On-The-Fly Path Reduction

Sebastian Biallas[1] Jörg Brauer[1] Dominique Gückel[2]
Stefan Kowalewski[1]

Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany

Abstract

Path reduction is a well-known technique to alleviate the state-explosion problem incurred by explicit-state model checking, its key idea being to store states only at predetermined breaking points. This paper presents an adaptation of this technique which detects breaking points on-the-fly during state-space generation. This method is especially suitable for the detection of breaking points in systems where static analyses yield coarse over-approximations. We evaluate the effectiveness of this technique by applying it to binary code verification.

Keywords: verification, model checking, state explosion, path reduction

1 Introduction

Despite a significant amount of research on abstractions, state explosion is still a major obstacle for the applicability of (explicit-state) software model checking to real-world applications [5]. One such abstraction for CTL model checking is the so-called *path reduction* [18]. The key idea of path reduction is to collapse single-successor chains in the state space if intermediate states cannot influence the validity of a specification. This means that states are only stored when visiting program locations that cause a branching in the state space or influence the validity of the CTL specification. These program locations are called *breaking points*. For instance, a program statement that alters the value of a variable used in an atomic proposition or reads a nondeterministic value, which causes a branching in the state space, is called a breaking point.

[1] Email: {lastname}@embedded.rwth-aachen.de

[2] Email: gueckel@embedded.rwth-aachen.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Storing states only at these specific locations reduces the memory footprint of the state space, possibly at the cost of increased runtime.

In their seminal paper, Yorav and Grumberg [18] have described the detection of breaking points using static analysis for the model checker Murφ. Due to the rather limited nature of the input language used in Murφ, breaking points can be determined accurately for this specific tool. However, this is not always so. A domain where this approach may lead to coarse over-approximations is binary code verification for embedded systems [13,14]. This has different reasons, for example:

- Programs for low-level platforms are frequently interrupt-driven. In this case, states have to be stored at any program location where interrupts may fire because the execution of an interrupt handler is optional, and thus causes a split in the state space.
- Nondeterminism is often introduced through the hardware itself. Reading the value of the same register, say, an input port, may lead to either deterministic or nondeterministic values, depending on the exact hardware configuration. Unfortunately, no static analysis techniques are known that can infer the state of the hardware as precisely as required.
- To guarantee termination of the model-checking process, states need to be stored in possibly nonterminating loops (for fixed-point detection). Thus, at least one location in each loop has to be breaking. Despite the advances in termination proofs for high-level programs [7], these techniques are not yet applicable to low-level code.

Consequently, path reduction for binary code model checking based on static analysis does not reach the effectiveness known from other domains [14, Sect. 6.2]. Binary code model checkers, however, typically generate state spaces using dedicated simulators of the target microcontroller. The exact configuration of the microcontroller is thus known during state-space building. For example, when simulating an indirect store instruction, the concrete target address of this instruction is known, and breaking-point detection does not have to rely on conservative over-approximations.

Our main contribution in this paper is a new technique called *on-the-fly path reduction*, which performs state-space reductions dynamically while state spaces are built (see Sect. 2). This technique is novel in that it performs tasks such as detecting fixed points while states are generated. We also detail how to expand counterexamples obtained with path reduction to concrete counterexamples [3,6] (see Sect. 3). To evaluate the effectiveness of on-the-fly path reduction, we compare its performance to results obtained using static detection of breaking points in Sect. 4. Finally, Sect. 5 presents related work and Sect. 6 concludes the paper.

2 Reducing Paths On-the-fly

We implemented the path reduction by static analysis (SPR) and the new on-the-fly path reduction (OPR) for the model checker [mc]square [13] which we used for our case studies. This section details the motivation and implementation of the algorithms.

2.1 Preliminaries

[mc]square uses an on-the-fly model checking algorithm according to [8]. This means the state space is generated on-the-fly: A state corresponds to a configuration of the microcontroller and if the model checker needs to follow successor states that are not created yet, the simulator is used to create them on demand. A path reduction algorithm needs to cope with the on-the-fly generation of states. That is, instead of returning the direct successors by executing a single instruction, the simulator should follow a path of subsequent instructions returning the next state that should show up in the reduced state space. What qualifies a state as the *next state* (and hence determines the reduced state space) needs to be selected in a way that the reduction is indistinguishable by CTL_{-X} logic. Here, CTL_{-X} denotes the logic CTL without the next time operator X, which cannot be used for reduced state spaces. We call states which match such a criterion and thus are used as the new successor states *breaking*. A path $(a, b_1, b_2, \ldots, b_n, c)$ where at most a and c are breaking is called *elementary path*. So each transition in the reduced state space corresponds to an elementary path in the original state space.

As an example, Fig. 1 shows a state space with 14 states labelled a to n. The corresponding reduced state space is shown in Fig. 2. Although only the four breaking states a, c, f and n remain, the state space is stuttering bisimilar [2,16]

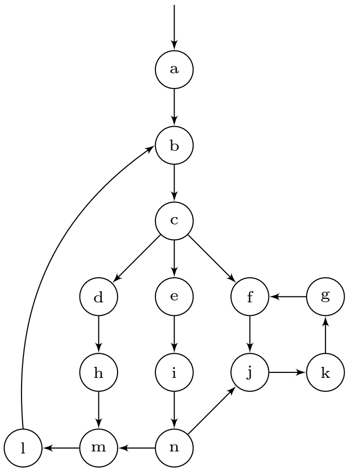

Fig. 1. Example state space

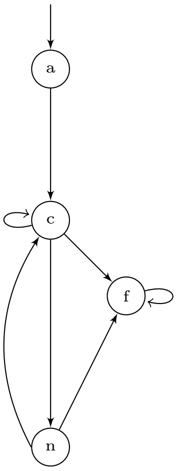

Fig. 2. With path reduction applied

to the original state space. This is achieved by merging elementary paths like (c, e, i, n), where the intermediate states have only one successor, into a single transition (c, n). Note that such a pruning of states is also possible for loops like (c, d, h, m, l, c) which is merged into the transition (c, c). Note also that state b does not show up in the reduced state space although it has two predecessors. The loop (f, j, g, k, f), however, requires special treatment, since none of its states have multiple successors.

We will now evaluate the criteria for a state to be breaking, i. e., to show up in the reduced state space. For SPR, this criterion solely depend on the program counter. Here, breaking program locations (*breaking points*) are found using static analysis. Afterwards, for each state with a program counter matching these values the path pruning algorithm stops during state space generation. The static analysis [18,14] selects each location as a breaking point,

- where a nondeterministic interrupt can be triggered, or
- where the corresponding instruction performs some nondeterministic operation as reading from an input register, or
- where the corresponding instruction alters the value of a variable (memory location) used as an atomic proposition in the CTL formula, or
- which is target of a jump (or other control transfer) instruction.

The first two conditions make locations breaking where the corresponding state has more than one successor in the state space. The third condition guarantees that all changes of variables used in the formula are visible to the model checker. Finally, the fourth condition ensures the existence of a breaking point in every loop in the program, such that every loop in the state space has at least one breaking state for fix-point detection.

As mentioned in the introduction, using static analysis to determine the breaking points usually yields a coarse over-approximation and thus assumes more locations breaking than necessary. For example, a read-operation on an I/O port which is configured for input returns a nondeterministic value, which in turn leads to a branching in the state space. The corresponding program location is thus marked as breaking. However, an I/O port may also be configured for output, which means that it stores a deterministic value. In this case, the corresponding program location does not need to be breaking. Since static analysis fails to accurately capture such bit-level dependencies of the hardware, it computes overly pessimistic results. Additionally, the fourth condition makes it impossible to prune long running loops of the program into a single transition. In the next subsection, we will focus on the new OPR that uses criteria which can be evaluated on-the-fly to determine whether a state is breaking and thus can handle these issues.

2.2 On-the-fly Path Reduction

The general idea of OPR is that we decide for each state (instead of each program location) whether it is breaking by evaluating its local neighborhood. This decision is made during state space generation, and we therefore need conditions that can be checked on-the-fly: While generating states along a elementary path, the OPR assumes a state as breaking if

- it has more than one successor, or
- the truth value of a atomic proposition changes after the transition to its (sole) successor, or
- it was already visited in this elementary path.

The first condition assures that states where a nondeterministic decision has to be taken (such as the execution of an interrupt or a read from a hardware register) show up in the state space. To maintain visibility of all changes to the model checker, the second condition assures that at most one transition might influence the formula in each elementary path. The last criterion is needed to guarantee termination and will be studied in detail later on. First, we formally describe the algorithm for the OPR successor state generation:

Algorithm 1 Generate successors of state in reduced state space

```
Input: $sourceState$
Output: successors of $sourceState$ in the reduced state space
 1: $successors \leftarrow createDirectSuccessors(sourceState)$
 2: $resultSuccessors \leftarrow \{\}$
 3: for all $state$ in $successors$ do
 4:    $current \leftarrow state$
 5:    $visited \leftarrow \{\}$
 6:    repeat
 7:       $visited \leftarrow visited \cup \{current\}$
 8:       $nextStates \leftarrow createDirectSuccessors(currentState)$
 9:       $next \in nextStates$
10:       $breaking \leftarrow |nextStates| \neq 1$ or $atomics(current) \neq atomics(next)$
11:       if not $breaking$ then
12:          $current \leftarrow next$
13:       end if
14:    until $breaking$ or $current \in visited$
15:    $resultSuccessors \leftarrow resultSuccessors \cup \{current\}$
16: end for
17: return $resultSuccessors$
```

For each direct successor, the inner loop (lines 6–14) of the algorithm follows its elementary path (line 12) until a breaking state is found. The first two breaking conditions are checked in line 10. In the next section it will be described how to implement the loop detection in line 7 and line 14, which is used to meet the third condition for breaking states.

2.3 Loop Detection

We describe three different criteria for the detection of loops in elementary paths, which can be checked one-the-fly:

- Stop if the same the same program counter is encountered twice.
- Stop if the same state is encountered twice.
- Stop if the same hash code of a state is encountered twice.

The first criterion is inspired by SPR and guarantees that every loop (in the program and thus in the state space) contains at least one breaking instruction. The drawback is that each loop without nondeterministic control flow (for example a memory copy or initialization), will create at least one state in the state space for each iteration.

To prune such loops, the second criterion takes the whole microcontroller configuration into account, that is, all states along elementary paths are temporarily stored and a state is assumed breaking once is already in this list. Since states have finite size, this is guaranteed to terminate for all loops. This detangles loops in the control flow from loops in the state space and hence allows for representing loops as single transitions.

The third criterion is an improvement of the second criterion. It takes just the 64 bit hash code of the raw state data as a criterion for detecting already encountered states. This is faster and less memory intense, since only the hashes of all intermediate states have to be stored while simulating along an elementary path. As our case study will show, the third criterion offers the high accuracy while being a very fast possibility to detect a loop in the state space.

To summarize, the advantages of the OPR introduced so far are:

- There is no need for static analysis, which yields more accurate results.
- Our algorithm is independent of the microcontroller simulator used for the state generation. For SPR, on the other hand, detailed knowledge of the microcontroller is necessary to detect breaking states.
- It is possible to prune program loops with many iterations into single transitions.

While this section dealt with reducing paths to alleviate the state explosion, we will discuss how to re-expand paths to create meaningful counterexamples in the next section.

3 Expanding Reduced Paths for Counterexamples

Counterexamples/Witnesses are paths (possibly with loops) in the state space, showing how some undesired property is reached or some desired property is never reached. Counterexamples provide crucial information to help understanding why formulae are valid or violated [4]. A counterexample for the formula $AG\ x \neq 5$, for instance, would show a path (and thus all nondeterministic inputs) that leads to a state where x equals 5. A counterexample for the formula $AF\ x = 6$, on the other hand, would show a path into a loop, such that $x = 6$ is never valid.

In a reduced state space, counterexamples are less useful. To illustrate, consider Fig. 3 where the counterexample trace (a, c, d) is shown; the formula is *false* in state d. States omitted by the path reduction (b and b') are shown as dotted circles. To understand such a counterexample trace, it is crucial to know which nondeterministic decision has been taken for the (a, b) transition. Unfortunately, this information is not readily available from the (a, c) transition visible in the counterexample trace: State c might be too far away to distinguish the (a, b) transition from the (a, b') transition without manual investigation.

To remedy this problem, we implemented means to reverse the effect of path reduction on given counterexample traces in [mc]square by re-expanding all reduced paths. Such re-expanded counterexamples are then identical to their corresponding traces in the original state space.

This is achieved in two steps. In the first step, all states omitted by the path reduction are recreated using the simulator. For states with only one successor, it is sufficient to create the direct successors until the next state on the reduced path is reached. For states with more than one successor, however, we have to find out which transition actually belongs to the counterexample. This corresponds to the decision between the (a, b) and (a, b') transitions in Fig. 3. To decide which of these nondeterministic transitions belongs to the counterexample, a breadth-first search for the target state c is started at node a. The search terminates when state c is found. States with more than one successor do not need to be followed because they are breaking and thus part of the reduced state space. The path leading to state c is then added to the counterexample making the nondeterministic decision shallow.

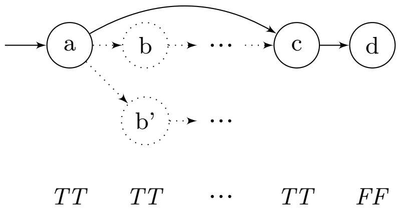

Fig. 3. Counterexample trace

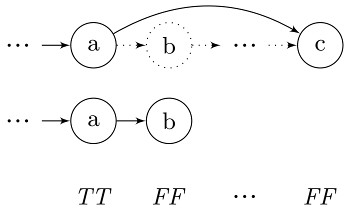

Fig. 4. Counterexample shortening

In the second step, states are dropped at the end of elementary paths where the violation of the formula manifests itself in the first transition but is noticed at the end of this path. Such a situation is depicted in Fig. 4 (upper part). Let us assume that after the (a, b) transition the formula is violated, i. e. the formula is *true* in state a but *false* in states b to c (recall that on each elementary path only the first transition might influence the formula). Since only a and c are stored in the reduced state space while the states in between are omitted, the violation of the formula is detected in state c, yielding a counterexample ending in c. It is desirable to have shortest counterexamples (which highlight the first instruction invalidating a formula), and we thus drop the states b to c. Formally, a path that ends in $(\ldots, a, c)$ in the reduced counterexample is transformed into $(\ldots, a, b)$, where b is the direct successor of a in the original state space. This is shown in the lower part of Fig. 4.

The remaining disadvantage of path reduction is the loss of the X operator for CTL. As we perform model checking on the level of machine instructions, the X operator is not of practical relevance anyway.

4 Case Studies

This section describes different experiments to examine the impact of OPR with respect to different evaluation criteria: the effects of different approaches for loop detection (see Sect. 4.1), a comparison to path reduction based on static analysis (see Sect. 4.2), and the effects of CTL specifications on the generated state spaces (see Sect. 4.3). All experiments were run on a SUN Fire X4600 M2 server equipped with eight AMD Opteron dual-core processors and 256 GiB of RAM. However, only a single processor was used in order to obtain unbiased results.

4.1 Variants of On-the-fly Path Reduction

The first case study focuses on the effects of different loop-detection criteria (cp. Sect. 2.3), which determine the termination of a path compression step. In order to obtain realistic results, the program to be verified has to execute several loop iterations, ideally with as little overlapping of the iterations as possible. A program called `vector` from our benchmark set satisfies this requirement. This program continuously reads inputs from the environment in a nonterminating loop. The values are then interpreted as integer vectors and used for different typical vector operations.

For each criterion, the model checker had to generate the entire state space of the program. The results of these runs are shown in Tab. 1. The reference values without any abstraction are shown in Tab. 2, in the entry for `vector`.

As expected, the “same pc” criterion results in the largest state space

Criterion	States stored	States created	Size [MB]	Time [s]
Same PC	1,502,901	179,089,399	408.67	2,123
Same hash	4,656	496,768,475	23.7	4,891
Identity	4,656	496,768,475	23.7	5,057

Table 1
Loop-abortion criteria

of the three criteria. Compared to the state space size without applying path reduction, this still amounts to a reduction by 96.83%. Comparing states for identity in a byte-wise fashion results in a much smaller state space, which is unsurprising. The interesting aspect of this experiment is thus to determine whether the effort for identity checking has significant advantages over the simpler checking for hash collisions. Regarding the number of stored states, there is no difference between hash collision detection and state identity detection for this particular program. This means that no two different states are mapped to the same hash values. The time required when checking for state identity, however, was slightly higher than for detecting hash collisions.

Judging from these results, we conclude that the checking for hash collisions is an adequate compromise between runtime and memory consumption concerns. Hence, in the following case studies, we use hash collision checking as the default criterion in OPR.

4.2 Comparison to Other Abstraction Techniques

For the second set of experiments, we have used [mc]square to generate state spaces on different levels of abstraction: (i) no abstraction, (ii) SPR, and (iii) OPR. For a thorough evaluation, we present experimental results for five different microcontroller programs. The results of the different runs are shown in Tab. 2.

The first program is called `light_switch` and models a reactive electrical switch based on a state machine. The program uses two hardware timers, but no interrupts. Model checking this extremely simple program with SPR results in a reduction of the state space size by 73.88%, but it increases the number of states created by 156.69%. In comparison, OPR reduces the state space even further by 88.43%, relative to the original results. The lower number of states stored in the state space also influences the number of states that have to be recreated during model checking. Hence, the number of states created increased by approx. 300%. For this small program, neither technique had a noticeable effect on the memory consumption or the runtime. The memory footprint in this case is largely influenced by the initial sizes of the hash tables used for storing state spaces.

The second program, called plant, controls a fictive chemical plant. Consisting of 225 lines of code, it is slightly longer than light_switch (162 lines). Two interrupts and one timer are used in plant. SPR has a significant effect, lowering the number of states stored by 93.44%. Again, OPR achieves better results by reducing the number of states by as much as 97.54%. The increase in the number of created states amounts to 13.96% for SPR, and 22.92% for OPR. Hence, compared to the according numbers for light_switch, the increase is rather modest. This means that either the model checker has to revisit fewer states, or that the length of the compressed paths is shorter. Memory consumption using any of the path reduction techniques dropped to the vicinity of the initial size of the hash tables.

In the next program, reentrance, a 16-bit integer variable is accessed concurrently in the main process and in an interrupt handler. As the ATmega has an 8-bit architecture, such accesses are non-atomic, thus leading to race conditions. The reduction in states stored achieved by SPR is 93.84%. OPR reduces the state space further by halving the number of remaining states, yielding a reduction of 96.92%. The increase in runtime due to revisits was 10.85% for SPR and 11.94% for OPR.

An automotive application was used as the fourth program. The program called window_lift implements the functionality of an electric window lift for cars. It is based on a state machine, which generates outputs depending on its current state. To fulfill its task, it uses three interrupts and one 16-bit timer. The state space for this program without any abstraction is rather large compared to the previous programs, consisting of more than 2.3 million states. SPR decreases this value by 92.2%, while OPR decreases it by 94.72%. The runtime required for model checking is approximately doubled, with a slight advantage for SPR. Memory consumption was reduced by 89.89% in case of SPR and by 92.05% in case of OPR.

Our fifth case study used the aforedescribed program vector (cp. Sect. 4.1). SPR resulted in a decrease of the number of states stored by 89.76%. OPR outperformed this by three orders of magnitude, reducing the number of states stored by 99.99%. This effectively reduced the amount of memory required for the state space from more than 11.5 GB (no abstraction) to 23.7 MB (OPR). SPR results in a reduction of 87.96% to approximately 1.4 GB, which can still render the program manageable for model checking on desktop computers. The time required when using SPR increased only by 1.55%, whereas OPR results in an increase of 533%.

Considering that far less states are stored using OPR, this increase is actually surprisingly low. An explanation for the large difference between the two approaches to path reduction is the different handling of loops. In order to guarantee termination of the state space generation in the presence of program loops, SPR has to assume at least one position in the loop to be breaking (cp. [13]). In our implementation, this position is indicated by the head of the loop. Thus, on each revisit of the program counter position of the head, SPR terminates the current chain and stores a state. OPR, on the other hand, does not have to store at the head of a loop (in fact, it is unaware of the existence of the program loop), unless it uses the *same program counter* approach for termination detection. Hence, OPR compresses loop iterations far more efficiently.

Program	**Options used**	**States stored**	**States created**	**Size [MB]**	**Time [s]**
`light_switch` 162 lines	none	4,268	6,296	21.6	0.42
	SPR	1,115	16,175	20.9	0.79
	OPR	494	25,223	20.7	0.88
`plant` 225 lines	none	130,524	135,949	52.28	2.33
	SPR	8,552	154,921	22.6	2.81
	OPR	3,205	167,114	21.4	3.03
`reentrance` 147 lines	none	107,649	110,961	44.4	2.60
	SPR	6,628	123,003	22.0	2.08
	OPR	3,312	124,207	21.3	1.40
`window_lift` 289 lines	none	2,342,564	2,589,665	633.9	47.59
	SPR	182,709	3,818,060	64.1	57.78
	OPR	123,585	4,123,385	50.4	59.49
`vector` 930 lines	none	47,477,797	48,419,003	11,508,422	772
	SPR	4,860,321	55,584,435	1,385,914	784
	OPR	4,656	496,768,475	23.7	4,891

Table 2
Effects of different path reduction techniques on five microcontroller programs

4.3 Influence of Formulae

So far, we examined the effect of OPR when checking the formula $AG\ TT$, which is *true* in every state. In this section, we will now evaluate the effects of OPR when checking actual formulae whose validity depend on variables. Since path reduction needs to store states when a transition influences the formula, we expect an increase in the size of the state spaces.

For the first examinations, we decided to use the `window_lift` program. The results are shown in Tab. 3. As described in Sect. 4.2, the program models an automotive electrical window lift, and is based on a state machine. The state machine is implemented using a global integer variable called `mode`, which is expected to assume only the values 0 to 6 at any time during execution. Hence, our first test was to check this using the formula

$$(1) \quad AG\,(\texttt{mode} \geq 0 \wedge \texttt{mode} \leq 6)\,,$$

which could be verified after 54.83s.

The second test was to verify whether the sequence of states assumed by `mode` satisfies a certain property. Whenever a sensor reports that there is an object stuck in the window ($\texttt{mode} = 5$), the window lift is expected to open completely ($mode = 6$) before allowing normal operation again ($\texttt{mode} = 0$), which can by specified by the formula

$$(2) \quad AG(\texttt{mode} = 5 \Rightarrow \\ \neg E\,(\texttt{mode} = 5 \wedge \neg\texttt{mode} = 6)\;U\;(\neg\texttt{mode} = 5 \wedge \neg\texttt{mode} = 6)).$$

The program `window_lift` contains a subtle error which prevents this property from being satisfied. The error is based on the simultaneous occurrence of two interrupts, which allows `mode` to skip the value 6. [mc]square correctly discovered this error and created a counterexample consisting of 912 states.

Our second test program for this case study was `plant`, also described in detail in Sect. 4.2. The first property to verify was, similar to `window_lift`, to verify that a global variable satisfies certain constraints, which can be specified by the formula

$$(3) \quad AG\,(\texttt{tank} \geq 0 \wedge \texttt{tank} \leq 4)\,,$$

which could also be verified by [mc]square. The second property was then to ensure the correct behavior of the plant in case of an emergency. For this purpose, [mc]square had to check the formula

$$(4) \quad AG(\texttt{PORTA} = 0x20 \Rightarrow AG\,\texttt{PORTA} = 0x20) \wedge EF(\texttt{PORTA} = 0x20),$$

which resulted in a counterexample with 1,643 states after expansion.

The truth value of all four formulae was the same compared to model checking without OPR. Since formulae (2) and (4) were violated, the model checker could prematurely stop the state space generation. Thus, the time for model checking and the size of the state space is not comparable to the

Program	Formula	States stored	States created	Size [MB]	Time [s]
`window_lift`	(1)	226,452	3,792,507	78.33	54.83
	(2)	11,665	207,964	24.52	3.64
`plant`	(3)	3,678	167,114	21.54	2.32
	(4)	77	1,839	20.67	0.6

Table 3
Influence of formulae on state space sizes

other case studies. For the verification of formula (1), we have an increase of 83.24% of the state space size, while the time decreased slightly, due to the smaller number of revisits. For formula (3), the increase of the state space size is negligible.

5 Related Work

Path reduction based on static analysis for the model checker Murφ was first described by Yorav and Grumberg [18]. This technique is used in a similar fashion in the Spin model checker [9] using a static analysis for its input language Promela. Spin uses an intraprocedural static analysis (using inlining), and compared to binary code, Promela is much simpler since (1) communication between concurrent processes can only be performed using distinguished statements and (2) it does not contain indirect control statements.

Later, Quiros [12] has adapted the approach of Yorav and Grumberg to a bytecode language used in a virtual machine. This bytecode language is similar to a parallel `while` language. This means that function calls are handled using inlining, communication is performed at certain program locations, and indirect control is not supported. Hence, SPR turns out to be effective for this domain. Our own prior work [14] adapts these earlier approaches to the domain of binary code verification by introducing tailored static analyses and revising breaking conditions for binary code.

Behrmann et al. [1] implemented a similar technique for the model checker Uppaal, which focuses on timed automata. Their approach is similar to our implementation of SPR: they decided for a static analysis of the control structure of the automata in order to obtain a so-called *covering set* of edges. This set is used in order to guarantee termination in case of loops in the state space. States that are targets of edges in the covering set have to be stored, which exactly corresponds to the breaking property used in SPR. As we have illustrated, this property can prove a significant disadvantage of SPR in the presence of long-running but terminating loops. Our contribution, OPR, can handle such loops without storing states in each iteration. Pelanek [11] conducted a survey of on-the-fly state space reduction techniques. He subsumes techniques preserving stutter equivalence under the term of *transition merging*. His survey, however, focuses on high-level representations.

Recently, Yang et al. [17] introduced dynamic path reduction for bounded model checking of sequential programs. However, even though their technique is named similarly, its purpose is to prune out infeasible executions paths introduced by nondeterministic conditionals, and thus, must not be confused with path reduction in the sense used in this paper. Their algorithm computes weakest preconditions and unsatisfiable cores using SMT solving. Thus, both their approach and their goals are fundamentally different from our work.

6 Concluding Discussion

6.1 Conclusion

This paper describes a new technique for dynamic path reduction and shows the predominance of this method over approaches based on static analysis for the specific application of binary code model checking. Further, it shows how counterexamples generated using this abstraction technique can be expanded in order to ease their comprehensibility. In terms of effectiveness, the OPR approach allows for formidable state space reductions, comparing it to static path reduction techniques. The smaller memory footprint, however, may lead to higher runtimes. Thus, OPR provides a technique that allows to trade runtime for memory.

6.2 Future Work

Another abstraction technique discussed by Yorav and Grumberg [18] is *dead variable reduction* (DVR), the key idea being to reset variables whose value is not going to be read in any subsequent program execution. However, DVR for binary code suffers particularly from the presence of indirect reads in binary code, where the source memory locations can often not be determined accurately using static analysis [14, Sect. 6.1]. Consequently, it will be of interest to evaluate if state space reductions as significant as those obtained through OPR can be achieved using an on-the-fly adaptation of DVR [10,15].

Acknowledgement

This work was supported by the DFG Cluster of Excellence on Ultra-high Speed Information and Communication (UMIC), German Research Foundation grant DFG EXC 89. Further, the work of Sebastian Biallas was supported by the DFG. The work of Jörg Brauer and Dominique Gückel was, in part, supported by the DFG Research Training Group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems* (AlgoSyn). We thank Bastian Schlich for sharing his thoughts on the ideas described in this paper.

References

[1] Behrmann, G., K. G. Larsen and R. Pelánek, *To store or not to store*, in: *Computer Aided Verification (CAV 2003)*, LNCS **2725** (2003), pp. 433–445.

[2] Browne, M., E. Clarke and O. Grumberg, *Characterizing finite kripke structures in propositional temporal logic*, Theor. Comput. Sci. **59** (1988), pp. 115–131.

[3] Chaki, S., A. Groce and O. Strichman, *Explaining abstract counterexamples*, in: *SIGSOFT FSE*, 2004, pp. 73–82.

[4] Clarke, E. M., "The Birth of Model Checking," Springer-Verlag, Berlin, Heidelberg, 2008, 1–26 pp.

[5] Clarke, E. M., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Progress on the state explosion problem in model checking*, in: *Informatics - 10 Years Back. 10 Years Ahead*, LNCS **2000** (2001), pp. 176–194.

[6] Clarke, E. M. and H. Veith, *Counterexamples revisited: Principles, algorithms, applications*, in: *Verification: Theory and Practice*, LNCS **2772** (2004), pp. 41–43.

[7] Cook, B., A. Podelski and A. Rybalchenko, *Termination proofs for systems code*, in: *PLDI* (2006), pp. 415–426.

[8] Heljanko, K., *Model checking the branching time temporal logic CTL*, Research Report A45, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland (1997).

[9] Holzmann, G. J., *The engineering of a model checker: The Gnu i-protocol case study revisited*, in: *Theoretical and Practical Aspects of SPIN Model Checking (SPIN 1999)*, LNCS **1680** (1999), pp. 232–244.

[10] Lewis, M. and M. Jones, *A dead variable analysis for explicit model checking*, in: *PEPM* (2006), pp. 48–57.

[11] Pelánek, R., *On-the-fly state space reductions*, Technical report, Masaryk University Brno, Czech Republic (2005).

[12] Quirós, G., "Static Byte-Code Analysis for State Space Reduction," Master's thesis, RWTH Aachen University (2006).

[13] Schlich, B., "Model Checking of Software for Microcontrollers," Dissertation, RWTH Aachen University, Germany (2008).
URL http://aib.informatik.rwth-aachen.de/2008/2008-14.pdf

[14] Schlich, B., J. Brauer and S. Kowalewski, *Application of static analyses for state space reduction to microcontroller binary code*, Sci. Comp. Program. (2010), to appear.

[15] Self, J. P. and E. G. Mercer, *On-the-fly dynamic dead variable analysis*, in: *SPIN*, LNCS **4595** (2007), pp. 113–130.

[16] van Glabbeek, R. and W. Weijland, *Branching time and abstraction in bisimulation semantics*, Journal of the ACM **43** (1996), pp. 555–600.

[17] Yang, Z., B. Al-Rawi, K. Sakallah, X. Huang, S. Smolka and R. Grosu, *Dynamic path reduction for software model checking*, in: *IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods* (2009), pp. 322–336.

[18] Yorav, K. and O. Grumberg, *Static analysis for state-space reductions preserving temporal logics*, Formal Methods in System Design **25** (2004), pp. 67–96.