning symbolic bounds to the problem's variables, to automatically decompose a verification task into smaller ones, which are prone to being independently analyzed in parallel and with tighter search spaces. An effective implementation of the technique is provided as an extension to the Kodkod relational constraint solver. Evaluation shows that, in average, the proposed technique outperforms the regular amalgamated verification procedure.

1 Introduction

The steady advancement of constraint solvers and model checkers renders the automatic analysis of software models increasingly efficient. Thus, high-level formal specification languages – like Alloy [9], B [1] or TLA$^+$ [10] – are currently backed up by effective tool support that promotes the effortless specification and analysis of complex systems. In fact, such frameworks have reached a level of maturity that enables their application in industrial scenarios [16].

Nonetheless, such tools are still affected by scalability issues. One approach to tackle this issue is to allow the user to provide additional *a priori* knowledge about the problem's domain, thus reducing its search space. For this effect, the Kodkod [21] model finder supports the definition of *partial instances*, obtaining impressive performance gains. Its language, based on relational logic, is sufficiently simple, yet powerful, to be used directly by end users, but its relevance also lies in its usage by the Alloy Analyzer to automate the analysis of Alloy specifications, and as an alternative back-end for ProB, B's model checker and animator.

Kodkod's partial instances define lower- and upper-bounds for the problem's variables, concretely stating which values must and may be assigned to a variable, respectively. While useful, such bounds are rather inflexible and often do not allow the user to specify all available partial knowledge. In this paper we advocate the support for richer partial instances by allowing the user to declare *symbolic* bounds for the problem's variables. We then show how such partial knowledge can be exploited to improve the performance of automated analysis procedures.

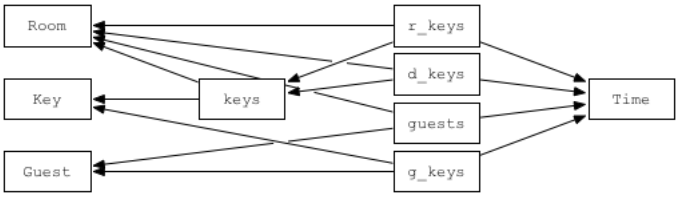


Fig. 1: Dependency graph for the hotel room locking system.

Verification is typically handled opaquely by solvers, resulting in the "*amalgamation*" of the variables and constraints into a single search problem. Symbolic bounds give rise to dependencies that can be used to automatically decompose the amalgamated problem into smaller ones. Candidate *partial solutions* can be generated from a subset of one such decomposed problem, and further refined in independent solving tasks taking into account the remainder problem. This strategy can lead to better scalability since these independent solving tasks *i*) have smaller search spaces (in particular once the symbolic bounds specified by the user are factored in) and *ii*) are prone to being executed in parallel.

Consider the analysis of the specification of a hotel room locking system [9]. The specification of this system in Kodkod consists of a set of `Room`s, a set of `Key`s (assigned to rooms through a relation `keys`) and a set of potential `Guest`s, restricted by appropriate constraints (e.g., the same key cannot be assigned to two different rooms). These elements are static components of the problem, in the sense that, although several assignments are possible, once defined they remain frozen. Other parts of the system are dynamic and evolve over time (explicitly modeled in Kodkod by the set of `Time` instants under analysis). This is the case, for example, of the keys registered at the rooms (relation `r_keys`) or assigned to guests (relation `g_keys`) at any given instant of time. The possible assignments to those relations depend on the static ones (e.g., relation `g_keys` must only relate existing `Key`s to potential `Guest`s inside the `Time` period under analysis). Unfortunately, in Kodkod this inclusion dependency cannot be captured by the bounds defined in partial instances, requiring the user to express it in a normal constraint. Using the extension proposed in this paper, such dependencies can be explicitly declared using symbolic bounds in the partial instance definition (in this case setting the upper-bound to the cross product of the static relations), resulting in the dependency graph depicted in Fig. 1. Using information from the dependency graph (in particular the number of dependencies), our proposed solving strategy will split the problem, first generating candidate partial solutions for a subset of variables. These partial solutions will then be incorporated into new problems extended with the remaining variables and the respective constraints, to be solved in parallel and with tighter bounds once the dependencies are resolved. Naturally, not all of these candidate partial solutions can be extended to full satisfiable instances, meaning that many of them can be discarded in this process.

This paper formalizes the strategy described above and implements it as an extension to Kodkod. Kodkod is well-suited to deploy such strategy due to: *i*)

its native support for partial instances, that will allow an efficient embedding of the partial solutions into the remainder problem; *ii*) its ability to incrementally generate solutions, that will allow the efficient iterative generation of the partial solutions; and *iii*) its powerful symmetry breaking mechanism, that will avoid the generation of isomorphic candidate partial solutions. The implementation of the strategy both relies and preserves these distinctive features. Experimental evaluation of this extension shows that it can indeed outperform Kodkod amalgamated execution for complex problems, particularly for satisfiable (SAT) problems. To balance the performance of the technique for unsatisfiable (UNSAT) problems, we propose a hybrid technique that minimizes performance deterioration in such scenarios, preserving the benefits of the decomposed strategy otherwise.

Section 2 formalizes symbolic partial instances and the decomposed strategy. Section 3 presents its implementation as an extension to Kodkod, which is then evaluated in Section 4. Section 5 compares this work with previously proposed techniques, and Section 6 wraps up and points directions for future work.

2 From Symbolic Bounds to Decomposed Model Finding

The proposed strategy is formalized over *relational model finding* problems as embodied by Kodkod [21]. This formalization is sufficiently powerful and flexible to express general analysis procedures like model checking and animation.

2.1 Relational Model Finding with Symbolic Bounds

Model finders search for variable bindings that satisfy certain problem constraints. In Kodkod, problems are represented by a set of *relations* $\mathcal{R}$ with the associated constraints specified in relational logic (a flavor of first-order logic enhanced with transitive closure). A valid *binding* $b : \mathcal{R} \to \mathcal{T}$, denoted by problem instance in Kodkod, assigns to each relation a tuple set from $\mathcal{T}$, constructed from a universe of atoms $\mathcal{A}$, such that a formula ϕ with free-variables from $\mathcal{R}$ holds. In order to restrict the search space, upper- and lower-bounds are imposed to these relations (known as a *partial instance*). The former typically encode typing restrictions, while the latter may encode partial knowledge about the problem. The tuples comprising these bounds must be uniform on their arity.

Definition 1. *A* (relational) model finding problem P *is a tuple* $\langle \mathcal{A}, l, u, \phi \rangle$ *where* $\mathcal{A}$ *is a universe of atoms,* $l, u : \mathcal{R} \to \mathcal{T}$ *assign to each relation variable* $r \in \mathcal{R}$ *lower- and upper-bounds, respectively, with* $l(v) \subseteq u(v)$*, and* ϕ *is a relational logic formula over* $\mathcal{R}$ *variables. A binding* $b : \mathcal{R} \to \mathcal{T}$ *is a* solution *of* P *if* ϕ *holds and* $b(v) \subseteq u(v) \backslash l(v)$ *for every* $r \in \mathcal{R}$*.*

For simplicity, components of P are denoted as $\mathcal{A}_P$, l_P, u_P and ϕ_P, respectively.

Fig. 2a depicts part of the encoding of the `Hotel` specification in Kodkod, for a problem of size $n = 2$ (n is the max size of sets `Room` and `Guest` and the exact size of `Time` and `Key`). The lower- and upper-bounds appear between square brackets in the declaration of the variables. Notice how equal lower- and upper-bounds fix the

```
A = {R1,R2,K1,K2,G1,G2,T1,T2}
R = Time   : [{T1,T2},{T1,T2}]
    Key    : [{K1,K2},{K1,K2}]
    Room   : [{},{R1,R2}]
    Guest  : [{},{G1,G2}]
    keys   : [{},{(R1,K1),(R2,K1),(R1,K2),(R2,K2)}]
    guests : [{},{(R1,G1,T1),(R2,G1,T1),...,
                  (R1,G2,T2),(R2,G2,T2)}]
    ...
    g_keys : [{},{(G1,K1,T1),(G2,K1,T1),...,
                  (G1,K2,T2),(G2,K2,T2)}]
φ = g_keys in Guest → Key → Time && ... &&
    all k:Key | one keys.k && ... &&
    all t:Time | one r.r_keys.t && ...
```

(a) The `Hotel` problem in normal Kodkod.

```
A   = {R1,R2,K1,K2,G1,G2,T1,T2}
R = Room   : [{},{R1,R2}]
    Key    : [{K1,K2},{K1,K2}]
    Guest  : [{},{G1,G2}]
    keys   : [{},Room → Key]
    Time   : [{T1,T2},{T1,T2}]
    guests : [{},Room → Guest → Time]
    ...
    g_keys : [{},Guest → Key → Time]
φ   = all k:Key | one keys.k && ... &&
      all t:Time | one r.r_keys.t && ...
```

(b) The `Hotel` problem with symbolic bounds.

Fig. 2: Hotel room locking system problem for $n = 2$.

valuation of `Key` and `Time`. The model finding procedure, denoted by $[\![P]\!] : \mathcal{R} \to \mathcal{T}$, searches for an instance for a problem P. Kodkod does so by encoding relations into matrices of boolean variables and computing a propositional formula by converting relational operators into matrix operations, which is then passed to an off-the-shelf SAT solver. If there is no satisfying solution, an empty binding $\bot : \emptyset \to \mathcal{T}$ is assumed to be returned (this is not the same as an empty model that binds every $\mathcal{R}$ variable to an empty tuple set).

Kodkod allows users to iterate through valid instances. This can be embodied by a scenario exploration operation [13] next, that given the previous problem and the last known solution, generates a novel problem to be solved:

$$\mathsf{next}(\langle \mathcal{A}, l, u, \phi \rangle, b) = \langle \mathcal{A}, l, u, \phi \wedge \neg \bar{b} \rangle$$

For a binding b, $\bar{b}$ denotes its encoding into a predicate that exactly characterizes it [13]: by adding the negation of this predicate to the iterated problem's constraints, only different instances will be generated.

Kodkod problems assume relations to be bound by constant tuple sets, as depicted in Fig. 2a for `Hotel`. As mentioned above, upper-bounds usually encode typing restrictions. For instance, we know that a valid binding for `g_keys` is included in the Cartesian product of `Room`, `Key` and `Time`. However, since we are forced to use constants in the declarations, the upper-bound must be approximated by the Cartesian product of the upper-bounds of those sets, and a constraint must be included in the problem to enforce the desired typing restriction (in Kodkod the Cartesian product is denoted by $\to$).

In this paper we propose to extend Kodkod problems with a notion of *symbolic bounds*, allowing users to bound relations with arbitrary Kodkod relational expressions that explicitly refer other relations rather than using just constant tuple sets. This will reduce the verbosity of the declarations and constraints, but most importantly, will expose the dependencies between relations in their declaration, that will be later exploited by the decomposed solving strategy presented in Section 2.2. Using this extension, the upper-bound of relation `g_keys` can now be

declared directly as the desired Cartesian product (Fig. 2b), avoiding the extra constraint in $\phi_{\texttt{Hotel}}$. From such bound declarations the dependency graph of the declared relations can be constructed (see Fig. 1).

Naturally, symbolic bounds must be *resolved* prior to being solved. This action, denoted by $\lfloor P \rfloor$ for a problem P, is achieved by iteratively replacing each relation reference with its respective bounds (lower or upper) and expanding the relational expression. This process must eventually result in constant bounds for every relations, thus the dependency graph must be acyclic. If the relations referred in the symbolic bounds are not bound exactly, the relational expression cannot exactly evaluated, and thus such relational expressions in must also be included in ϕ_P. For instance, the upper-bound `Room` $\rightarrow$ `Key` of relation `keys` would be resolved to `{(R1,K1),(R2,K1),(R1,K2),(R2,K2)}` and constraint `keys in Room` $\rightarrow$ `Key` would be added to $\phi_{\texttt{Hotel}}$, since `Guest` is not exactly bound.

2.2 Decomposed Model Finding

A binding over a subset of variables $\mathcal{R}_p \subseteq \mathcal{R}$ is a candidate *partial solution* of a problem if it is within the bounds defined for $\mathcal{R}_p$ variables and the conjuncts of the constraint defined exclusively over $\mathcal{R}_p$ hold. Let $b|_A$ denote the domain restriction of mapping b to set A, and through an abuse of notation, $\phi|_{\mathcal{R}_p}$ denote the conjuncts of ϕ that refer exclusively to $\mathcal{R}_p$.

Definition 2. *A binding $b : \mathcal{R}_p \rightarrow \mathcal{T}$ is* candidate partial solution *of a model finding problem $P = \langle \mathcal{A}, l, u, \phi \rangle$ with symbolic bounds if $\mathcal{R}_p \subseteq \mathcal{R}$ for $l, u : \mathcal{R} \rightarrow \mathcal{T}$, and it is a solution of the* partial (model finding) problem $P{\downarrow} = \lfloor \langle \mathcal{A}, l|_{\mathcal{R}_p}, u|_{\mathcal{R}_p}, \phi|_{\mathcal{R}_p} \rangle \rfloor$.

This definition assumes the empty binding $\perp$ to be a candidate partial solution of every problem. Partial solutions can be embedded into the bounds of the original problem, binding $\mathcal{R}_p$ relations exactly, leaving only $\mathcal{R}_r = \mathcal{R} \backslash \mathcal{R}_p$ to be solved. Moreover, symbolic bounds in $\mathcal{R}_r$ referring to $\mathcal{R}_p$ variables will be assigned stricter tuple sets after resolution. For relations depending uniquely on $\mathcal{R}_p$ (or other exactly bound relations) resolution will exactly calculate the value of the relational expressions in the symbolic bounds, avoiding the need for additional constraints in ϕ, as in Fig. 2b. Let $\oplus$ denote the overriding of mappings.

Definition 3. *A candidate partial solution $b : \mathcal{R}_p \rightarrow \mathcal{A}$ can be* integrated *into a model finding problem $P = \langle \mathcal{A}, l, u, \phi \rangle$ as $\lfloor \langle \mathcal{A}, l \oplus b, u \oplus b, \phi \rangle \rfloor$, denoted by $P \oplus b$.*

The integration of a candidate partial solution b does not entail a SAT problem by itself, since there may not exist an extension to b for which ϕ holds.

The decomposed model finding strategy will generate candidate partial solutions until an instance is found to an integrated problem. This strategy is encoded in Algorithm 1, relying only on regular model finding procedures. Essentially, given the current state of the partial problem $P{\downarrow}$, the procedure successively generates candidate partial solutions p, that are integrated into the full problem until a full solution is found or $\perp$ is returned, rendering P UNSAT. Suppose P

Input: A model finding problem $P = \langle \mathcal{A}, \mathcal{R}, l, u, \phi \rangle$ and a subset of variables $\mathcal{R}_p \subseteq \mathcal{R}$.
Output: A solution for the problem P or $\bot$, and the updated formula for the partial problem.

$P{\downarrow} \leftarrow \langle \mathcal{A}, l|_{\mathcal{R}_p}, u|_{\mathcal{R}_p}, \phi|_{\mathcal{R}_p} \rangle$
repeat
  $p \leftarrow [\![\lfloor P{\downarrow} \rfloor]\!]$;
  $b \leftarrow [\![\lfloor P \oplus p \rfloor]\!]$;
  if $b = \bot$ **then**
    $P{\downarrow} \leftarrow \mathsf{next}(P{\downarrow}, p)$;
  end
until $b \neq \bot \vee p = \bot$;
return $\langle b, \phi_{P{\downarrow}} \rangle$;

Algorithm 1: Algorithm for decomposed model finding.

```
Room   : [{R1},{R1}]
Key    : [{K1,K2},{K1,K2}]
Guest  : [{G1},{G1}]
keys   : [{(R1,K1),(R1,K2)},
          {(R1,K1),(R1,K2)}]
```

(a) Candidate partial solution p obtained from $\lfloor$`Hotel`$\downarrow\rfloor$.

```
Time   : [{T1,T2},{T1,T2}]
r_keys : [{},{(R1,K1,T1),(R1,K1,T1),(R1,K2,T2),(R1,K2,T2)}]
...
g_keys : [{},{(G1,K1,T1),(G1,K1,T1),(G1,K2,T2),(G1,K2,T2)}]
```

(b) Result of integrating and resolving p into `Hotel`, i.e. $\lfloor$`Hotel` $\oplus\, p\rfloor$.

Fig. 3: Decomposed model finding for `Hotel`.

to be the `Hotel` problem in Fig. 2b and $\mathcal{R}_p$ to include `Room`, `Key`, `Guest` and `keys`. Solving $\lfloor P{\downarrow} \rfloor$ could produce the candidate partial solution in Fig. 3a. After being integrated into P and resolving the symbolic bounds as $\lfloor P \oplus p \rfloor$, the constant bounds in Fig. 3b are obtained. Note how these are considerable smaller than those defined in the amalgamated problem in Fig. 2a, potentially speeding up the solving of the integrated problem. This algorithm is prone to being parallelized on the exploration of the integrated problems, as will be shown in Section 3.

The procedure is complete, since every partial solution may be eventually explored. To speed up instance iteration, the current formula $\phi_{P{\downarrow}}$ of the partial problem is also returned to avoid the generation of candidate partial solutions p that have already been fully explored (i.e., for which problem $[\![\lfloor P \oplus p \rfloor]\!]$ has already returned $\bot$). This process cannot be managed by `next` since the partial solution under instance b may still produce additional full solutions. Iteration of decomposed problems is thus captured by the following definition, that given the output of Algorithm 1 returns a new problem that can be inputed back into it:

$$\mathsf{next}(\langle \mathcal{A}, \mathcal{R}, l, u, \phi \rangle, \langle b, \phi_{P{\downarrow}} \rangle) = \langle \mathcal{A}, \mathcal{R}, l, u, \phi \wedge \neg \bar{b} \wedge \phi_{P{\downarrow}} \rangle$$

2.3 Criterion for Decomposing Problems

The previous section has shown how to decompose the model finding of a problem given a subset of variables $\mathcal{R}_p$. This subset can be defined manually, but ideally,

it should be derived automatically, and several criteria can be proposed to do so. The usage of symbolic bounds enabled us to define a simple criterion that lead to substantial gains in efficiency in most examples in our evaluation.

Looking at the entailed dependency graph and given a threshold t, relations with outdegree (number of dependencies) bigger than t, or that depend directly or indirectly from one of those, are left out of $\mathcal{R}_p$. The intuition behind this criterion is that variables with more dependencies benefit more from prior solving. If more than a single connected component is left, only the largest one if kept in $\mathcal{R}_p$. The number of candidate partial solutions should be manageable, and such disconnected components usually give rise to an explosion of non-symmetric solutions, unlike connected relations whose valuations are most likely restricted by the constraints. We also found that setting t to the maximum outdegree typically provides optimal performance. In our running example, this would assign to $\mathcal{R}_p$ `Room`, `Key`, `Guest` and `keys`, resulting in a behavior similar to that of Fig 3. The criterion was applied to all examples considered in Section 4.

3 Decomposed Kodkod

This section describes a concrete implementation of the decomposed strategy described in the previous section as an extension to Kodkod [11].

3.1 Implementation Overview

The decomposed solver implements the strategy presented in Section 2.2: given a problem P and a set of $\mathcal{R}_p$ variables, the procedure automatically extracts $\phi_P|_{\mathcal{R}_p}$ depending on the occurrence of $\mathcal{R}_p$ variables, and then solves the problem following the general idea behind Algorithm 1. $P{\downarrow}$ is deployed as a regular Kodkod problem and generates candidate partial solutions p_i, and for each p_i, an integrated problem $P \oplus p_i$ is created that can also be deployed under regular Kodkod. To avoid unnecessary translations, $\phi_P|_{\mathcal{R}_p}$ is not included in the integrated problem, since it is already known to hold for p_i. However, unlike the abstract formalization from Section 2.2, these integrated problems are launched in parallel rather than explored sequentially. The number i of candidate partial solutions is unknown *a priori*, so a (configurable) threshold is imposed on the number of launched parallel threads. The state of $P{\downarrow}$ is also internally preserved, rather than being constructed at each iteration, benefiting from the performance gains of incremental SAT solving. When one of the $P \oplus p_i$ procedures finishes and is SAT, the full solution s_{i_k} is pushed into a blocking queue that the user can inspect. UNSAT integrated problem are discarded. Remainder integrated problems keep being solved and launched in the background until the blocking queue fills up, providing a buffer of full solutions.

When the user asks for another solution succeeding s_{i_k}, the system iterates the $P \oplus p_i$ problem (by negating the full solution s_{i_k} into it), and pushes it to the execution queue (which is LIFO since it is cheaper to solve iterated problems). Nonetheless, other integrated problems executing in the background could have

already pushed solutions into the queue, so there is no guarantee that succeeding full solutions will share the same partial solution. Thus, although the set of candidate partial solutions explored is identical, iteration order differs from that of Algorithm 1. The set of solutions returned by each of the integrated problem is disjoint since partial solutions are unique. Moreover, SAT integrated problems are directly and independently iterated (unlike the sequential Algorithm 1 that iterated the overall decomposed problem).

For UNSAT problems, every candidate partial solution must be explored, which may entail an overwhelming number of integrated problems to solve. As will be evident in Section 4, this may be a bottleneck for the decomposed strategy. To address this issue, a *hybrid* strategy is proposed where the integrated problems are paired with a thread solving the amalgamated problem P. In the worst case, P will finish first and be handled as a regular model finding problem (terminating the running integrated problems); in the best case a SAT (or *every* UNSAT) integrated problem will finish before P, terminating it. This guarantees no repeated full solutions are returned. This strategy resembles portfolio parallel SAT solving [8], where identical solvers with different parameters competitively solve the same problem. Nonetheless, the hybrid approach is expected to have slightly deteriorated performance due to cache interference.

3.2 Symmetry Breaking

Symmetry breaking greatly reduces the number of generated solutions by determining equivalences between atoms and avoiding the generation of instances considered isomorphic. This is particularly relevant when solving partial problems as it determines the number of integrated problems that will be launched. Kodkod's symmetry breaking procedure starts by detecting the symmetries of a problem based on its bounds [21], and then generates a symmetry breaking predicate [6] that is added to the problem's constraint. This section describes how these procedures were adapted in order to be sound in the decomposed scenario. Symbolic bounds are assumed to be resolved at this point.

Symmetry detection searches for atom permutations that map valid bindings to valid bindings and invalid to invalid based on the declared bounds. For instance, relation `s : [{},{(A1,B1),(A2,B1)}]` induces a symmetry `{A1,A2}`, since permuting these two atoms results in identical bounds (see [21] for technical details). Thus, solutions `s = {(A1,B1)}` and `s = {(A2,B1)}` are considered isomorphic. Clearly, the fixed valuation for $\mathcal{R}_p$ relations in integrated problems cannot be considered as these would break additional symmetries: assuming $\mathcal{R}_r$ `= s` and $\mathcal{R}_p$ `= r :[{},{(A1,B1),(A2,B1)}]`, partial solution `r = {(A1,B1)})` would break the symmetry between `A1` and `A2`, distinguishing `s = {(A1,B1)}` from `s = {(A2,B1)}`. The bounds of $P{\downarrow}$ may also cause incongruences if considered independently. For `r :[{},{(A1,B1),(A2,B1)}]` and `s : [{},{(A1,B1),(A1,B2)}]` no symmetries are detected, but if $\mathcal{R}_p$ `= r` and $P{\downarrow}$ considers only `r`, symmetry `{{A1,A2}}` would be detected, meaning that it will return, for instance, solution `r = {(A1,B1)}` and not `r = {(A2,B1)}`. The issue persists in integrated problems, as considering only `s` would result in the symmetry `{B1,B2}`. To preserve the

soundness of the symmetry detection procedure, the original bounds P_l and P_u of every relation $\mathcal{R}$ must be considered in both the partial and integrated problems. Relations not relevant to each problem should then be ignored when generating the symmetry breaking predicate. Our extension implements this strategy.

The generation of the symmetry breaking predicate imposes an ordering on the boolean variables resulting from the translation of the relations $\mathcal{R}$ into SAT, constructs a lexicographical order over them and generates predicates that force minimal valuations (see [6] for details). The main insight is that the variable ordering must be preserved between the partial problem and the integrated problems, otherwise the procedure will not be sound. Consider an example with `r : [{},{A1,A2}]` and `s : [{},{A1,A2}]`, producing 4 boolean variables, $\mathtt{r}_{\mathtt{A1}}, \mathtt{r}_{\mathtt{A2}}, \mathtt{s}_{\mathtt{A1}}$ and $\mathtt{s}_{\mathtt{A2}}$, denoting whether `A1` and `A2` belong to `r` and `s`, respectively. Since `A1` and `A2` are symmetric, a lexicographical order $[\mathtt{r}_{\mathtt{A1}}, \mathtt{s}_{\mathtt{A1}}] \leq [\mathtt{r}_{\mathtt{A2}}, \mathtt{s}_{\mathtt{A2}}]$ will be constructed, allowing 10 different valuations for the boolean variables. Now, if $\mathcal{R}_p$ `= s`, 3 partial solutions will be generated, `s = {}`, `s = {A2}` and `s = {A1,A2}`, giving rise to 3 integrated problems with $[\mathtt{r}_{\mathtt{A1}}, \mathsf{F}] \leq [\mathtt{r}_{\mathtt{A2}}, \mathsf{F}], [\mathtt{r}_{\mathtt{A1}}, \mathsf{F}] \leq [\mathtt{r}_{\mathtt{A2}}, \mathsf{T}]$ and $[\mathtt{r}_{\mathtt{A1}}, \mathsf{T}] \leq [\mathtt{r}_{\mathtt{A2}}, \mathsf{T}]$. These allow only 9 valuations: solution `r = {A2}` and `s = {A1}` will be disregarded. If the ordering is preserved, problems with $[\mathsf{F}, \mathtt{r}_{\mathtt{A1}}] \leq [\mathsf{F}, \mathtt{r}_{\mathtt{A2}}], [\mathsf{F}, \mathtt{r}_{\mathtt{A1}}] \leq [\mathsf{T}, \mathtt{r}_{\mathtt{A2}}]$ and $[\mathsf{T}, \mathtt{r}_{\mathtt{A1}}] \leq [\mathsf{T}, \mathtt{r}_{\mathtt{A2}}]$ allow the expected 10 solutions. Our implementation guarantees that the ordering is preserved between partial and integrated problems by prioritizing $\mathcal{R}_p$ variables.

4 Empirical Evaluation

To evaluate the performance of the procedure, several Kodkod problems with scalability problems were collected. `Hotel(1)` is a SAT version of the `Hotel` specification where a counter-example is found, and `Hotel(2)` is a fixed UNSAT version. `RBT(1)` is a structural problem that generates red-black trees (SAT) with n nodes, while `RBT(2)` checks whether every red-black tree is balanced (UNSAT). `Hand` is a structural problem that models the Halmos handshake puzzle for n persons: `Hand(1)` generates instances of the puzzle (SAT) and `Hand(2)` checks whether the answer to the puzzle holds (UNSAT). Regarding dynamic problems, `Dijk` models Dijkstra's mutual exclusion algorithm for n processes and mutexes: `Dijk(1)` searches for a valid instance (SAT) while `Dijk(2)` checks whether deadlocks may occur (UNSAT). `Ring` models a leader election algorithm over ring network topologies: `Ring(1)` checks a liveness property that fails (SAT), in `Ring(2)` the liveness property holds (UNSAT), and `Ring(3)` checks a safety property that holds (UNSAT). Finally, `Span` models a distributed algorithm that calculates the spanning tree of a graph, with `Span(1)` and `Span(2)` searching for instances with different properties (both SAT). For `Ring` and `Span`, n denotes the number of nodes in the network. Since Kodkod may only perform bounded model checking, trace length t of 15 was imposed on `Dijk`, 20 on `Hotel` and `Ring`, and 9 on `Span`. These problems, available in the code repository [11], range from very few candidate partial solutions to tens of thousands, as well as from low to high satisfiability ratios. All were modeled with symbolic bounds, which have a

larger impact in the search space of `Dijk` and `Hotel`, and decomposed according to the criterion from Section 2.3.

4.1 Setup

Tests were run in amalgamated, parallel and hybrid mode, with and without symbolic bounds, for increasing n sizes, with a timeout (TO) of 10000 seconds. Problems solved under a second are not presented since performance differences would be negligible. The most efficient SAT solvers supported by the latest version of Kodkod [20] were used, namely Glucose [2] and MiniSat [7]. The performance tests were run on commodity hardware, namely in a quad core 4 GHz Intel Core i7 with hyperthreading, with 8 GB memory and running OS X 10.10.

Decomposed problems were solved with 4 parallel integrated problems (3 in hybrid mode). Our tests show that, as expected, the performance of the purely parallel approach increases with the number of threads. However, in hybrid mode, the performance is deteriorated when the amalgamated problem terminates first, due to cache interference. For instance, in `RBT(1)` for $n = 10$, an integrated problem terminates first, at 3.8s, 4.0s and 4.3s for 2, 4 and 6 threads, respectively. In contrast, in `RBT(2)` at $n = 8$ the amalgamated problem terminates first, at 1.3s, 1.5s and 1.8s for 2, 4 and 6 threads, respectively. The 4 threads provide a reasonable balance between the benefits of the decomposition while still relying on the amalgamated problem in the worst case scenario.

Since the parallelization of the solving process is at the core of our approach, its performance was also compared with that of state-of-the-art parallel SAT solvers over amalgamated problems. Both Syrup [2] (Glucose's parallel version) and Plingeling [5] (Lingeling's parallel version) were considered, which ranked at the top of the most recent SAT race[1]. Plingeling is the only parallel SAT solver currently distributed with Kodkod, although Syrup could be trivially added since Glucose is already supported. It should be noted, however, that, unlike our technique, these parallel SAT solvers do not yet support incremental executions, and as such cannot be used to efficiently iterate through alternative solutions.

The results for the SAT and UNSAT problems using Glucose are summarized in Tables 1 and 2, respectively, including the number of candidate partial solutions ($p_{\#}$), the satisfiability ratio ($p_{\%}$, estimated for larger $p_{\#}$ values), the performance of the amalgamated (T_0), purely parallel with regular (T_p) and symbolic bounds (T_s) and hybrid procedures with symbolic bounds (T_h), as well as the performance gain between T_0 and T_h (G). The results are detailed in Fig. 4 for `RBT` and `Hotel`, including Syrup's performance.

4.2 Satisfiable Problems

For most problems (`Dijk(1)`, `Hand(1)`, `Hotel(1)` and `RBT(1)`) the hybrid approach considerably outperforms the amalgamated execution, even for problems with a large number of candidate partial solutions and reduced satisfiability ratio,

[1] `http://baldur.iti.kit.edu/sat-race-2015/`.

like RBT(1). In fact, the speedup may reach orders of magnitude, like Hand(1) at $n = 16$, Hotel(1) at $n = 11$ and RBT(1) at $n = 12$, with speedups of 916x, 1086x and 118x, respectively. At larger n values, several amalgamated problems timeout while the decomposed procedure still takes few seconds to execute. As Figs. 4a and 4c show, for SAT problems the performance of amalgamated executions tends to increase exponentially, unlike the decomposed strategy. For Span(2) speedups are less significant, going up to 4.6x. Finally, for Ring(1) and Span(1), results range from slowdowns of 0.5x to speedups of 1.2x. Here, it can be seen that the purely parallel approach would actually be outperformed by the amalgamated procedure, but the hybrid mode balances the losses. However, these problems were solved below 6s, thus these differences are not very significant. For the specifications for which symbolic bounds impact the search space, (Dijk(1) and Hotel(1)) the purely parallel approach shows in average a 4x speedup. For the other specifications, performance differences are marginal, as expected.

Results with MiniSat (not shown in the table) in general mirror those obtained with Glucose. For instance, Hand(1) at $n = 14$, Hotel(1) at $n = 9$ and RBT(1) at $n = 10$, have speedups of 528x, 376x and 14x, respectively. Regarding the comparison with parallel SAT solvers, Syrup follows the tendency of Glucose, as hinted by Figs. 4a and 4c, albeit with considerably improved performance. Thus, problems that were considerably outperformed by the decomposed strategy remain so: Hand(1) at $n = 16$, Hotel(1) at $n = 11$ and RBT(1) at $n = 12$, have speedups of 650x, 255x and 7x, respectively. For Ring(1), Span(1) and Span(2), which are solved below 6s, Syrup actually performs slightly worse than Glucose. For the considered specifications, Plingeling is usually outperformed by Syrup, so the same conclusions apply.

4.3 Unsatisfiable Problems

Although, as expected, the purely parallel execution is often outperformed by the amalgamated execution, the hybrid execution is able to compensate the losses. In fact, results show that the amalgamated approach is never more than 2x faster than the hybrid approach for the considered specifications under considerable n sizes. Figs. 4b and 4d depict the overall tendency, with the hybrid execution mostly accompanying the performance of the amalgamated execution.

Specifications where the amalgamated execution outperforms the hybrid strategy are usually balanced by the results of their SAT counter-part. For instance, for the UNSAT Dijk(2), the hybrid approach is about 1.5x slower than the amalgamated approach for every n size; however, for the SAT Dijk(1), the speedup of the hybrid approach ranges from 20x to 30x. Thus, in average, the performance of the decomposed strategy outperforms the amalgamated approach. Interestingly, for RBT(2) and Ring(2), the hybrid approach actually outperforms amalgamated execution: for RBT(2) at $n = 11$ there is a 6x speedup, and for Ring(2) at $n = 5$ a 553x speedup. For larger n values the amalgamated execution times out, while the hybrid approach still terminates within reasonable time. The impact of using symbolic bounds is also manifest in the UNSAT scenarios.

Model	n	$p_{\#}$ ($p_{\%}$)	T_0	T_p	T_s	T_h	G
Dijk(1)	27	28 (0.93)	25.3	4.8	1.1	**1.1**	23.3
Dijk(1)	28	29 (0.93)	30.6	4.8	1.2	**1.2**	26.4
Dijk(1)	29	30 (0.93)	43.9	5.4	1.3	**1.3**	32.7
Dijk(1)	30	31 (0.94)	28.3	5.7	1.4	**1.4**	20.1
Hand(1)	14	1 (1.00)	81.5	0.6	0.6	**0.6**	127.3
Hand(1)	15	0 (0.00)	2.1	0.1	0.1	**0.1**	14.3
Hand(1)	16	1 (1.00)	1496.2	1.6	1.6	**1.6**	916.3
Hand(1)	17	0 (0.00)	40.6	0.1	0.1	**0.2**	240.7
Hand(1)	18	1 (1.00)	TO	4.7	4.7	**5.1**	$+\infty$
Hand(1)	19	0 (0.00)	2724.8	0.2	0.2	**0.2**	13910.2
Hand(1)	20	1 (1.00)	TO	874.5	871.3	**1047.2**	$+\infty$
Hotel(1)	8	12833 (0.60)	64.1	2.9	1.0	**0.9**	71.6
Hotel(1)	9	211470 (>0.6)	216.0	3.4	1.0	**0.9**	236.5
Hotel(1)	10	>999999 (>0.6)	224.4	4.0	1.1	**0.9**	237.2
Hotel(1)	11	>999999 (>0.6)	1106.1	4.5	1.2	**1.0**	1086.3
Hotel(1)	12	>999999 (>0.6)	184.8	4.9	1.3	**1.0**	178.2
RBT(1)	9	4862 (0.01)	1.6	0.5	0.5	**0.5**	2.9
RBT(1)	10	16796 (0.00)	55.8	3.1	3.3	**4.0**	13.9
RBT(1)	11	58786 (0.00)	240.8	1.6	1.6	**2.0**	121.6
RBT(1)	12	208012 (0.00)	2350.5	17.0	17.3	**19.9**	118.2
RBT(1)	13	>999999 (0.00)	TO	133.3	133.5	**164.6**	$+\infty$
Ring(1)	8	16072 (>0.1)	**1.9**	193.8	196.2	4.2	0.5
Ring(1)	9	125673 (>0.1)	1.5	1.1	1.2	**1.2**	1.2
Ring(1)	10	>999999 (>0.1)	**1.6**	2.2	2.1	2.0	0.8
Ring(1)	11	>999999 (>0.1)	**2.6**	61.8	62.1	5.1	0.5
Ring(1)	12	>999999 (>0.1)	**3.4**	20.4	19.1	5.4	0.6
Span(1)	14	>999999 (1.00)	**1.1**	1.7	2.0	1.8	0.6
Span(1)	15	>999999 (1.00)	**1.3**	2.4	2.4	2.2	0.6
Span(1)	16	>999999 (1.00)	**2.0**	2.6	2.7	2.7	0.7
Span(2)	14	>999999 (1.00)	15.2	3.5	3.5	**3.3**	4.6
Span(2)	15	>999999 (1.00)	9.0	4.2	4.0	**4.2**	2.1
Span(2)	16	>999999 (1.00)	6.3	4.4	4.5	**5.0**	1.3

Table 1: Summary of the SAT performance tests.

Likewise the SAT case, MiniSat results are similar to those of Glucose. For instance, `Dijk(2)` preserves the 1.5x slowdown in average, albeit at slightly higher performance times, while `Ring(2)` at $n = 5$ the amalgamated execution times out while the hybrid takes 18s. Comparing with parallel SAT solvers, the results are also similar to those obtained for the SAT problems. For instance, for `Ring(2)` at $n = 5$ the speedup persists but reduced to 244x, while the slowdown at `Dijk(2)` is in average increased to 2x. This phenomenon is hinted in Figs. 4b and 4d, where Syrup has a similar growth curve to that of Glucose but with better performance. Plingeling continues to be in general outperformed.

4.4 Threats to Validity

The performance of the decomposed strategy is highly dependent on the order on which the candidate partial solutions are generated, since they determine the satisfiability of the integrated problems. However, since the generation of partial solutions is extremely efficient, and the UNSAT integrated problems are often quickly discharged, our technique has been able to handle problems with very large number of partial solutions and very small satisfiability ratio, like `RBT(1)`.

The partition criteria automatically inferred from the symbolic bounds is not necessarily optimal. However, manual experiments have not found any better

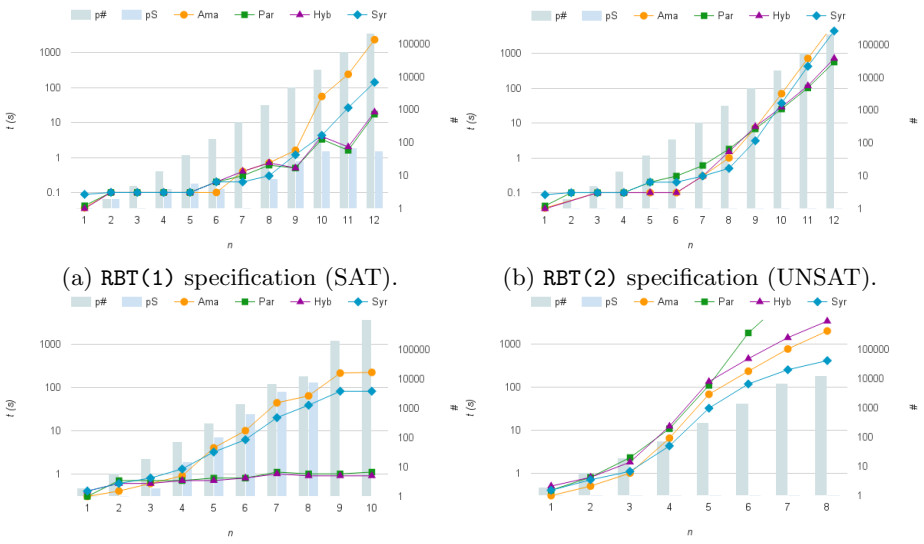
Model	n	$p_{\#}$ ($p_{\%}$)	T_0	T_p	T_s	T_h	G
Dijk(2)	27	28 (0.00)	**23.9**	79.0	55.2	37.6	0.6
Dijk(2)	28	29 (0.00)	**24.7**	98.7	67.8	37.1	0.7
Dijk(2)	29	30 (0.00)	**31.1**	115.4	80.4	47.2	0.7
Dijk(2)	30	31 (0.00)	**32.0**	132.8	93.7	49.6	0.6
Hand(2)	12	1 (0.00)	**6.0**	6.6	6.9	**5.1**	1.2
Hand(2)	13	0 (0.00)	0.2	0.1	0.1	**0.1**	1.8
Hand(2)	14	1 (0.00)	**127.3**	724.2	162.7	**122.9**	1.0
Hand(2)	15	0 (0.00)	2.5	0.1	0.1	**0.1**	16.8
Hand(2)	16	1 (0.00)	**2537.5**	TO	TO	2912.8	0.9
Hotel(2)	4	75 (0.00)	**6.5**	12.7	10.8	12.3	0.5
Hotel(2)	5	312 (0.00)	**68.2**	128.7	109.9	134.6	0.5
Hotel(2)	6	1421 (0.00)	**234.7**	1973.7	1820.5	460.3	0.5
Hotel(2)	7	7016 (0.00)	**772.1**	TO	TO	1416.5	0.5
Hotel(2)	8	12833 (0.00)	**2023.6**	TO	TO	3432.4	0.6
RBT(2)	9	4862 (0.00)	**7.3**	6.7	6.8	7.9	0.9
RBT(2)	10	16796 (0.00)	70.1	25.0	25.4	**28.8**	2.4
RBT(2)	11	58786 (0.00)	721.7	100.3	102.8	**118.1**	6.1
RBT(2)	12	58786 (0.00)	TO	567.8	564.9	**721.7**	$+\infty$
Ring(2)	4	24 (0.00)	10.8	0.7	0.7	**3.1**	3.5
Ring(2)	5	89 (0.00)	4486.8	6.6	6.6	**8.1**	552.8
Ring(2)	6	415 (0.00)	TO	238.3	237.1	**318.0**	$+\infty$
Ring(3)	5	89 (0.00)	**1.4**	2.5	2.4	2.8	0.5
Ring(3)	6	415 (0.00)	**4.9**	13.4	13.3	8.9	0.6
Ring(3)	7	2372 (0.00)	**14.6**	105.8	104.8	24.4	0.6
Ring(3)	8	16072 (0.00)	**76.1**	1100.5	1098.1	134.7	0.6
Span(1)	5	58 (0.00)	**0.2**	7.6	1.2	0.3	0.7
Span(1)	6	457 (0.00)	**0.6**	342.9	7.8	1.0	0.6
Span(1)	7	5777 (0.00)	**4.4**	663.4	337.2	9.0	0.5
Span(2)	5	58 (0.00)	**0.4**	3.1	2.9	0.7	0.6
Span(2)	6	457 (0.00)	**0.6**	30.0	30.1	1.1	0.5
Span(2)	7	5777 (0.00)	**1.9**	786.9	780.4	4.2	0.5

Table 2: Summary of the UNSAT performance tests.

partition for the considered problems. Nonetheless, the soundness of the decomposed strategy would be preserved by alternative partition criteria, and since our tool accepts the set $\mathcal{R}_p$ of variables that will determine the partial problem, the user is free to manually define the decomposition or experiment with other automated criteria.

5 Related Work

The decomposition of Alloy models into smaller problems to improve the performance of the solving process has been previously explored [22]. Likewise our technique, constraints are split in two, and solutions to the first are fed as partial information to the second. Partition criteria are chosen by testing candidates at small scopes. The iteration of solutions and symmetry breaking are not addressed. Evaluation mainly focuses on the small scope tests for SAT problems, with speedups not reaching an order of magnitude. The partitioning and parallelization of Alloy analysis procedures has also been proposed [18]. Here, each parallel problem solves the same constraints but within a restricted search space, defined by a range of solutions. Ranges are derived from the structure of the models, disregarding the constraints, resulting in unpredictable complexity. This is addressed by allowing the dynamic partition of problems. In a different



(a) `RBT(1)` specification (SAT).

(b) `RBT(2)` specification (UNSAT).

(c) `Hotel(1)` specification (SAT).

(d) `Hotel(2)` specification (UNSAT).

Fig. 4: Performance times for the `RBT` and `Hotel` specifications.

study [17], the same authors explore a technique to infer partitions on the SAT propositional variables from high- level Alloy models with small scopes. Likewise our strategy, both techniques obtain speedups up to two orders of magnitude before the amalgamated analyses timeout, but since evaluation is performed in clusters direct comparisons should be read with care. By allowing the definition of symbolic bounds, our approach is able to explore additional knowledge about the problem's domain without burdening the user, since that information would still have to be integrated in the constraints otherwise.

Techniques have been proposed to extract finer Kodkod partial instances from high-level specifications, still relying on its constant tuple set bounds. In [15] an extension to the Alloy language for the specification of instances is proposed, that can be mapped into Kodkod bounds. Our approach extends the expressiveness of partial instances at the Kodkod level. Since Alloy natively support binding expressions in the declaration of the relations, symbolic bounds could easily be retrieved from regular Alloy specifications without any extension to the language.

Many techniques have been proposed for parallelizing SAT solvers [8,14], most based on the *Conflict-Driven Clause Learning* (CDCL) algorithm and exploiting clause learning and sharing. In general, these fit into two families [8]: competitive (or portfolio) approaches, where the solvers explore the same search space, the fastest returning the solution; and cooperative approaches, following a divide and conquer strategy, where the search space is split and the solution is built from the results of the solvers. Focusing on top ranking solvers from the latest SAT competitions, Plingeling [4] falls in the former category, deploying solvers with different configurations, with minimal clause sharing, while Syrup [3] follows

an hybrid approach, with an initial portfolio phase that switches to cooperative after a certain threshold. These solvers, however, do not support incremental solving, and thus cannot be used to effectively iterate solutions. Moreover, our technique could be adapted to run in a distributed environment, unlike modern solvers based on clause sharing.

A parallel SAT solving approach that is more closely related to ours is the one followed by JaCk-SAT [19]. Here, the set of boolean variables is split through heuristics, and the clauses are divided accordingly. Problems are then deployed in parallel to solve the two sets of variables independently; solutions are then checked over the clauses referring to both sets of variables, and are rejected if not. This process can be repeated recursively. This technique, however, is not able to compete with the performance of modern parallel SAT solvers.

6 Conclusions

This paper proposes the usage of symbolic partial knowledge to enhance the analysis of declarative specifications through their automatic decomposition into partial solutions and subsequent parallelization of the solving process with tighter search spaces. This strategy is formalized and an effective implementation for the Kodkod constraint solver is provided. This extension is able to automatically analyze relational model finding problems, while still preserving the ability to iterate over solutions and the soundness of the symmetry breaking algorithm, and exploit partial knowledge for increased efficiency.

Our evaluation has shown that, even in commodity hardware, the technique is able to outperform amalgamated problems for most satisfiable specifications; the hybrid approach addressed the worst case scenarios, providing balanced results in comparison with the amalgamated execution. In fact, it rivals with state-of-the-art parallel SAT solvers. We also show that decomposing the problems based on the dependency degree of the problem's variables is a suitable partition criteria.

Although we believe Kodkod to be powerful enough to be used by end users, we expect the extension presented to be exploited by analyzers for high-level specifications. In the future we intend to derive symbolic bounds directly from the binding expressions of Alloy's declarations, thus benefiting the large community of Alloy users. The decomposed strategy is already being used in the back-end of Electrum [12], a temporal extension to Alloy. As in `Hotel`, in such scenarios the partition criteria naturally degenerates into a division between the static and dynamic variables. To support full (non-bounded) model checking for such problems, we are currently exploring a generalization where Kodkod is used for the generation of the static partial solutions, while the integrated dynamic problems are checked in parallel by off-the-shelf model checkers, such as NuSMV.

References

1. J. Abrial. *The B-book – Assigning programs to meanings.* Cambridge University Press, 2005.

2. G. Audemard and L. Simon. Glucose, version 4.0. Available at `http://alloy.mit.edu/kodkod/download.html`, October 2014.
3. G. Audemard and L. Simon. Lazy clause exchange policy for parallel SAT solvers. In C. Sinz and U. Egly, editors, *SAT'14, held as part of VSL'14*, volume 8561 of *LNCS*, pages 197–205. Springer, 2014.
4. A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Technical Report 10/1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, 2010.
5. A. Biere. Plingeling, version ayv-86bf266-140429. Available at `http://fmv.jku.at/lingeling/`, April 2014.
6. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR'96*, pages 148–159. Morgan Kaufmann, 1996.
7. N. Eén and N. Sörensson. MiniSat, version 2.2.0. Available at `http://minisat.se/MiniSat.html`, July 2010.
8. S. Hölldobler, N. Manthey, V. H. Nguyen, J. Stecklina, and P. Steinke. A short overview on modern parallel SAT-solvers. In *AICACSIS'11*, pages 201–206. IEEE, 2011.
9. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.
10. L. Lamport. *Specifying Systems, The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
11. N. Macedo. Pardinus, version 0.3. Available at `https://github.com/nmacedo/Pardinus/`, September 2016.
12. N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *FSE'16*. ACM, 2016.
13. N. Macedo, A. Cunha, and T. Guimarães. Exploring scenario exploration. In *FASE'15*, volume 9033 of *LNCS*, pages 301–315. Springer, 2015.
14. R. Martins, V. M. Manquinho, and I. Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.
15. V. Montaghami and D. Rayside. Extending Alloy with partial instances. In *ABZ'12*, volume 7316 of *LNCS*, pages 122–135. Springer, 2012.
16. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.
17. N. Rosner, C. G. L. Pombo, N. Aguirre, A. Jaoua, A. Mili, and M. F. Frias. Parallel bounded verification of Alloy models by TranScoping. In *VSTTE'13*, volume 8164 of *LNCS*, pages 88–107. Springer, 2013.
18. N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias. Ranger: Parallel analysis of Alloy models by range partitioning. In *ASE'13*, pages 147–157. IEEE, 2013.
19. D. Singer and A. Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In *PPAM'07*, volume 4967 of *LNCS*, pages 249–258. Springer, 2007.
20. E. Torlak. Kodkod, version 2.1. Available at `http://alloy.mit.edu/kodkod/download.html`, September 2015.
21. E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS'07*, volume 4424 of *LNCS*, pages 632–647. Springer, 2007.
22. E. Uzuncaova and S. Khurshid. Constraint prioritization for efficient analysis of declarative models. In *FM'08*, volume 5014 of *LNCS*, pages 310–325. Springer, 2008.