ACM Copyright Notice

Published in: ***Proceedings of IEEE/ACM international Conference on Automated Software Engineering (ASE'10),*** September 2010

"Search-carrying code"

Cite as:

Ali Taleghani and Joanne M. Atlee. 2010. Search-carrying code. In Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE '10). ACM, New York, NY, USA, 367-376. DOI:

BibTex:

inproceedings{Taleghani:2010:SC:1858996.1859079,
 author = {Taleghani, Ali and Atlee, Joanne M.},
 title = {Search-carrying Code},
 booktitle = {Proceedings of the IEEE/ACM International Conference on Automated Software Engineering},
 series = {ASE '10},
 year = {2010},
 pages = {367--376}
}

DOI: https://doi.org/10.1145/1858996.1859079

Search-Carrying Code

Ali Taleghani
David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada
ataleghani@uwaterloo.ca

Joanne M. Atlee
David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada
jmatlee@uwaterloo.ca

ABSTRACT

In this paper, we introduce a model-checking-based certification technique called *search-carrying code* (SCC). SCC is an adaptation of the principles of proof-carrying code, in which program certification is reduced to checking a provided safety proof. In SCC, program certification is an efficient re-examination of a program's state space. A code producer, who offers a program for use, provides a *search script* that encodes a search of the program's state space. A code consumer, who wants to certify that the program fits her needs, uses the search script to direct how a model checker searches the program's state space.

Basic SCC achieves slight reductions in certification time, but it can be optimized in two important ways. (1) When a program comes from a trusted source, SCC certification can forgo authenticating the provided search script and instead optimize for speed of certification. (2) The search script can be partitioned into multiple partial certification tasks of roughly equal size, which can be performed in parallel. Using parallel model checking, we reduce the certification times by a factor of up to n, for n processors. When certifying a program from a trusted source, we reduce the certification times by a factor of up to $5n$, for n processors.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; K.7.3 [**The Computing Profession**]: Testing, Certification, and Licensing

General Terms

Verification

Keywords

Software Certification, Model Checking

1. INTRODUCTION

Component-based software engineering promises rapid development and extension of systems through the assembly of pre-existing and third-party components. However, before a code consumer uses an acquired component in her software product, she must *certify* that the component fits her needs. If a component comes from a non-trusted source, then the code consumer will want to check that the component is safe and performs as advertised. Even if a component comes from a trusted source, the code consumer may still want to assess whether the component exhibits additional desired properties, beyond those claimed by the code producer.

Proof-carrying code (PCC) [18] has been advanced as a means to ease certification. The premise of PCC is that proof checking is faster and simpler than theorem proving. In PCC, the code producer verifies via theorem proving that his program satisfies a set of predefined safety properties, and provides as evidence a *safety proof*. The code consumer certifies the program by checking the validity of the accompanying safety proof against the code. However, PCC certification can only (re)verify the properties that are substantiated by the safety proof. Also, because reasoning about general properties of programs is complex, PCC has so far focused on program-independent security properties (e.g., memory safety, type safety, resource bounds).

In this paper, we introduce the concept of **search-carrying code (SCC)**, in which we apply the goals of PCC to programs and properties that can be model checked. Specifically, we explore whether information collected during successful *verification* of a program (via explicit-state model checking) could be used to ease subsequent *certification* (via explicit-state model checking) of the same program. Our approach focuses on paths through a program's reachability graph. A **reachability graph** is a graphical representation of a program's set of possible executions, in terms of execution states (i.e., program counter, variable values, etc.) and transitions (i.e., program statements) between execution states. As a code producer's model checker explores a program's reachability graph, it records its search path as a *search script*, which effectively acts as a *certificate* of the verification. The code consumer's model checker takes the script as input and uses it to speed up the task of re-examining the program.

Basic SCC achieves slight reductions in certification times. Both traditional model checking and SCC certification check properties in every state of a program's state space. However, whereas traditional model checking must *ascertain* a program's reachable state space, SCC certification can simply *confirm* that the provided script truthfully reflects the state space. We take advantage of this distinction and modify the data structures that the model checker uses to keep track of visited states, thereby realizing a small, up to 5% speed-up in certification time.

In the special case where a code consumer trusts the code producer but wants to check additional properties, and the veracity of the search script is not in question, SCC certification need not au-

ASE'10, September 20–24, 2010, Antwerp, Belgium.

thenticate the script and can instead be optimized for speed. In **trustful SCC**, the provided search script encodes a **perfect search** of a program's state space that visits each program state exactly once, avoiding paths that lead back to already visited states. The resultant speed-up in certification time depends on the structure of a program's reachability graph: in particular, it depends on the ratio of the number of transitions to the number of states.

More significantly, SCC certification can be parallelized more effectively than traditional model-checking tasks can. The main challenge in parallel model checking is balancing the workload among parallel processors. However, in SCC, the search task is known in advance and is encoded in the search script. The search script can be partitioned into multiple tasks, each covering a cohesive region of the search space that a processor can explore independently, thereby avoiding the inter-processor communication that is usually necessary to balance workloads dynamically.

The contributions described in this paper are as follows:

- We propose a model-checking-based certification technique called search-carrying code (SCC), in which a code producer provides a search script that a code consumer uses to direct the search of its model checker. We discuss how the technique verifies the veracity of a provided search script. We also show how SCC certification can be optimized in cases where the source of a component is trusted and the goal is simply to check additional properties.

- We present a greedy algorithm to partition an SCC search script into several scripts, which can direct several partial certifications of a program in parallel.

- We discuss requirements for the code producer's and consumer's model checkers. While we have embedded our SCC algorithm into Java Pathfinder (JPF) [27], we expect that SCC can be applied to other explicit-state model checkers.

- We evaluate the performance of SCC on a suite of Java programs. By combining SCC and parallel model checking, we achieve an average speed-up of up to n using n processors. When certifying a program from a trusted source and employing parallel model checking, we report an average speed-up of up to $6n$ using n processors.

This paper is organized as follows. In Section 2, we describe the basic algorithms and technologies needed to support SCC-based certification, including the special case of trustful SCC. Section 3 describes how to parallelize SCC to take advantage of parallel model checking and presents the results of our evaluation. Section 4 discusses outstanding issues and known limitations of SCC, and Section 5 presents related work. We conclude with Section 6.

2. SEARCH-CARRYING CODE

Software model checking exhaustively examines a program's state space, checking conformance with desired properties. During verification of a program, the emphasis is on finding bugs and ultimately showing that a program is error-free. For certification, the goal is to confirm that a program behaves as advertised, and possibly to check for additional non-advertised properties.

Ideally, it is faster to certify a program than it was to verify it in the first place. The goal of **search-carrying code (SCC)** is to use information collected during model-checking-based verification of a program to speed up model-checking-based certification of the program. A *code producer's* model checker performs a traditional exhaustive search and verification of a program's state space. At the same time, it constructs a *search script* that represents its search of the state space. The search script is a sequence of transitions (i.e., program statements) and their resultant execution state IDs (i.e., not state encodings); the script corresponds to a depth-first search of the program's reachability graph. During certification, a *code consumer's* model checker uses the provided search script to direct its search of the program's state space and *certifies* that the program satisfies its advertised properties, plus any additional desired properties. In general, SCC can be used to certify safety properties of programs, where the properties are expressed as program invariants or assertions, and to confirm absence of deadlocks. We discuss properties in more detail in Section 4.

Search-carrying code possesses many of the same benefits of proof-carrying code. First, the burden of verifying the program is borne by the code producer, whereas the code consumer simply re-checks the program. Second, the search script can be generated automatically by the code producer's model checker, as we show in the next section. Third, SCC certification can detect accidental or malicious deviations between a program and associated search script. There are three types of deviation: (1) the script includes nonexistent transitions, (2) the script omits a transition, or (3) the script incorrectly claims that a transition leads to an already-visited state. The first two types of deviation are easily detected: in the first case, the program has no program statement that matches the script's transition instruction; and in the second case, the script says to end the examination of a state before all of its transitions have been explored. In both cases, the model checker detects the discrepancy and the certification fails. The third type of deviation is more menacing because, if undetected, it results in a partial search of the program's state space: the mislabelled state is deemed to have already been visited, so the model checker does not test the state and does not explore the state space that is reachable from it. To detect this third type of deviation, SCC certification must–in addition to visiting and testing all of the program's states–explore all of the transitions emanating from these states, to see if they lead to new unvisited states. Thus, the search script encodes the program's entire reachability graph.

Given that SCC certification entails re-exploring a program's entire reachability graph, it might seem surprising that SCC achieves any savings at all. As will be seen, small savings come from being able to *confirm* the script's encoding of the reachability graph, rather than *determining* the reachability graph, as is the case in traditional model checking. More significant savings come from parallelizing SCC certification. We describe parallel SCC in Section 3.

In the special case of *trustful* SCC certification, the code producer and the verification results are trusted. However, the code consumer wants to certify additional properties of the program, and the code producer is unable or unwilling to check these. Because the code producer is trusted, the code consumer may choose not to check the veracity of the script. As a result, we can aggressively optimize the certification task for speed. We describe trustful certification in Section 2.3.

2.1 Search Script Construction

An SCC search script records all transitions in a program's reachability graph and the state ID of each transition's destination state. Consider Figure 1, which depicts the reachability graph of an artificially simple program. Transition labels abstractly represent the program statements being executed. The script for this program is:

Trans instr:	–	t_1	t_1	B	t_2	t_1	B	t_2	t_1	B	B	t_3	t_1	B	B	B	B	t_2	B
State ID:	S1	S2	S1	S2	S3	S1	S3	S4	S2	S4	S3	S5	S4	S5	S3	S2	S1	S4	S1

where the t_is encode program statements (e.g., the byte code instruction, or a combination of byte code and thread ID) and *B*s rep-

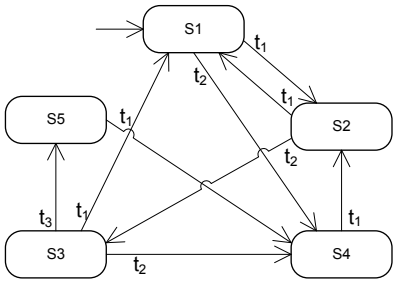

Figure 1: Sample reachability graph of a program

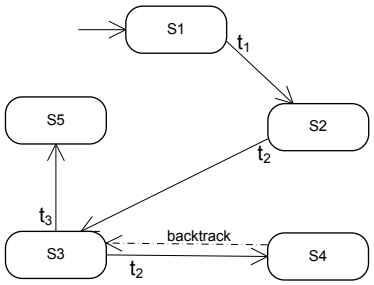

Figure 2: Perfect search of a program's state space

resent backtracks. Reading the script from left to right, the search starts in the program state labelled $S1$; it explores the program statement represented by transition t_1, which results in a program state labelled $S2$; and so on.

SCC uses encodings of program statements in the script rather than transition IDs, so that different model checkers interpret the script in the same way, and search the state space in the same order. Below is an example partial script in which transition instructions are expressed as byte-code instructions:

Trans instr:	–	$aload\_0$	$aload\_1$	B	$getfield\#5$
State ID:	S1	S2	S1	S2	S3

For the remainder of this paper, we will abstract instructions to transition IDs for the clarity of presentation.

To minimize the size of the search script, destination states are represented by state ID rather than by explicit state encodings. State IDs are unique identifiers that the code producer's model checker creates and assigns to states during the course of the verification search, starting with identifier $S1$ and incrementing by 1 each time a new state is discovered. A transition that leads to a new state is referred to as a **productive transition**; its destination state has an ID that is larger than the largest state ID encountered so far.

2.2 Search Script Usage

During SCC certification, the model checker follows the instructions given in the provided search script, checking properties and authenticating the search script on-the-fly. Any discrepancy causes the certification to fail. Discrepancies are identified in three ways

1. The script instructs the model checker to explore a nonexistent transition (i.e., a nonexistent program statement).

2. The script instructs the model checker to backtrack when the state still has unexplored transitions.

3. The script asserts that two transitions have the same destination state (with the same state ID), but the model checker determines that the two program states have different fingerprints. To facilitate this check, the model checker maintains during certification a mapping FP from state IDs to fingerprints. A each new state is reached (in order of state ID), its ID and fingerprint are entered into FP. When a state is revisited (indicated in the script by a destination state whose ID that is lower than the highest ID), its previous fingerprint is retrieved from FP and compared against the new fingerprint. A mismatch is a discrepancy. Our map FP is slightly more efficient than a hash table of visited states because it grows more gracefully.

2.3 Trustful Certification

In cases where a program comes from a trusted source and the code consumer trusts the results of the code producer's verification, SCC can still be useful, especially to check additional properties. Perhaps the code consumer found the program in a trusted software repository and is interested in using the program, but only if some additional properties are met. The code producer might not be available or willing to perform additional checks.

When the code consumer trusts the source of the program, he might also trust the veracity of the search script. If so, the certification need only examine the program's states, to test properties. It need not explore all of the transitions in the program's reachability graph, checking whether any reachable state has been missed. To see the difference, consider again the reachability graph in Figure 1. An exhaustive search of the graph explores all nine transitions, visiting the same states multiple times. In contrast, a **perfect search** traverses a spanning tree of a program's state space: it explores only *productive transitions* and visits each state exactly once[1]. Figure 2 depicts a depth-first, perfect search of the graph from Figure 1. Solid lines represent productive transitions and dashed lines represent backtracks to parent states. The corresponding search script is:

Trans instr: $|t_1|t_2|t_2|\mathrm{B}|t_3|$

The script does not record the transitions' target state IDs because trustful certification does not check the veracity of the script.

Trustful SCC effects a perfect search of a program's state space. The code producer provides a program and matching trustful search script. During certification, the code consumer's model checker uses the search script to direct its search of the program's state space. Neither state fingerprints nor hash-tables of visited states are created or maintained, resulting in additional speedup.

2.4 Evaluation of SCC

We implemented SCC certification in Java Pathfinder (JPF) [17]. JPF is an explicit-state model checker for Java byte-code programs. We refer to the resulting model checker as **JPF-scc**. For convenience, we implemented SCC verification and certification in the same model checker but, in practice, these tasks might be performed by separate tools. JPF, and our modified variants, employ partial-order reduction and two types of symmetry reduction: (1) states that are identical except for unreferenced objects (i.e., garbage) are considered to be equivalent, and (2) states that are identical except for the order in which classes and objects are loaded are considered to be equivalent. We discuss the compatibility of SCC with various state-space reduction techniques in Section 4.

We evaluated our work on a suite of nine Java programs that have been used in previous empirical studies. Table 1 lists each program including its source, the parameter values that we used (e.g., instantiating 8 dining philosophers), the numbers of invariants and assertions that we checked for each program, the number of states in the reachability graph, the ratio of transitions to states, and the

[1]Backtracking does not constitute "visiting" a state because the work of constructing and testing the state is already done.

Table 1: Java Programs Used for Evaluation

Src	Program	Parameters	Properties #Inv/#Assert	#States	#Trans #States	Time (sec)
[17]	Dining Philosopher (8)	#philosophers	1/2	209014	5.2	220
[23]	Bounded Buffer (5,4,4)	buffersize,#prod,#cons	1/3	786987	7.1	1088
[21]	Nasa KSU Pipeline (4,1)	stagesize,#listeners	3/4	59512	4.1	45
[23]	Nested Monitor (5,4,4)	buffersize,#prod,#cons	6/4	71941	6.9	99
[23]	Pipeline (7)	stagesize	3/3	82011	6.1	101
[23]	RWVSN (4,4)	#readers,#writers	3/4	227116	5.1	245
[23]	Replicated Workers (5,2)	#workers,#items	4/5	710022	5.1	860
[7]	Sleeping Barber (2,4,3)	#barber,#customers,#chairs	4/5	1452194	4.3	1308
[6]	Elevator(5,10,10)	#elevators,#floors,#people	4/8	386032	8.4	1167

Table 2: Results for SCC Verification and Certification

		SCC certification			Trustful SCC certification		
Program (size)	Verification (sec)	Script size(KB)	Certification (sec)	Speed-up	Script size(KB)	Certification (sec)	Speed-up
Dining Philosopher (4KB)	221	104	213	1.03	25	35	6.3
Bounded Buffer (6KB)	1090	329	1036	1.04	74	130	8.4
NASA KSU Pipeline (2KB)	45	91	44	1.02	22	9	5
Nested Monitor (7KB)	100	45	97	1.01	9	14	7.1
Pipeline (5KB)	102	51	99	1.01	11	15	6.7
RWVSN (9KB)	246	102	236	1.03	29	39	6.3
Replicated Workers (15KB)	862	272	819	1.05	72	140	6.1
Sleeping Barber (8KB)	1310	492	1245	1.05	187	245	5.3
Elevator (29KB)	1169	248	1122	1.04	54	129	9

time to model check the program using JPF. We also checked each program for deadlock violations. We ran our experiments on an Intel Pentium 4 3.2GHz machine with 1.5GB of memory, running Windows XP. We ran each experiment ten times and report the average of the ten runs.

We evaluate the utility of SCC on the basis of how long it takes to perform SCC certification, compared to the time it would take a code consumer to reverify a program. Table 2 shows the results for SCC certification using JPF-scc. Column *Verification* shows the time incurred by the code producer to model check the program and create the search script, including the time to write the script to disk. Column *Certification* reports the time incurred by the code consumer to certify each program, including the time to read the search script from disk. Column *Speed-up* shows the speed-up of a certification search compared to a traditional JPF search, as reported in Table 1. For example, the time to certify the Sleeping Barber program and to check the script is 1245 seconds, which is 1.05 times faster than JPF verification of the same program.

The speed-ups of SCC are very small and are mainly due to keeping a map of fingerprints (FP) instead of a hash table. For our set of programs, we report an overhead of 2% to 5% for keeping and maintaining a hash table. SCC saves this small cost. Because of the way that JPF maintains hash tables and resizes tables as needed, the savings increase with the size of the program's state space.

Table 2 also shows the runtime performance of trustful SCC certification. For example, the time to certify the Pipeline program is 15 seconds, which is 6.7 times faster than traditional JPF verification of the same program. The speed-up of trustful SCC certification is proportional to the ratio of the number of transitions to the number of states in the program's reachability graph; this is also the ratio of unproductive to productive transitions. The speed-up is slightly better than the ratio because of the savings from not creating and comparing fingerprints.

2.5 Search Script Size

The feasibility of SCC depends not only on runtime performance but also on the size of the search script. Given a program whose reachability graph has S states and T transitions, SCC will produce a search script containing at most $2T$ instructions (T forward transitions and at most T backtracks) and trustful SCC will produce a search script that has at most $2S$ instructions. Because the number of states and transitions are exponential in the size of the program, one might expect that script size is an issue.

Fortunately, search scripts contain lots of replication (e.g., byte code instructions, backtrack commands), which makes them good candidates for compression. ZIP data compression [22] reduced the sizes of our search scripts by factors of 550 to 650. Table 2 shows the size in KB of the compressed search script for each program, for both SCC and trustful SCC certification. It also shows the size of each program's *class* files along with the program name. The sizes of compressed scripts are on the order of $(T \times 10^{-4})$ KB for SCC and $(S \times 10^{-4})$ KB for trustful SCC. Extrapolating to larger programs, with 100 million states and a billion transitions, the script sizes might be on the order of 100MB for SCC and 10MB for trustful SCC. Such script sizes are large but are manageable.

3. PARALLEL SCC

The promise of parallel model checking [25] is that we can reduce search times by distributing the search among multiple parallel processors. In general, it is difficult to balance a model-checking task evenly among processors because the size of the search space is not known in advance. Attempts to partition the workload in advance (e.g., assigning states to processors based on state information) have resulted in substantial communication overheads, due to the need to transfer new states to their designated processors. Even on a shared-memory architecture, this style of parallel model checking can suffer considerable overhead because processors need to coordinate their shared access to each others' worklists.

In SCC, the certification workload is known in advance, in the form of a search script. As such, it is possible to partition the workload into multiple search tasks of roughly equal size. In the following sections, we first describe how to partition an SCC search script and then explain the optimizations for trustful certification.

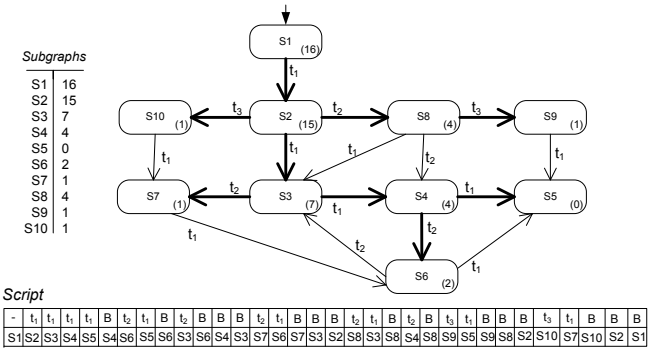

Figure 3: Reachability graph with its script and $Subgraphs$

```
1  Input: Script; /* search script encoding reachability graph */
2  Input: Subgraphs; /* root and size of subgraphs in Script */
3  Input: k; /* number of partition regions to generate */
4  i = 0;
5  while {i < k−1}{
6      Search Subgraphs for p_i whose size is closest to |Script|/(k−i);
7      Remove search script for p_i from Script;
8      Remove all states in p_i from Subgraphs;
9      Update the sizes of subgraphs left in Subgraphs;
10     Compute path to initial state of p_i;
11     i++;
12 }
```

Figure 4: Partitioning algorithm

3.1 Partitioning the State Space

The goal of parallel SCC is to partition the search script into multiple non-overlapping search tasks, each of which covers a contiguous region of the program's reachability graph that can be searched separately.

Let $Script$ be the full search script of a program, as described in Section 2.1, and let $|Script|$ be the size of the script in terms of the number of transitions. Prior to certification, the certifier's model checker constructs a partition $P = \{p_1, ..., p_k\}$ of $Script$ into k search tasks. Each partition region $p_i \in P$ corresponds to a subgraph in the program's reachability graph, *and* to a partial search script $Script_i$ that is a substring of $Script$. A subgraph p_i consists of all states that can be reached via productive transitions from p_i's root and all transitions, productive and unproductive, originating from those states. For example, consider the reachability graph in Figure 3, in which thick edges represent productive transitions. In this example, the subgraph rooted at state $S4$ consists of the states $S4$, $S5$ and $S6$; the subgraph would not include $S3$ because it is reached via an unproductive transition from $S6$. The size of each p_i is the number of transitions emanating from states in p_i.

To facilitate script partitioning, SCC verification generates, along with the search script, a list $Subgraphs$ that records for each program state s the number of transitions in the subgraph rooted at s. Basically, during verification, the model checker performs a depth-first search of the program state space. As each new state s is encountered, an entry indexed by state ID is added to $Subgraphs$. As s's child states are explored and the sizes of their subtrees are computed, the size of s is updated. The $Subgraphs$ list is provided to the code consumer, along with the program and search script. In SCC certification, the size of a $Subgraphs$ list is less than 10% of the size of the search script, and in trustful SCC certification, the size of $Subgraphs$ is less than 20% of the size of the search script.

Figure 3 shows an example reachability graph with its corresponding $Script$ and $Subgraphs$. The $Subgraphs$ table shows for each state s (left column) the size of the subgraph (right column) rooted at s. For example, the subgraph rooted at state $S4$ consists of the states $S4, S5, S6$ and has size four (i.e., the four transitions originating from those states). The value in parentheses below each state identifier in the reachability graph in Figure 3 shows the same information.

Figure 4 gives an overview of our partitioning algorithm. It takes as inputs the search script $Script$ and the $Subgraphs$ list that are provided by the code producer, and the number of partitions k to generate (based on the number of available parallel processors). In the ith iteration, the algorithm searches $Subgraphs$ for a subgraph whose size is closest to $1/k-i$ of the number of transitions not yet assigned to a partition region (line 6); this subgraph becomes a new partition region p_i. Next, the partial search script $Script_i$ for partition region p_i is extracted from $Script$ (line 7). The algorithm also removes all states in p_i from $Subgraphs$ (line 8). We describe both processes in Section 3.1.1. The algorithm then updates the sizes of the remaining subgraphs in $Subgraphs$ (line 9). Note that only the sizes of ancestor states of p_i need be modified, and their sizes are reduced by the size of p_i. We describe how ancestor states are identified in Section 3.1.2. Finally, the algorithm constructs the path from the program's initial state to the initial state of search task $Script_i$ (line 10). We discuss the rationale and process for constructing this initialization path in Section 3.1.2.

Figure 6 shows the result after one iteration of our partitioning algorithm as applied to Figure 3, for $k = 3$. The subgraph p_1, rooted at state $S4$, is selected for extraction and its subscript is removed from $Script$ (the dark line in $Script$ shows from where the subscript was extracted). The states in p_1 have been removed from $Subgraphs$ and the sizes of $S4$'s ancestors ($S1, S2, S3$) have been reduced by $S4$'s size. The initialization path for p_1 is a sequence of transitions from the program's initial state to the subgraph's initial state. Dashed states in each of the resulting partition regions represent states that do not belong to the region but that are still reached as part of that region's search task; they are reached when exploring transitions that emanate from states within the region.

Figure 5 shows the final partition of the graph from Figure 3 into three regions. The scripts for p_1 and p_2 contain initialization paths to their respective root states. The resultant search scripts represent the certification tasks to be distributed among parallel processors.

The complexity of our partitioning algorithm is $O(k(S + T))$: steps 6, 8 and 9 each have running times of $O(S)$ for a reachability graph with S states, and steps 7 and 10 each have running times of $O(T)$. In our experiments, we noticed that this overhead translates into approximately 0.5% to 3% of the total certification time.

3.1.1 Updating Data Structures

In this section, we discuss how $Script$ and $Subgraphs$ are updated as our partitioning algorithm extracts each partition region p_i. We remove from $Script$ the subscript that represents the search of region p_i. Let s_i be the ID of the root state of p_i. Because $Script$ records a depth-first search of the reachability graph, and because state IDs reflect the order in which the states are discovered in this search, the subscript-to-be-removed starts *after* the leftmost instance of s_i and ends *before* the subsequent backtrack from s_i (to a state ID less than s_i). Thus, the subscript for region p_1 in Figure 6, with start state $S4$, is

p_1:

t_1	B	t_2	t_1	B	t_2	B	B
S5	S4	S6	S5	S6	S3	S6	S4

After discarding trailing backtrack commands, we obtain a search script $Script_1$ that specifies the search of region p_1, starting from the initial state of p_1:

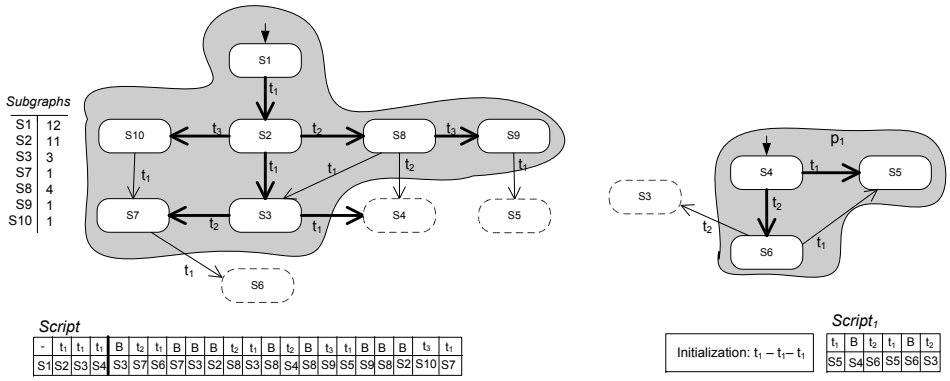

Figure 5: Result of partitioning after one iteration of algorithm

p_1 $(S4)$: t_1 B t_2 t_1 B t_2
S5 S4 S6 S5 S6 S3

Given a partition region p_i, updating $Subgraphs$ entails removing all entries that correspond to states in the region (line 8 in our partitioning algorithm). Again, let s_i be the ID of the root state of p_i. Any state in $Script_i$ whose ID is greater than or equal to s_i refers to a state in the region p_i and must be removed from $Subgraphs$. For example, in $Script_1$, states $S4$, $S5$, and $S6$ are removed from $Subgraphs$.

Each iteration of the partitioning algorithm produces a script for a different partition region. When the algorithm terminates, what remains of $Script$ forms a search script for the kth region. Figure 5 shows the search scripts for each partition region.

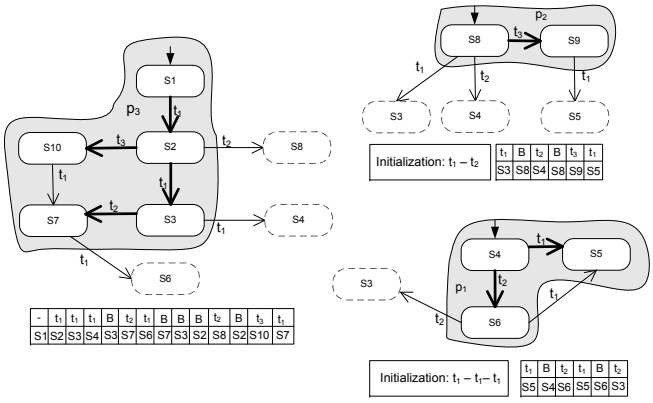

Figure 6: Subgraphs with scripts and initialization paths

3.1.2 "Constructing" Initial States

Each $Script_i$ starts at the root state of a partition region p_i. We could attempt to construct the corresponding "initial" program state for each search task, but JPF program states are complex and are difficult to construct and restore: they comprise not only the variable valuation but also information about threads and the progress of the search. Instead, we prefix each search script with an **initialization path**: a sequence of transitions from the program's initial state to the start state of the search task. We discuss in Section 3.4 the overhead incurred by this decision.

To construct the initialization path, the original $Script$ is scanned from start to end. Every time a transition is reached, it is pushed onto a stack. Every time a backtrack command is read, the top transition is popped off the stack. When a state ID s is first encountered, the transitions in the stack make up the initialization path from the program's initial state to state s. For example, the initialization path to p_1's root state is: t_1 t_1 t_1. Note that this algorithm does not construct the shortest path to a given state, but it does construct the shortest path with respect to the given script.

The states along the initialization path are all ancestor states of s in the reachability graph. Thus, we can use the same process to update the sizes of the subgraphs remaining in $Subgraphs$ after removing all states of p_i from $Subgraphs$ (line 9 of the algorithm).

3.1.3 Correctness

Our partitioning algorithm divides a search script in such a way that the resultant subscripts cover all states and transitions of the original script. Each iteration of the partitioning algorithm extracts a search subscript that corresponds to a *leaf subgraph* p_i of a program's reachability graph: the subgraph is rooted at state s_i, it include all states that are reachable from s_i via productive transitions, and includes all transitions originating from those states. Thus, the corresponding subscript is a contiguous substring of the original search script, starting with the first occurrence of initial state s_i and ending before the first backtrack from state s_i.

Because the extracted subscripts correspond to leaf subgraphs in the original reachability graph, their extractions do not affect the continuity of what remains of $Script$. When the algorithm terminates, what remains of $Script$ is a search subscript for a *contiguous* kth subgraph: the subgraph is rooted at the program's initial state s_1, includes states that are reachable from s_1 via productive transitions *up to and excluding the root states of the extracted partition regions*, and all transitions originating from those states. In this manner, the algorithm splits $Script$ without removing any states or transitions (except backtrack transitions).

If the provided $Subgraphs$ is not accurate with respect to the program's reachability graph, the partitioning algorithm will still produce sub-script that cover disjoint regions and, taken together, cover the program's entire reachability graph. For example, if one or more entries in $Subgraphs$ list incorrect sizes of subgraphs, then the algorithm will simply produce partitions whose sizes have a larger standard deviation. Alternatively, if one or more states are missing from $Subgraphs$, this will be detected in line 8 when the states of a region are removed from $Subgraphs$. If $Subgraphs$ contains extra entries, this will be discovered if the algorithm ever chooses one of those states to be the root of a region; otherwise, it will have no effect on the algorithm.

3.2 Parallel Certification

The program and search scripts are distributed to parallel processors, which run the certifier's model checker. Each processor keeps its own local copy of FP, mapping state IDs to program-state fingerprints. If a processor detects any discrepancy between its search

Table 3: Results for Parallel SCC Certification

	SCC certification						Trustful SCC certification					
# sub-scripts	10		50		100		10		50		100	
Program	Max task	Speed up	Max task	Speed up	Max task	Speed up	Max task	Speed up	Max task	Speed up	Max task	Speed up
Dining Phil	13%	8	4%	22	2%	38	11%	39	4%	103	3%	133
Bounded Buffer	11%	9	4%	25	3%	30	12%	48	4%	140	2%	270
Nasa KSU Pipe	12%	8	4%	22	3%	25	11%	30	5%	64	3%	104
Nested Monitor	11%	9	5%	18	3%	28	10%	56	4%	136	3%	175
Pipeline	12%	8	6%	15	2%	39	13%	39	5%	96	3%	155
RWVSN	11%	9	4%	22	3%	27	11%	38	5%	80	2%	194
Replicated Workers	12%	8	5%	18	2%	40	12%	34	4%	100	3%	130
Sleeping Barber	11%	9	4%	23	3%	28	11%	31	4%	85	2%	164
Elevator	10%	10	4%	23	2%	45	12%	57	5%	132	4%	160
Average	11%	9	4%	21	3%	33	11%	41	4%	104	3%	165

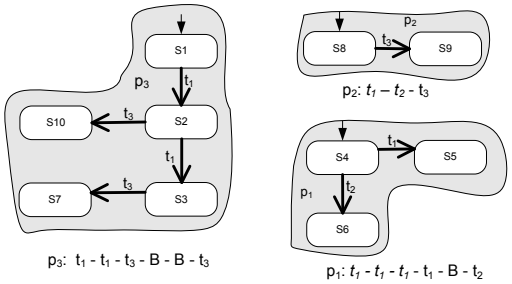

Figure 7: Script partition for trustful SCC

Table 4: Avg. and Max. Lengths of Initialization Paths

# sub-scripts	10		50		100	
Program	Avg path	Max path	Avg path	Max path	Avg path	Max path
Dining Phil	11	16	13	16	12	18
Bounded Buffer	75	615	217	6742	139	6678
RWVSN	50	495	57	785	101	845
Sleeping Barber	12	25	17	35	25	31
Elevator	50	68	71	75	71	78
Average	**31**	**153**	**53**	**877**	**53**	**876**

script and the program, it raises an error. In addition, once all processors have finished their certification tasks, the processors' FP maps are compared to ensure that all processors map state IDs to the same fingerprints. Any mismatch is reported as an error. This final check on the veracity of the search scripts performs at most nS comparisons, where n is the number of processors and S is the total number of states.

3.3 Parallel Trustful Certification

The algorithm for partitioning a search script for trustful certification is similar to the algorithm presented in Figure 4, but is applied to a trustful $Script$ (which contains no unproductive transitions). The only difference between the algorithms is that the partitioning algorithm for trustful certification removes the productive transitions that span regions (e.g., the transition from $S3$ to $S4$ in Figure 6). Figure 7 shows the partitions that we obtain for parallel trustful certification of the sample reachability graph given in Figure 3. The regions represent spanning subtrees of the original reachability graph.

3.4 Implementation and Evaluation

We implemented parallel SCC in Java Pathfinder and refer to the resulting model checker as **JPF-pscc**. For convenience, JPF-pscc supports both verification and certification modes. In the verification mode, JPF-pscc generates a search script to be used during certification. In certification mode, JPF-pscc can be used to partition the search script into k scripts or to model check the program using one of k scripts to direct its search. At the end of a certification task, JPF-pscc outputs its FP map. At present, a separate program is needed to compare the FPs from all certification tasks.

To evaluate the performance of parallel SCC, we used JPF-pscc to partition each program's state space into 10, 50 and 100 certification tasks (i.e., sub- search scripts). Because the sizes of the resulting scripts are not exactly equal, we report for each program the time it takes to examine the largest sub-script. To this time we have added (1) the time it takes to partition the search script and (2) the time it takes to compare all FP maps sequentially. In practice, the actual time of this latter task would be less because the search tasks would finish at different rates and FP maps could be compared against the current master map as tasks complete.

Table 3 shows the results for parallel SCC certification and parallel trustful SCC certification. For each certification method and the number of partitions (10, 50, or 100), column *Max task* lists the size of the largest search script for each program; for SCC certification, size is reported as a percentage of the program's number of transitions, and for trustful SCC certification, size is reported as a percentage of the program's number of states. Columns *Speed-up* report the speed-up in certification time over the time to verify the entire program using JPF, as reported in Table 1. In SCC certification, the size of the largest sub-script determines the optimum number of processors to use during certification. For example, when partitioning the reachability graph of the dining philosophers example into 10 subgraphs for SCC certification, the size of the largest resulting subgraph is 13%. Thus the optimum number of parallel processors to use is 8. Taking this into consideration, the results show that speed up for parallel SCC certification is on average a factor of n, for n processors. Trustful SCC certification can achieve a speed up of up to a factor of $5n$, for n processors.

The speed-up factors reported in Table 3 are not simply the product of the speed-up factors reported for nonparallel SCC certification (in Section 2) and the number of parallel processors employed. This is partly because of the time needed to compare FP maps at the end of certification, and partly because the search tasks vary in size and we report the timings associated with the largest task. Most of the certification sub-scripts are prefaced with an initialization path, which affects the size of the script. Table 4 reports the average (column *Avg path*) and longest (column *Max path*) initialization paths for the scripts generated for parallel SCC certification for five programs. The initialization paths for the remaining programs show a similar pattern and are not shown because of space considerations. The reported average values are for all nine programs. Most path lengths are relatively short, and JPF-pscc can explore approximately 1000 transitions per second. The lengths of initialization paths for trustful SCC certification are similar.

4. DISCUSSION

In this section we discuss some outstanding issues of SCC, including some of our design decisions, restrictions on the properties that can be checked, requirements on the model checker(s) used, scalability, and compatibility with search-space reduction techniques.

4.1 Properties

Safety properties play an important role in formal verification because they assert that the system stays within required bounds and does not perform any "wrong" actions [14]. SCC can be used to certify invariants and program assertions, and can also check for deadlock violations. Because the search script encodes all transitions of a program's reachability graph, SCC can also be used to check invariants over consecutive states, such as the property

$$(x = 5) \rightarrow next(x = 8)$$

which states that if the value of x is 5, then in the next state its value will be 8. Even when certification is parallelized, each SCC search task is responsible for covering a set of contiguous states and all of their outgoing transitions. Thus, every pair of consecutive states is captured in a search script, making it possible to certify invariants over consecutive states. In contrast, trustful SCC does not cover all transitions, so it does not cover all pairs of consecutive states. Thus, trustful SCC can soundly certify only state properties.

4.2 Scalability

A number of factors affect the scalability of search carrying code. For one, SCC certification is limited to finite state programs. However, this limitation applies in general to explicit-state model checking. Thus, if a program can be verified using explicit-state model checking, then it can be verified and certified using SCC. If the code producer uses abstractions to produce a finite state space for SCC verification, then the certifier must use the same abstractions and must check that the abstractions preserve the properties being proven.

Another factor is that the results of our experiments (reported in Table 3) suggest that the benefits of parallelization diminish as we increase the number of sub-scripts we divide an SCC script into. Our partitioning algorithm does not partition a script into sub-scripts of exactly equal size, plus the resulting sub-scripts are prefaced by initialization paths of varying lengths. As such, the speed up in certification time is bounded by the amount of time it takes to certify the largest sub-script. In the worst cases, when a script is partitioned into 50 or 100 sub-scripts, the largest sub-script is 2 to 3 times the size that would be expected if the sub-scripts were truly equal sized. We do not know whether the observed diminishing of returns is due to the small sizes of the programs in our test suite, or is inherent to our approach. More experiments on larger programs are needed to answer this question.

A more serious issue is the size of the search script that the code producer provides, likely over a network, to the code consumer. The size of a compressed script, in number of bytes, is on the order of the number of states in the program's state space – which could be very large in the worst case, where the program's state space is at the limit of what can be model checked. In this paper, we assign responsibility of partitioning the script to the code consumer, on the assumption that she knows how many processors are available and thus knows how many sub-scripts to create. However, in cases where the script is large, it may be prudent for the code producer to partition the search script. This would certainly be the case if it turns out that there is a limit to how evenly the script can be partitioned into sub-scripts, as discussed above.

4.3 Using Different Model Checkers

In our work, we have shown how SCC can be used when the certifier and verifier use the same model checker. However, ideally, the code consumer should be able to use a model checker of her own choosing (adapted to use search scripts), and not be restricted to using the same model checker as the code producer. We have not evaluated using SCC with different model checkers, but we expect that different model checkers can be used for verification and certification, as long as they satisfy certain requirements. To start, both model checkers must be explicit-state model checkers, as SCC does not support symbolic model checking. Second, the two model checkers must agree on how to interpret transition statements in the search script (e.g., they must both match a byte-code instruction to the same instruction in the program being explored). Third, the certifier model checker must create unique fingerprints for distinct states (for checking the veracity of the script), but there are no constraints on how this should be done.

If state-space reduction techniques are used, then the situation is more complicated because the techniques could change the size and shape of the reachability graph. Discussion of how SCC interacts with state-space reduction techniques follows.

4.4 Model-Dependent Reduction Techniques

A key question of any new model checking technique is whether and how it works in combination with existing search-reduction techniques. We discuss model-dependent reduction techniques in this section and property-dependent techniques in the next section.

We expect SCC to complement model-dependent reduction techniques, as long as (1) the reduction techniques are applied in advance or on-the-fly, so that the search script encodes the reduced reachability graph, and (2) the verifier and certifier model checkers agree on the abstractions applied. We consider only automated reduction techniques; techniques that rely on user-input (e.g., abstraction functions [5]) are not safe, because a malicious code producer could specify an unsound abstraction.

Symmetry Reduction [9] reduces the size of the state space by exploiting symmetries among states. The idea is that states are grouped into equivalence classes, and the model checker can discard a state if an "equivalent" one has been explored before. There are a number of different techniques for identifying symmetries [16], but the ultimate effect is that symmetric states are assigned the same fingerprint.

In SCC verification, symmetries result in a reduced reachability graph being explored, and a smaller search script being generated. If the same model checker is used during SCC certification, it identifies the same symmetries, symmetric states are assigned the same fingerprint, and the shape of the reduced reachability graph matches the search script. If the code producer and consumer use different model checkers, the checkers must implement the same reductions.

Currently, it is not realistic to expect different model checkers to use the exact same symmetry reductions. But if model checkers were parameterized with respect to their state-space reduction techniques and algorithms, then requiring both model checkers to use the same symmetry reductions would not be a limitation. In fact, there has already been some work along these lines [8, 11].

Partial Order Reduction (POR) [10] is an automated path-reduction technique that finds transitions that are independent of each other and whose interleaved executions all lead to the same state, regardless of the order of their execution. POR executes only one of the possible interleavings.

During SCC verification, the model checker detects independent transitions, explores only one interleaving, and records only one interleaving in the search script. The entire interleaving is

recorded as a single transition in the search script. If the same model checker is used during SCC certification, then the certifier's model checker identifies the same sets of independent transitions, chooses the same interleavings (as long as decisions are deterministic), and disables the other interleavings. As a result, the POR interleavings chosen during certification match the search script.

Because a POR interleaving is treated as a single, compound transition, it is never partitioned among different subscripts and during certification, an entire interleaving is assigned to a single processor. Thus, POR does not interfere with SCC, even after parallelization.

If different model checkers are used for SCC verification and SCC certification, they must both use the same POR heuristics to (1) determine which transitions are independent and (2) select which interleaving to explore; and the heuristics must be deterministic. It might seem unrealistic for both model checkers to use the same heuristics, but we believe a parameterized approach to state-space reductions, as described above, could address this limitation.

4.5 Property-Specific Reduction Techniques

The goal of property-specific reduction techniques is to reduce the search space (and search script) to those program states that are relevant to the property being checked. Such reductions are problematic for SCC because the code producer does not know in advance which properties are of interest to the code consumer and thus cannot apply the appropriate reductions. Moreover, the code consumer cannot simply apply the reduction techniques herself because the resulting reduced program would no longer correspond to the supplied search script. Such techniques can only be useful if they can be applied to the search script rather than to the program.

Consider program slicing [28], which is a commonly used property-specific reduction technique that reduces the size of the search space by ignoring program statements that are not relevant for a given property. Traditional program slicing cannot be used in conjunction with SCC for the reasons given above, but it might be possible for the code consumer to slice the search script instead, given that the script's transition instructions (which are bytecodes) literally encode the program statements. The certifier model checker would need to be able to determine from a transition instruction in the search script whether the transition is relevant to the property being checked. It would also need to perform a definition-use analysis on the script, which is a much larger artifact to analyze than the original program. Lastly, not all irrelevant transitions can be removed from the search script because the sliced script must still be a valid path in the program's reachability graph.

We are still investigating the problem of script slicing. Although it seems to be possible, it is not clear whether the resulting reductions will be significant. In general, the savings achieved by program slicing cannot be predicted in advance, and it is possible that slicing provides no significant savings at all – especially when checking a large collection of varied properties, such as during certification.

5. RELATED WORK

In previous work [26], we suggest using information gathered during one model checking run to speed up subsequent runs, and we evaluated a JFP-based prototype on six Java programs. We extend that work here by evaluating it on more and larger programs, protecting against tampering of the search script, and parallelizing the certification searches.

Our work is inspired by proof-carying code (PCC) [18]. Significant infrastructure is needed to support PCC, including inference rules for reasoning about code, a formal language for expressing safety properties and proofs, and an algorithm for checking a program and its safety proof. Most research on PCC focuses on reducing the size of proofs [4] and generalizing the kinds of properties that can be proved [1, 19]. In contrast, SCC-based certification can be implemented by making modest changes to existing model-checking technologies. SCC search scripts are created automatically, and can be decomposed into an arbitrary number of smaller scripts. While a PCC certificate encodes the proof of particular properties, an SCC script is independent of any properties. Thus, the SCC script can be reused to speed-up checks of newly identified properties.

Techniques most closely related to SCC are abstraction-carrying code (ACC) [29] and model-carrying code (MCC) [24]. In both cases, the program to be certified is accompanied by an abstract model of the program. In ACC, this abstract model is an abstract interpretation of the program. In MCC, the model is an extended finite-state automaton over the alphabet of system calls, and is synthesized from the program's execution traces. In both cases, the abstract models are property independent. Certification is a two-step process: (1) certifying that the model is a faithful abstraction of the program and (2) certifying that the model respects the desired properties. In ACC and SCC, certification is done offline. In MCC, model veracity is checked at runtime by monitoring the program, which incurs a performance penalty of 2% to 30% [24]. ACC and MCC support richer property languages (e.g., temporal logic) than SCC does. In addition, ACC and MCC can accommodate infinite-state programs. However, ACC and MCC models are conservative abstractions, which means that they are susceptible to spurious errors. Worse, an MCC model may be unsound if it is synthesized from a deficient set of traces. In contrast, an SCC script results in a sound and complete search of the state space. Importantly, it is not known whether ACC or MCC can be parallelized.

Lauterburg et al. [15] propose a technique to support incremental state-space exploration that is based on storing information regarding the entire reachability graph. They use the information to speed up the model checking of subsequent versions of the same program–a task that is comparable to our SCC certification. The maximum reported speed up is a factor of approximately two, and their technique has not been parallelized.

Early work on parallel model checking tried to speed up the process of finding errors by distributing the model-checking search among multiple workstations on a network (distributed memory). The challenge was to distribute the workload evenly among the parallel processors. Stern and Dill [25] parallelized Murφ, an explicit-state verifier, by pre-assigning states to processors based on a hash of the fingerprint. However, this approach could result in an imbalance in workload among the processors. Moreover, substantial state information must be passed between processors whenever a new state is discovered and must be transferred to its assigned processor. Subsequent works by others investigate how to improve local and global load balancing [3, 13] and reduce communication overhead [20].

On a shared memory architecture, communication among processors is negligible, but the processors must synchronize their access to shared variables, such as when they deposit states into each other's worklist or access a shared hash-table of state fingerprints. Thus, there is the extra challenge of keeping processors utilized and not waiting too long to access shared resources. Interestingly, some researchers report [2, 12] that, beyond an optimal number of processors, the search time starts to *increase* with the number of additional processors because the synchronization overhead dominates any benefit from parallelization. Parallelized SCC does not suffer from this overhead because the reachability graph is parti-

tioned in advance in such a way that no communication or synchronization among processors is necessary. Each processor works independently of others, and shares information with an administrator process (which collects and compares fingerprint maps) only at the end of its search task.

6. CONCLUSION AND FUTURE WORK

We have proposed search carrying code (SCC) as a technique to certify software that was previously verified by software model checking. We have shown that SCC certification can determine if the provided search script (i.e., the program's *certificate*) is not a faithful representation of the program's state space, and that SCC certification can be parallelized to speed up the certification task. In the special case where the source of a program is trusted, we show how SCC certification can be further optimized by searching a spanning-tree representation of the program's reachability graph and eliminating data structures and checks on the search script.

In the future, we would like to continue to address limitations of SCC. Our current focus is on alternative representations and encodings of the information in the search script in order to reduce its size. We are also investigating in more detail how SCC can be used in combination with various search reduction techniques. Finally, we would like to evaluate SCC on industrial-sized programs.

Acknowledgments - We would like to thank Peter Mehlitz and Willem Visser for helping us understand Java Pathfinder. We would also like to thank the reviewers for their insightful comments that helped to improve this paper.

7. REFERENCES

[1] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.*, 32(3):1–67, 2010.

[2] J. Barnat, L. Brim, and P. Rockai. Scalable multi-core LTL model-checking. In *SPIN*, pages 187–203, 2007.

[3] J. Barnat and P. Ročkai. Shared hash tables in parallel model checking. *Elect. Notes Theor. Comp. Sci.*, 198(1):79–91, 2008.

[4] F. Besson, T. Jensen, and T. Turpin. Small witnesses for abstract interpretation-based proofs. In *Proceedings of the 16th European conference on Programming*, pages 268–283, 2007.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of Princ. of Prog. Lang.*, pages 238–252, 1977.

[6] D. Davies. http://ddavies.home.att.net/NewSimulator.html.

[7] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proc. of the Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 73–89, 2006.

[8] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *Proc. of Int. Conf. on Software Engineering*, pages 177–187, 2001.

[9] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Form. Meth. System Design*, 9(1-2):105–131, 1996.

[10] P. Godefroid. Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. In *Lecture Notes in Computer Science*, volume 1032, 1996.

[11] J. Hatcliff, M. B. Dwyer, C. S. Păsăreanu, and Robby. Foundations of the bandera abstraction tools. In *The essence of computation: complexity, analysis, transformation*, pages 172–203, 2002.

[12] C. P. Inggs and H. Barringer. Ctl* model checking on a shared-memory architecture. *Form. Methods Syst. Des.*, 29(2):135–155, 2006.

[13] R. Kumar and E. Mercer. Load balancing parallel explicit state model checking. In *Workshop on Parallel and Distributed Methods in Verification*, pages 19–34, 2005.

[14] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, 2001.

[15] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. of 30th Int. Conf. on Software Engineering*, pages 291–300, 2008.

[16] A. Miller, A. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comp. Surv.*, 38(3), 2006.

[17] Nasa. Java pathfinder, version 4. In *http://javapathfinder.sourceforge.net*, 2007.

[18] G. C. Necula. Proof-carrying code. In *Symp. on Prin. of Programming Languages*, pages 106–119, 1997.

[19] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *33rd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages*, pages 320–333, 2006.

[20] D. M. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal Parallel Distributed Computing*, 47(2):153–167, 1997.

[21] D. Park, U. Stern, and D. Dill. http://verify.stanford.edu/uli/icse/workshop.html.

[22] PKZIP. Zip, 1989.

[23] Santos Laboratory. http://www.cis.ksu.edu/santos/case-studies/counterexample_case_study.

[24] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. Duvarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of 19th Symp. on Operating Sys. Principles*, pages 15–28, 2003.

[25] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Proc. of the Conf. on Computer Aided Verification 97*, volume 1254, pages 256–267, 1997.

[26] A. Taleghani. Using software model checking for software component certification. In *Companion Proc. of ACM/IEEE Int. Conf. on Soft. Eng.*, pages 99–100, 2007.

[27] W. Visser, G. Brat, K. Havelund, and S. Park. Model checking programs. In *Proc. of Int. Conf. on Automated Software Engineering*, pages 3–12, 2000.

[28] M. Weiser. Program slicing. In *Proc. of Int. Conf. on Software Engineering*, pages 439–449, 1981.

[29] S. Xia and J. Hook. Certifying temporal properties for compiled C programs. In *Proc. of the Conf. on Verif., Model Check., and Abstr. Interpret.*, pages 161–174, 2004.