Subsumer-first: A new Heuristic for Guided Symbolic Reachability Analysis

Andrey Rybalchenko[1] and Rishabh Singh[2]

[1] MPI-SWS
[2] MIT

Abstract. State space exploration using symbolic techniques provides a basis for the verification of software systems. The exploration procedure has direct impact on the overall effectiveness of the verification efforts. For example, choosing the breadth-first exploration strategy results in a verification tool that finds counterexamples quickly, but may sacrifice the efficiency of the reachability analysis. The existing exploration heuristics, such as A^* search or chaotic iteration, are geared towards optimizing only one objective, e.g., counterexample length or convergence of fixpoint computation, while neglecting various others that maybe of significant importance as well. In this paper, we present a new subsumer guided heuristic for symbolic state space exploration that supports both efficient counterexample discovery and quick convergence of the reachability computation. The Subsumer-first heuristic leverages the results of partial fixpoint checks performed during the symbolic state space exploration. We present an application of the heuristic for improving efficiency of abstraction-based software verification. Our experimental evaluation of the heuristic in a predicate abstraction-based tool indicates its practical applicability, as we observe significant efficiency improvement (median of 40%) on difficult benchmarks from the transportation domain.

1 Introduction

Model Checking [7] is a popular verification technology employed to verify both hardware and software systems [1, 5, 14]. State space exploration using symbolic techniques provides a basis for the verification of software systems. Explicit-state model checking which is commonly used to analyze software systems face the fundamental problem of *state space explosion* during this exploration. Scalability is one of the major challenges that model checking techniques face today which limits their applicability to verifying large systems. There are various efforts made to overcome this bottleneck. Techniques like abstraction interpretation [8] try to abstract only relevant properties of the program to prove its correctness. Symbolic model checking [4] avoids explicit construction of the state space by performing symbolic fixpoint computation. Partial-order reduction [17] tries to explore only representative transitions and ignores other redundant transitions. But these techniques still suffer from having to explore a huge state space, which in turn is greatly dependent on the search strategy used to explore the state space. The exploration procedure used has a significant direct impact on the overall effectiveness of the verification efforts.

Directed model checking techniques [10,11] try to direct the state space search to avoid the potential blowup faced by uninformed model checking techniques. Various heuristic strategies [9, 13, 15] have been proposed for searching the state space efficiently. A^* and greedy search are the most prevalent approaches that have been taken to guide the exploration. Saturating the strongly connected components first [3] is also proposed for efficient search. But all these searching techniques aim to optimize only one objective, e.g. the counterexample length or the convergence of fixpoint computation. Heuristics directed towards quickly finding the *error* state may not scale very well in the absence of error states and vice versa.

We present in this paper a novel subsumer guided symbolic state space exploration strategy that supports both efficient counterexample discovery and quick convergence of reachability. The subsumer-first heuristic leverages the results of partial fixpoint checks (*subsumes* checks) performed during the symbolic state space exploration. The search strategy saves in the abstraction computation phase across the refinement iterations. It also discovers thicker counterexamples (in terms of number of states), which in turn usually provide better predicates and helps reach the fixpoint faster.

We present an application of the *subsumer-first* heuristic for improving the efficiency of abstraction-based software verification within the framework of CEGAR [6] based approaches. The empirical evaluation on industrial benchmarks of this search strategy implemented in ARMC [18], a predicate abstraction based model checker, presents its efficacy. We observe significant efficiency improvements (median of 40% reduction, 1.6 speedup) on difficult benchmarks from the transportation domain. The subsumer guided search on an average leads to a significant reduction in the number of abstraction entailment queries, total number of states explored and the total time taken. In some cases it is orders of magnitude faster than the breadth-first strategy successfully employed previously.

2 Motivating Example

In this section, we present the rationale behind our approach of subsumer guided search with the help of a motivating example. In explicit-state model checking, the nodes in the abstract reachability tree correspond to the states of the program. Each state is a combination of a program counter location and a set of predicates true in that state. The predicates are boolean expressions over the program variables, which are either true or false in any given state.

We say a state s_1 subsumes a state s_2 iff

- $pc(s_1) = pc(s_2)$, s_1 and s_2 share the same program counter location.
- $preds(s_1) \subset preds(s_2)$, the set of predicates true in state s_1 are a strict subset of the ones true in state s_2.

The set of nodes reachable from state s_2 is also a strict subset of the set of nodes reachable from s_1.

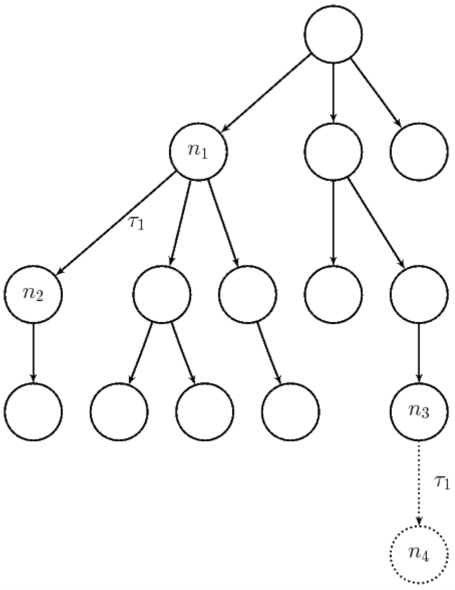

Fig. 1. Abstract Reachability Tree

Consider the abstract reachability tree in Figure 1. Assume we have built the abstract tree in the breadth-first manner. Suppose the newly computed node n_3 subsumes node n_1. All the nodes that are reachable from node n_1 are also reachable from n_3. Now there are a few alternatives to resume the search from this point onwards.

One possible option is to continue the search in breadth-first manner, i.e. append node n_3 at the end of the queue of nodes to be expanded and continue. The problem with this approach is that potentially a significant amount of time may be wasted in exploring states inside the subtree rooted at n_1 which are anyways going to be explored from state n_3 after sometime. This approach suffers from large redundant abstract state computation.

Another option might be to delete all the nodes in the subtree rooted at n_1 and resume the search in the breadth-first manner. Since all the nodes reachable from n_1 are guaranteed to be reached from n_3, it is a sound step to perform. The problem with this approach is that we may lose on already computed states (nodes inside subtree rooted at n_1) and would need to recompute them again. If the number of nodes in the subtree to be deleted is large, we potentially lose significantly in the re-computation of all the pruned nodes.

Our approach instead of deleting the subtree rooted at node n_1, keeps the subtree intact. The node n_1 is pruned from the tree. The node n_3 is scheduled for expansion in the queue $\mathcal{Q}$ before the nodes in the subtree rooted at n_1 which are present in $\mathcal{Q}$. From the monotonicity of the $\mathsf{post}^{\#}$ operator, the successor node of n_3 on transition τ_1, $n_4 = \mathsf{post}^{\#}(n_3, \tau_1)$ is guaranteed to be at least as large as n_2. If the successor node n_4 is larger than n_2, then the same algorithm applies recursively: the node n_2 is pruned and n_4 is scheduled ahead of the nodes in the subtree rooted at node n_2. But if the successor node n_4 is equal to node n_2, then we prune the node n_4. Now we can reuse all of the previously computed subtree of n_2. So if at some point, the newly computed successor node becomes exactly equal to some previously computed node, we can stop searching the space further from that node and reuse the whole of previously computed subtree. It is easy to envision the cases where large savings can be achieved. The child and parent pointers need to be appropriately modified and maintained during the complete iteration.

Another big advantage of this approach is that since bigger abstract states get priority for expansion first, we search for error states in a much larger state space. The counterexamples involving the larger abstract states are given priority over the ones involving smaller states. Therefore the counterexamples we get in the abstract-check-refine loop are much thicker in terms of number of concrete states contained inside them. Another way to imagine about thicker counterexamples can be to consider them as a combination of multiple counterexamples. While refining multiple counterexamples simultaneously, the chances of getting better set of predicates become much larger which significantly helps in reaching the fixpoint faster. Refining multiple counterexamples simultaneously has been shown empirically to perform well in practice [16], in accordance with our observations.

3 Preliminaries

This section provides basic definitions together with a brief description of predicate abstraction [2] and refinement-based approach for proving safety properties (which can be skipped by experts in predicate abstraction).

Programs and Computations A program $P = (\Sigma, \mathcal{T}, s_I, s_E)$ consists of

- Σ : a set of *program states*,
- $\mathcal{T}$: a finite set of *program transitions* such that each transition $\tau \in \mathcal{T}$ is associated with a binary *transition relation* $\rho_\tau \subseteq \Sigma \times \Sigma$,
- s_I : an *initial* state, $s_I \in \Sigma$,
- s_E : an *error* state, $s_E \in \Sigma$.

Our exposition does not assume any further state structure. Though, for the sake of concreteness we point out that usually a program state $s \in \Sigma$ is represented by a valuation of program variables, and program transitions $\mathcal{T}$ correspond to program statements as written in a programming language.

A program *computation* $\sigma = s_1, s_2, \ldots$ is a sequence of program states that starts at the initial state, i.e., $s_1 = s_I$, and each pair of consecutive states (s_i, s_{i+1}) is related by some program transition $\tau \in \mathcal{T}$, i.e., $(s_i, s_{i+1}) \in \rho_\tau$. A program state s is *reachable* if it appears in some program computation. Let Reach be the set of all reachable states. The program P is *safe* if the error state s_E is not reachable in any computation, i.e., if $s_E \notin \mathsf{Reach}$.

A program *path* π is a sequence of program transitions. We write $\pi \cdot \tau$ to denote an extension of a path π by a transition τ. A program path $\pi = \tau_1, \dots, \tau_n$ is *feasible* if it induces a computation, i.e., if there is a sequence of states $s_1, \dots, s_{n+1}$ such that $(s_i, s_{i+1}) \in \rho_{\tau_i}$ for each $1 \leq i \leq n$.

Predicate Abstraction and Refinement We can verify program safety by computing the set of reachable program states and checking if it contains the error states. The set of reachable program states Reach can be constructed incrementally by iterating the "one-step" reachability operator post that maps a set of states $S \subseteq \Sigma$ into a set of immediate successors. Formally, for each transition $\tau \in \mathcal{T}$ we define

$$\mathsf{post}(\tau, S) \quad = \quad \{s' \in \Sigma \mid \exists s \in S : \ (s, s') \in \rho_\tau\} \ ,$$

and then extend canonically to aggregate over all program transitions

$$\mathsf{post}(S) \quad = \quad \bigcup_{\tau \in \mathcal{T}} \mathsf{post}(\tau, S) \ .$$

Then, the set Reach of all reachable states consists of the states reachable from the initial state s_I by any finite number of post-applications:

$$\begin{aligned} \mathsf{Reach} \quad &= \quad \bigcup_{i \geq 0} \mathsf{post}^i(\{s_I\}) \\ &= \quad \mathsf{lfp}(\mathsf{post}, \{s_I\}) \ . \end{aligned}$$

The set Reach of reachable states is generally not computable, since the number of iterations required to reach the fixpoint can be very large or infinite. For practical safety verification, we observe that any sufficiently precise *over-approximation* of Reach can be used to check program safety: if the error state is not present in the approximation then it is not reachable. Thus, by adjusting the precision of over-approximation we can achieve desired practical effectiveness of the iterative reachability computation.

The framework of abstract interpretation formalizes the approximation-based approach by defining the effect of over-approximation using an *abstraction* function α as a basic building block [8]. The abstraction function α maps a set of program states to its over-approximation. Formally, we require $S \subseteq \alpha(S)$ for any set of state S, and $\alpha(S) \subseteq \alpha(T)$ for any set of states T such that $S \subseteq T$. We apply abstraction after each application of the "one-step" operator post

$$\mathsf{post}^\#(S) \quad = \quad \alpha(\mathsf{post}(S)) \ ,$$

and then obtain the desired over-approximation of the reachable states

$$\mathsf{Reach}^\# \quad = \quad \mathsf{lfp}(\mathsf{post}^\#, \alpha(\{s_I\})) \ .$$

The main challenge in applying the abstract interpretation framework amounts to choosing the abstraction function α that is precise enough *and* can be efficiently computed in practice.

Predicate abstraction is a prominent approach to automate the construction of α using automated theorem prover [12]. It requires a finite set of *predicates* $Preds = \{P_1, \ldots, P_n\}$, where each predicate P_i represents a set of program states $P_i \subseteq \Sigma$. An over-approximation of the state is constructed from $Preds$. Automated refinement techniques are used to determine the set of predicates that define the abstraction function.

4 Algorithms

We present our algorithm to combine the subsumer-first search strategy to the CEGAR framework. In each iteration of the abstract-check-refine loop, the method abstractCheck is called with queue $\mathcal{Q}$ initially containing only the *start* state. If no *error* state is reached and a fixpoint is reached, i.e. $\mathcal{Q}$ becomes empty, the program is declared SAFE. Otherwise if an *error* state is encountered, the abstract counterexample is checked whether it is a valid concrete counterexample. If yes, the counterexample is returned as a concrete counterexample presenting the violation of the safety property. Otherwise the spurious counterexample is refined by adding new predicates which refute its concrete existence and again a new iteration of abstractCheck is initiated with the newer set of predicates.

4.1 abstractCheck

The abstractCheck algorithm in Figure 2 keeps expanding the states until either the queue becomes empty or some *error* state is reached(line 1). It dequeues the first element n in the queue $\mathcal{Q}$(line 2). Then for all possible *enabled* transitions τ, i.e. all transitions which are valid to make from n, the next state m is computed using $\mathsf{post}^{\#}$(line 4). SubsumedSubtree contains the list of nodes that are present in the subtree rooted at node p which is subsumed by m. In line 7, it is computed for all such p and then their union is taken. The child pointer from the parent of p, p_p is modified to now point to m(lines 8-9). Also, in line 10 the child pointers of p are accordingly moved to now point from m.

If the node m is subsumed by some other node p in the tree, then the node m is not scheduled in $\mathcal{Q}$. It should be noted that it can never be the case that m subsumes some node in the tree and at the same time is subsumed by some other node in the tree. We maintain the invariance that no node present in the tree is subsumed by any other node present in the tree.

If no node subsumes m, we schedule m in front of all the nodes that are present in both the SubsumedSubtree list and the queue $\mathcal{Q}$(line 24). If no nodes present in the SubsumedSubtree are present in $\mathcal{Q}$ or the SubsumedSubtree is empty, m is appended at the end of $\mathcal{Q}$ (line 22).

```
abstractCheck(IterId)
1. while $\mathcal{Q} \neq \emptyset$ do
2.     $n$ := dequeue($\mathcal{Q}$)
3.     forall enabled($n, \tau$)
4.         $m$ := $post^{\#}(n, \tau)$
5.         SubsumedSubtree := $\emptyset$
6.         forall subsumed($p, m$)
7.             SubsumedSubtree $\cup$ := computeSubtree($p$)
8.             AbstractChild $-$ := abstract_child($p_p, \tau_1, p$)
9.             AbstractChild $+$ := abstract_child($p_p, \tau_1, m$)
10.            forall child($q, p$)
11.                AbstractChild $-$ := abstract_child($p, \tau_2, q$)
12.                AbstractChild $+$ := abstract_child($m, \tau_2, q$)
13.         if subsumed($m, p$)
14.             AbstractChild $+$ := abstract_child($n, \tau, p$)
15.         else
16.             AbstractChild $+$ := abstract_child($n, \tau, m$)
17.             if SubsumedSubtree == $\emptyset$
18.                 enqueue($\mathcal{Q}, m$)
19.             else
20.                 SubsumerPosition := computePosition($\mathcal{Q}$, SubsumedSubtree)
21.                 if SubsumerPosition == $-1$
22.                     enqueue($\mathcal{Q}, m$)
23.                 else
24.                     insert($\mathcal{Q}$, SubsumerPosition, $m$)

computePosition($\mathcal{Q}$, SubsumedSubtree)
1. for $i$ in range(len($\mathcal{Q}$))
2.         n=$\mathcal{Q}(i)$
3.         if $n \in$ SubsumedSubtree
4.             return $i$
5. return $-1$

computeSubtree(SId)
1. SubTree := {SId}
2. until SubTree converges
3.     forall $n \in$ SubTree
4.         forall child($m, n$)
5.             if $m \notin$ SubTree
6.                 SubTree $\cup$ := $m$
7. return SubTree
```

Fig. 2. Subsumer-first Algorithm

4.2 computePosition

The function computePosition returns the index of the first node in the queue $\mathcal{Q}$ that is also present in the SubsumedSubtree. It traverses over all the nodes in $\mathcal{Q}$ from the beginning (line 1) and returns the least index i, such that $\mathcal{Q}(i) \in$ SubsumedSubtree (line 4). If no such i is present, it returns -1.

4.3 computeSubtree

The function computeSubtree computes the nodes in the subtree rooted at the node SId using the *abstract_child* relation : $n \times \tau \rightarrow n$. Since there might be loops following the *abstract_child* pointers, computeSubtree keeps adding nodes to the subTree list until it converges (lines 2-6). The cycles (by following the child pointers) might be introduced while manipulating the child pointers when handling the subsumer and subsumed nodes.

5 Experiments

We present the results of subsumer-first search heuristic and compare it with the breadth-first search strategy (both implemented in the ARMC model checker) on a set of benchmarks, some of which come from train control systems. We evaluated the heuristic on some of the most difficult benchmarks from the transportation domain. To get more coverage, we added some smaller benchmarks as well to the evaluation set. The results for computing the fixpoint given a fixed abstraction sufficient for verifying the safety property are presented in Table 1 and the results for the complete abstraction-refinement loop are presented in Table 2. The first two rows in the tables present the performance of breadth-first search and subsumer-first search respectively. The third row presents the percentage decrease in the running time, entailment queries and the number of states. The experiments were run on a dual core 3.16 GHz Intel Pentium processor machine with 2 GB of RAM.

Table 1 presents the experiment results on the last iteration computation of ARMC given a sufficient set of predicates at the start of search to refute all spurious counterexamples and verify the safety property. This provides us a notion of reaching the fixpoint faster for a fixed abstraction. The last 3 columns of the table present the relative sizes of the benchmarks in terms of number of variables, transitions and locations in their control flow graphs. From the results, it is quite evident that subsumer-first strategy significantly outperforms the breadth-first search strategy consistently across all benchmarks. The subusmer-first strategy takes lesser time (mean decrease 31.8%, median 33.3%), produces lesser number of entailment queries (mean 35.8%, median 42.1%) and explores lesser number of states (mean 32.9%, median 35.5%). The subsumer-first strategy was on average 1.68 times faster, and the median speedup was 1.54.

The results in Table 2 presents the results for the complete abstraction-refinement loop starting with an empty initial abstraction. The results show that the subsumer-first strategy mostly outperforms the breadth-first startegy in terms of running time(mean 31%, median 46.9%), number of entailment queries of theorem prover (mean 30.5%, median 20.8%) and the total number of states explored (mean 15.5%, median 29.7%). The subsumer-first strategy was 2.9 times faster on average (with median of 1.88) than the breadth-first strategy on these benchmarks. Since, the subsumer-first approach refines thicker counterexamples, it usually finds more number of predicates but still on average reaches the fixpoint faster. In some cases, it might happen that while refining a shorter, thinner and local counterexample the breadth-first strategy might get lucky and the new predicates discovered may prune a large state space in the next iteration. But in general both from our experience from the experiments and as presented in [16], refining more number of counterexamples simultaneously provides a higher chance of discovering better predicates and faster fixpoint arrival.

Odometrys1ub and model-test19 seems to belong to those exception cases where the thicker counterexamples take some time to find the right set of predicates whereas the thinner counterexample finds a good set of predicates luckily. But even then the subsumer-first strategy remarkably generated lesser number of entailment queries on the odometrys1ub benchmark.

6 Discussion and Future Work

We present in this paper a useful heuristic which addresses the issues of efficient counterexample discovery and faster convergence of reachability computation simultaneously. The subsumer-first heuristic can also be thought of as a combination of breadth-first search for exploration with depth-first search in terms of subsumer nodes. It tries to get benefits of both approaches and produces short and thick counterexamples. Refining thicker counterexamples gives a better chance to get good predicates after refinement. These predicates can potentially rule out lot many spurious paths in later iterations. The optimality of the strategy can not be guaranteed as sometimes some lucky predicates can be discovered from other counterexamples which may prune the search space far more. However, in practice the subsumer-first strategy performs usually well.

This heuristic can easily be integrated with other heuristic state space exploration strategies to achieve even more savings, e.g in case of saturating the strongly connected components(SCC) first heuristic, the subsumer-first heuristic can be used inside a particular SCC during its saturation. We present in this paper one application of this heuristic in predicate abstraction based model checking. Its adaptation for integration with lazy abstraction and partial order reduction techniques would certainly be an interesting next step.

One interesting case occurs when a node m subsumes a node p, we change the child pointer of parent of p_p to m. But now if some node n subsumes p_p, it may be the case that there are some states in the subtree of m which are not reachable from n but still we schedule n before all of m's subtree. Using more fine-grained information about the transition system might help in predicting even better positions for scheduling the new nodes in the queue $\mathcal{Q}$.

Benchmark	time	# queries	# states	# vars	# trans	# locs	speedup
odometrys4lb[a]	787m	18.3M	11486	15	3337	150	1.59
odometrys4lb -sub-first	**494m**	**9.4M**	**6207**				
	37.2%	48.6%	45.9%				
odometryls2lb[a]	227m	7.9m	8184	16	6127	214	2.18
odometryls2lb -sub-first	**104m**	**3.8m**	**3886**				
	54.2%	51.9%	52.5%				
odometryls1ub[a]	243m	13.3M	12762	16	6127	214	3.19
odometryls1ub -sub-first	**76m**	**4.5M**	**4624**				
	68.7%	66.2%	63.8%				
odometrys1ub	34m	1.6M	2073	15	3337	150	1.79
odometrys1ub -sub-first	**19m**	**0.7M**	**1033**				
	44.1%	56.2%	50.2%				
timing	29m	0.4M	3425	47	99093	4954	1.0
timing -sub-first	29m	**0.39M**	**3378**				
	0%	2.5%	1.4%				
gasburner	17m	3.5M	3309	19	3124	152	1.89
gasburner -sub-first	**9m**	**1.7M**	**1791**				
	47%	51.4%	45.9%				
odometryls1lb	12m	0.8M	1439	16	6127	214	3.0
odometryls1lb -sub-first	**4m**	**0.3M**	**632**				
	66.7%	62.5%	56%				
rtalltcs	4m	2.5M	1789	20	18757	122	2.0
rtalltcs -sub-first	**2m**	**1M**	**796**				
	50%	60%	55.5%				
odometrys1lb	2m	0.2M	681	15	3337	150	2.0
odometrys1lb -sub-first	**1m**	**0.1M**	**425**				
	50%	50%	37.6%				
triple2	2m	0.77M	610	3	8	3	1.0
triple2 -sub-first	**2m**	**0.70M**	**520**				
	0%	9%	14.8%				
odometry	1m	0.14M	246	15	437	28	1.5
ododmetry -sub-first	**40s**	**0.09M**	**193**				
	33.3%	35.7%	21.5%				
bakery3	11s	0.25M	1311	9	31	3	1.1
bakery3 -sub-first	**10s**	**0.19M**	**986**				
	9%	24%	24.8%				
model-test01	32s	0.18M	1578	16	110	36	1.1
model-test01 -sub-first	**29s**	**0.18M**	**1565**				
	9.4%	0%	0.8%				
model-test07	39s	0.25M	1998	16	124	40	1.05
model-test07 -sub-first	**37s**	**0.24M**	**1902**				
	5.1%	4%	4.8%				
model-test13	**2m**	0.9M	5766	16	110	36	1.0
model-test13 -sub-first	2m	**0.75M**	**4791**				
	0%	16.7%	16.9%				
model-test19	3m	0.9M	5256	16	124	40	1.5
model-test19 -sub-first	**2m**	**0.6M**	**3499**				
	33.3%	33.3%	33.4%				

Table 1. Experiments with last iteration of ARMC with fixed set of predicates, fixpoint computation

[a] These benchmarks are not present in Table 2 as they either TIMED-OUT (> 1500m) or RESOURCE-ERROR due to memory requirements

Benchmark	time	# queries	# iter	# preds	# states	speedup
odometry	109m	9.3M	65	218	680	13.6
odometry -sub-first	**8m**	**1.6M**	**37**	**153**	**295**	
	92.7%	82.8%			56.6%	
odometryls1lb	60m	7.1M	32	**97**	1439	2.07
odometryls1lb -sub-first	**29m**	**3M**	**29**	102	**539**	
	51.7%	57.7%			62.5%	
triple2	13m	6.5M	65	254	519	6.50
triple2 -sub-first	**2m**	**2.1M**	**45**	**219**	**248**	
	84.6%	67.7%			52.2%	
odometrys1lb	9m	1.1M	**20**	**72**	681	1.0
odometrys1lb -sub-first	**9m**	**1.0M**	22	83	**345**	
	0%	9%			49.3%	
odometrys1ub	**195m**	14.4M	37	**157**	**2073**	0.59
odoemtrys1ub -sub-first	329m	**11.4M**	**33**	257	2379	
	-68.7%	20.8%			-14.8%	
gasburner	175m	48.9M	64	**198**	3309	1.88
gasburner -sub-first	**93m**	**17.3M**	**61**	220	**1604**	
	46.9%	64.6%			51.5%	
timing	51m	1M	14	14	3425	1.04
timing -sub-first	**49m**	**1M**	14	14	**3378**	
	3.9%	0%			1.4%	
rtalltcs	38m	27M	**30**	**56**	1789	1.03
rtalltcs -sub-first	**37m**	**25.3M**	40	74	**1258**	
	2.6%	6.3%			29.7%	
bakery3	2m	2.6M	**34**	67	1419	3.75
bakery3 -sub-first	**32s**	**0.9M**	36	**58**	**885**	
	73.3%	65.4%			37.6%	
model-test01	4m	1.7M	58	115	**1207**	2.0
model-test01 -sub-first	**2m**	**1.5M**	**54**	**100**	1565	
	50%	11.8%			-29.7%	
model-test07	5m	2.4M	58	115	**1372**	1.67
model-test07 -sub-first	**3m**	**2.2M**	**56**	**104**	1902	
	40%	8.3%			-38.6%	
model-test13	17m	6.6M	63	140	**4708**	1.89
model-test13 -sub-first	**9m**	**5.2M**	**61**	**136**	4791	
	47%	21.2%			-1.8%	
model-test19	**19m**	**7.7M**	62	137	**5256**	0.83
model-test19 -sub-first	23m	9.2M	**59**	**135**	8135	
	-21%	-19.5%			-54.8%	

Table 2. Experiments with full ARMC abstraction-refinement iterations

References

1. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213. ACM, 2001.
2. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
5. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395. IEEE Computer Society, 2003.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169. Springer, 2000.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
9. K. Dräger, B. Finkbeiner, and A. Podelski. Directed model checking with distance-preserving abstractions. In *SPIN*, pages 19–34, 2006.
10. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2-3):247–267, 2004.
11. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *SPIN*, pages 57–79, 2001.
12. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83. Springer, 1997.
13. A. Groce and W. Visser. Heuristic model checking for java programs. In *SPIN*, pages 242–245, 2002.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
15. S. Kupferschmid, K. Dräger, J. Hoffmann, B. Finkbeiner, H. Dierks, A. Podelski, and G. Behrmann. Uppaal/DMC- Abstraction-based heuristics for directed model checking. In *TACAS*, pages 679–682, 2007.
16. K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.
17. D. Peled. All from one, one for all: on model checking using representatives. In *CAV*, pages 409–423, 1993.
18. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, pages 245–259, 2007.