Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Selected Dynamic Issues in Software Model Checking[⋆]

Viet Yen Nguyen[1], **Theo C. Ruys**[2]

[1] Software Modelling and Verification Group, RWTH Aachen University, Germany, e-mail: `nguyen@cs.rwth-aachen.de`
[2] RUwise Consultancy, Deventer, The Netherlands, e-mail: `theo.ruys@gmail.com`

Received: date / Revised version: date

Abstract Software model checking has come of age. After one and a half decade, several successful model checking tools have emerged. One of the most prominent approaches is the *virtual machine* based approach, pioneered by Java PathFinder (JPF). And although the virtual machine based approach has been rather successful, it lags behind classic model checking in terms of speed and memory consumption. Fortunately, with respect to the implementation of virtual based model checkers, there is still ample room for innovation and optimizations.

This paper presents three novel (optimization) techniques that have been implemented into MoonWalker, a software model checker for .Net programs.

– .Net specifies an exception handling mechanism called structured exception handling (SEH). SEH is one of the most sophisticated and fine-grained exception handling mechanisms for application platforms. Its implementation within MoonWalker is the most sophisticated in a model checker to date.
– To decrease memory use within MoonWalker, a collapsing scheme has been developed for collapsing the metadata used by *stateful dynamic partial order reduction*. The reduction of memory is – in some cases – more than a factor of two.
– Finally, to decrease the verification time, the Memoised Garbage Collection (MGC) algorithm has been developed. It has a lower time-complexity than the often used Mark & Sweep garbage collector. Its main idea is that it only traverses changed parts of the heap instead of the full heap. The average time reduction is up to 25%.

We have used the Java Grande Forum benchmark suite to compare MoonWalker against JPF and observed that the average performance of MoonWalker is on par with JPF.

[⋆] The MoonWalker model checker has been developed within the Formal Methods & Tools group of the University of Twente, The Netherlands.

1 Introduction

Software has become a normal part of our daily lives. People are working with software more than ever before, most of the times without even being aware of it. The complexity of the design and implementation of a software system also has grown rapidly in the last decade. Televisions and mobile telephones are more complex and provide more functionality than a high-end workstation did less than a generation ago. Unfortunately, with the impressive advancement of technology and functionality comes unforeseen failure. The list of famous computer errors (better known as *bugs*) is long, and several had extreme consequences.

A mechanical and mathematically inspired technique that is specifically successful in finding errors in (functional designs of) software is called model checking. Model checking [10, 4] is the *formal verification* of a *model* ($\mathcal{M}$) against a *specification* (p). The verification algorithm is an automatic and systematic process that formally concludes whether $\mathcal{M} \models p$ or $\mathcal{M} \not\models p$. Models are specified in abstract, high-level specification languages, e.g., PROMELA [24]. The property p is specified in a temporal logic, e.g. LTL. The idea is to first use a model checker to formally verify the design of a software system, and then afterwards, refine the (now correct) verification model to a computer program. This original approach of applying a model checker to a high-level model $\mathcal{M}$ is referred to as *classic* model checking.

Within the software development cycle, model checkers are often used to validate the initial design of a system before actually implementing it. The process of model checking usually consists of three parts: modelling, specification and verification. During the modelling phase, an abstraction is made from the design under verification. This abstraction – the model – is then verified against the specification. This traditional approach has its disadvantages. For, (i) creating an abstraction at the right level is considered difficult, (ii) the abstraction is crafted manually and prone to human error, (iii) the model and its semantics are bounded to the expressiveness of the modelling language, which generally tend to be rigourously formalised (e.g., process algebra's, state machines) [12] and (iv) after validation, the model still has to be transformed to an implementation. As fully automated code generators do not exist (yet), parts of this refinement step have to be done manually.

Software model checking overcomes the labor intensive problems of classic model checking by verifying the implemented system directly instead of the abstract model. This approach has been pioneered by Klaus Havelund [21] in the first version of Java PathFinder (JPF). This initial version of JPF comprised a Java to PROMELA translator that enabled the verification of Java programs using the model checker SPIN [24]. This experiment highlighted many challenges associated with the software model checking approach. The foremost problem was overcoming the semantic gap between Java and PROMELA. This was solved by introducing the bytecode interpretation approach in the second iteration of JPF [43,49]. Ever since, more techniques have been developed to make software model checking more effective [30]. These developments have given rise to other software model checkers, like XRT [20], BOGOR [39], BANDERA [11,12,46] and MOONWALKER [1,40,51]. The latter is a software model checker developed at the University of Twente.

With respect to speed, software model checkers are roughly two orders of magnitude slower than state-of-the-art model checkers that use special modelling languages. For instance, for SPIN it is not uncommon to analyse more than 10^6 states per second. By comparison, a software model checker like JPF typically analyses less than 10^4 states per second. There are several reasons for this difference in speed between *classic* and *software* model checkers. For example, the average size of the state descriptor, i.e., the state vector, is different for classic and software model checkers. The size of the state vector for a medium size PROMELA model is between 50-200 bytes. For a small Java/C# application, the state vector easily grows to more than 10^3 bytes. Several algorithms within the model checker (hashing, state matching) are linear in the size of the state vector. Another reason for the difference in speed is the dynamic nature of object-oriented software. Objects are created on the heap. After each transition, the software model checker has to start the garbage collector to get rid of unreachable objects. Garbage collection is a expensive operation. But that is not all. If you take into account the granularity of transitions, software model checkers appear to be even slower. Without partial order reduction, a transition within in software model checkers consist of a single, low-level bytecode instruction. A transition in a PROMELA model is much more high level. For example, an assignment statement in PROMELA is always atomic and will result in a single transition in SPIN. In Java/C#, a (complicated) assignment statement could be compiled to several bytecode instructions, which result in several transitions and intermediate states.

Apart from these speed issues, there is always the intrinsic memory problem of (software) model checkers. Due to the state space explosion, for medium to industrial-size models, model checkers typically run out of memory: there is simply not enough memory to store all states. This applies especially to software model checkers, where states are big and the transitions between states are relatively small. Memory runs out before a substantial part of the system could have been analyzed.

So although there are structural differences between classic and software model checkers which are hard to overcome, there is still much room for improvements in the implementation of software model checkers. In this paper we describe three dynamic techniques that helped us to improve the performance of the MOONWALKER model checker. As such, this paper is a follow up of our papers on software model checking at TACAS 2009 [1,35]. In this paper, we expand on the architecture and design on MOONWALKER of [1] and expand on the memoised garbage collection of [35]. More importantly, we present two novel dynamic techniques for software model checkers: an efficient implementation for structured exception handling and a collapsing scheme to reduce the memory footprint of stateful dynamic partial order reduction.

Outline of the paper. First, Section 2 discusses related work in the realm of software model checking. Section 3 gives a high-level architectural description of MOONWALKER. It outlines the most important aspects of MOONWALKER's implementation, which are needed to understand the remainder of the paper. The predominant part of the paper, however, is devoted to three novel (optimization) techniques that have been implemented in MOONWALKER 1.0. Before discussing these optimization techniques, Section 4 describes the benchmark suite that has been used to evaluate the effectiveness of the techniques. As we evaluate the techniques also in isolation, we have chosen to discuss the benchmark suite before the actual techniques. Section 5 then describes MOONWALKER's implementation of handling exceptions. Section 6 discusses MOONWALKER's solution to reduce the memory footprint incurred by stateful extensions to dynamic partial order reduction. The material in Sections 5 and 6 has not been published before, other than in [34]. Section 7 describes MOONWALKER's memoised garbage collection algorithm, and has been published in a shortened form in [35]. Section 8 compares MOONWALKER to another software model checker in its class: JPF. Section 9 summarises this paper and describes several ideas for future work.

2 Related Work

Formal tools for testing and verification of object code, and that of .NET programs in particular, have gained increased attention in both academia and industry. For testing for example, PEX [42] is known for its automatic generation of test suites for .NET programs. Testing is however only suitable for sequential programs. For multi-threaded programs, the test-cases generated by PEX do not account for the possible interleavings and have as a result a dramatically reduced coverage. Context-bounded verification [36] tries to improve this by accounting for interleavings, but only to a given upperbound of context-switches between threads. The result is only a partial coverage of a multi-threaded program. Bounded model checkers, like CBMC [8], also achieve partial coverage by accounting for all interleavings up to an user-defined depth. In this paper however, we emphasise *software* model checkers that ensure *full coverage* for multi-threaded programs. In this line, three types of model checkers can be distinguished.

– *Translation based checkers.* In this approach, the source program is translated to the input language of some existing model checker. An example of this approach is the pair of tools: MODEX [26] and SPIN. MODEX (perhaps better known as its predecessor FEAVER) is a tool that can be used to mechanically extract high-level verification models from implementation level C code. These verification models can then be verified with SPIN. Another example of this approach is the first version of the Java PathFinder [21], that translated Java programs to PROMELA models, which were then verified with SPIN.
– *Abstraction based checkers.* An abstraction tool constructs an abstract (over-approximated) model of the original source program. This abstract model is subsequently analyzed. An example of this approach is SLAM [5,53], developed at Microsoft Research which does reachability analysis of sequential C programs. Another successful tool following the abstraction approach is BLAST [22,6,47].
– *Bytecode checkers.* Bytecode checkers are built around the virtual machine of some intermediate representation (in this context called bytecode). The effect of every intermediate instruction is analyzed by the virtual machine based checker. Such bytecode checkers indirectly analyze the complete full program: no abstractions are necessary.

MOONWALKER follows the bytecode based approach to software model checking. Below we briefly discuss some important software model checkers that also use this approach.

JPF. Java PathFinder (JPF) [43] is a very successful software model checker for Java bytecode. It pioneered the concept of implementing a software model checker as a virtual machine that simulates the (binary) code of the application to be checked, i.e., simulating the Java Virtual Machine (JVM). JPF is an explicit-state model checker that systematically explores the state-space of a Java program, thereby generating it on-the-fly. It reduces the size of the state-space by applying partial order reduction techniques [19], as well as (heap) symmetry reduction [9,39]. The size of each individual state is reduced using the recursive indexing method [23]. By systematically exploring the state-space the JPF aims to find deadlocks and uncaught exceptions. JPF is an open source application and is available from [49].

XRT. At Microsoft Research, XRT [20], an exploration framework for .NET has been developed. XRT is a state exploration framework that follows similar goals as BOGOR and JPF, using the approach of execution on the virtual machine level as pioneered by JPF. It supports the full safe (verifiable) CIL, and provides extension points on various levels, including the instruction set, the state representation, and the exploration strategy. It has been developed from the beginning together with one particular extension in mind, namely the unrestricted support of mixed concrete/symbolic state and exploration.

Within XRT, instructions are represented by a language called XIL, which is an abstraction of CIL. Instead of the stack-based virtual machine of CIL, the virtual machine for XIL is a register machine. Although it means that a new virtual machine had to be developed, the use of a register language as an intermediate language opens new opportunities for optimizations. Instruction rewriters are being used to alter the code of methods. XRT is currently not publicly available.

BOGOR [39] is another successful software model checking approach. BOGOR accepts programs written in the BOGOR Input Representation (BIR) bytecode. The correlation between BIR and Java is very high, making BOGOR an obvious choice for the verification of object-oriented programs. There exist modules for BOGOR that perform symmetry reductions as well as partial-order reductions. Furthermore, BIR is an extensible language. It is possible to introduce new language features to extend BIR's syntax. More specifically, one can introduce new native types, and define operations on such types. The model checking framework of BOGOR is also extensible. An advantage of JPF and XRT over BOGOR is that they work directly (but *only*) on standardized bytecode formats: Java resp. CIL bytecode. BOGOR is much more of a model checking framework that can be used to build a sophisticated model checker than an off-the-shelf model checker like JPF and XRT. BANDERA [11,12,46] for example is a toolchain that translates Java bytecode to BIR, uses BOGOR to do the model checking and then maps counterexamples, if any, back to Java source code level. BOGOR is available from [48].

3 MoonWalker

In this section we briefly discuss the architecture and design of MOONWALKER model checker. The optimization techniques that are presented in the later sections are all implemented in MOONWALKER.

MOONWALKER[1] is a software model checker for the verification of CIL bytecode programs. CIL stands for Common Intermediate Language and is the platform independent bytecode used within Microsoft's .NET. MOONWALKER targets programs compiled against the MONO development platform [50], an open source implementation of the .NET development platform. MONO offers a free and open implementation of the Common Language Infrastructure (CLI) published by ECMA as standard 335 [15].

[1] MOON is an anagram of MONO. WALKER refers to the nature of the tool: walking the state space of CIL bytecode programs.
MOONWALKER was previously known as MMC: the Mono Model Checker. Due to several name clashes – e.g. the Mobility Model Checker for the pi-calculus [44], Microsoft's Management Console, the Mickey Mouse Club – the name has been changed to MOONWALKER.

feature	0.5	1.0
Number of CIL instructions supported	58	74
Verification of deadlocks and assertion violations	√	√
Structured state collapse compression (recursive indexing)	√	√
Backtracking between consecutive states using small deltas	√	√
Heap canonicalisation using symmetry reductions	√	√
Reference counting garbage collection	√	
Mark & Sweep garbage collection	√	√
POR by distinguishing thread-safe and thread-unsafe instructions	√	√
POR using object escape analysis		√
Stateful DPOR		√
Collapsing of interleaving information for stateful DPOR		√
Memoised garbage collection		√
Structured exception handling mechanism		√
Error tracer		√
Extensive testing framework		√

Table 1. Most important features of MOONWALKER 0.5 and 1.0.

The development of MOONWALKER has been initiated to get experience with the design and implementation of a software model checker. The core of MOONWALKER was intended to serve as a sandbox for further research on software model checking. Originally aimed as a proof-of-concept prototype, over time MOONWALKER developed into a competitive model checker for CIL bytecode programs, that can be used by C# programmers to verify their (multi-threaded) code. The intended audience for MOONWALKER are software developers who want to contribute to and experiment with a software model checker for .NET. Consequently, extensibility and simplicity of design have been a prime concern. Furthermore, readability and reusability of the implementation have been important objectives.

MOONWALKER uses the *virtual machine* approach of verification: the effect of every CIL bytecode instruction is analyzed by the tool. MOONWALKER systematically explores all reachable states of the application under verification, which involves executing bytecode instructions, storing and restoring states, and checking for safety properties. During exploration, MOONWALKER will check for deadlocks and assertion violations. MOONWALKER does not (yet) support temporal logic model checking.

The approach and design of MOONWALKER are based and inspired by the Java PathFinder (JPF), which pioneered the virtual machine approach. Although the object-oriented design, the actual implementation of MOONWALKER (in C#) and organisation of the classes and algorithms are different, all credits for the verification approach should go to the developers of JPF.

With MOONWALKER 1.0, however, there is now a competitive software model checker readily available for the .NET framework. It benefits from CIL's language-agnosticism: CIL is not only the target of C#, but also for functional languages like F# and declarative languages like Prolog.NET. See [52] for a complete overview. This means that MOONWALKER is not just a model checker for C#, but can in principle be applied to programs written in many different programming languages. Finally, version 1.0 of MOONWALKER incorporates some new techniques not yet available in other model checkers, which will be discussed in later sections of this paper.

Table 1 lists the most important features of MOONWALKER version 0.5 [40] and MOONWALKER version 1.0 [1].

Implementation. The current version of MOONWALKER is version 1.0.1. The total development of the tool took roughly two man years of work. The code base of MOONWALKER 1.0.1 consists of 17k lines of C# code and constitutes 475Kb of source code. Both a binary and source distribution are available from [51].

MOONWALKER 1.0.1 supports 74 CIL bytecode instructions. These are all possible instructions that can be emitted by MONO's C# 1.0 compiler. Support for the last nine missing instructions is future work.

3.1 Architecture

MONO applications – CIL bytecode programs – are run on a virtual execution system (VES). We have adopted the concept of implementing a model checking virtual machine (VM), capable of systematically exploring the state space of a software application. The exploration is performed by iteratively executing instructions that are read from the compiled CIL bytecode. To have full control over this process, we manage all VM structures ourselves. The client application does not see any difference between running on MONO or on MOONWALKER, i.e., MOONWALKER behaves exactly the same as the MONO VES would. This is made easier by a large extend by the fact that MOONWALKER itself runs on the MONO VES. Calls and operations can be 'passed down' by MOONWALKER by performing exactly the same action as the client application.

Figure 1 gives a schematic overview of the architecture of MOONWALKER, its most important building blocks and interaction between those blocks. Central to the architecture is the *Explorer*. This component drives the state space exploration.

On the right-side of the figure we find the parts of MOONWALKER that are used for storing and restoring states: the state storage and a backtrack stack. On the left-hand side of Fig. 1, the part of the MOONWALKER that provides the virtual execution environment is depicted. It is based on two components: instruction executors and an active state.

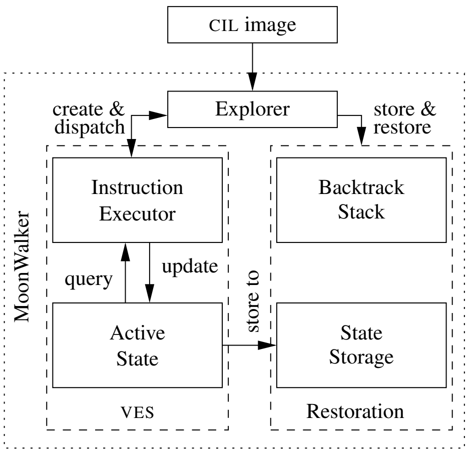


Figure 1. MOONWALKER architecture

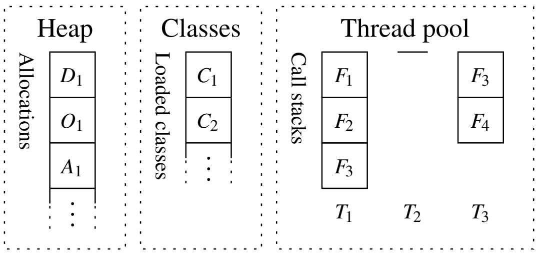


Figure 2. Active state of the virtual machine.

The *instruction executors* (IEs) are objects responsible for executing the CIL instruction. There are many IEs: one for each type of CIL instruction. Each IE is implemented by its own C# class. This approach closely resembles the *command* design pattern [18]. The merit of using the command pattern in MOONWALKER is that code to execute CIL-instructions can be seen as first-class-citizens. This allows us to add more CIL-instructions to MOONWALKER without modifying existing code. Furthermore, meta-data (such as the list of exceptions an instruction can throw, the safety of instructions, etc.) that is associated with the instructions can be added to the executor classes, thereby eliminating the need for big look-up structures. The construction and invokation of the executors is done by the explorer.

The *active state* holds the current state of the VES of MOONWALKER. The active state is being queried and updated by the instruction executors, and it can be stored in the state storage. Although at a design level the active state is not that hard to grasp, it is its size and complexity that gave us a hard time managing the structure at run-time in an efficient way.

Fig. 2 shows an example of an active state. Three components can be identified, i.e. the heap, classes and thread pool. The *heap* holds all dynamic allocations: objects, arrays and delegates. The static fields are stored in the *classes* component. The *thread pool* contains the concurrent processes that are currently running. For each process it contains the complete stack of method calls by that process.

Algorithm 1: Explorer()

```
Data: backtrack stack $B$, state matcher $M$, state $s$
$thread \leftarrow 0$
$s \leftarrow$ initial state
repeat
    // 1. Forwarding
    $s \leftarrow$ ExecuteStep($thread$, $s$)
    // 2. Error checking
    if $s$ violates assertion $\vee$ $s$ deadlocks then
        break
    // 3. State matching
    if $s$ is matched by $M$ then
        $Working(s) \leftarrow$ empty set
        OnSeenState($s$)                              // hook
    else
        $Working(s) \leftarrow Enabled(s)$
        update $M$ to match $s$
        OnNewState($s$)                               // hook
    // 4. Backtracking
    while ($Working(s)$ is empty) $\wedge$ ($B$ is not empty) do
        pop $s$ from $B$
        Backtracked($s$)                              // hook
    // 5. Next transition decision
    if $Working(s)$ is not empty then
        push $s$ on $B$
        $t \leftarrow$ a removed transition from $Working(s)$
        add $t$ to $Done(s)$
        ThreadPicked($s$, $Thread(t)$)                // hook
until $B$ is empty
```

3.2 Explorer

As stated previously, the Explorer's task is to systematically drive the exploration algorithm. The explorer uses a depth-first-search (DFS) strategy to visit all reachable states. To detect cycles in the exploration graph, visited states are stored in the *state storage*, and each new state is compared to all stored ones. Backtracking is done by keeping a *backtrack stack* of the states that form the current path being explored. During exploration, MOONWALKER will check for deadlocks and assertion violations. A deadlock is a state where there are no runnable processes in the system, but not all processes have terminated. An assertion is a user-defined condition that has to hold in a certain state.

The pseudo algorithm of the Explorer is shown in algorithm 1. It is a stateful depth-first search explorer. Additional features like heap symmetry reduction, partial order reduction and state collapsion and backtracking by deltas, are left out of this algorithm. They will be discussed later in this section. The algorithm uses the following disjoint sets:

- $Working(s)$, is the set of transitions that have not been explored yet in state s.
- $Done(s)$, is the set of transitions that have already been explored in state s.

1. Forwarding. Initially, the explorer is engaged by picking thread 0, the main thread, as the first thread to forward. One forward step by `ExecuteStep` is called a normal step. A normal step explores one transition, followed by zero or more thread-safe instructions by the same thread. A *thread-safe* instruction is an instruction whose execution is not visible (or relevant) to any of the other threads. An *thread-unsafe* instruction is an instruction that *might* influence other processes. The *merging* of thread-safe instructions into one transaction can be considered a mild form of partial order reduction [19]. SPIN employs a similar technique called statement merging [24]. Note that within MOONWALKER instruction merging is not a feature, but rather a necessity. Without instruction merging the number of states would soon become unmanageable.

2. Error Checking. After a forward step, a successor state is reached that becomes the current state. It is checked for assertion violations and deadlocks. Upon an error, MOONWALKER will by default stop exploration and pass control to the error tracer, which will output the execution trace leading to the error state.

3. State Matching. This phase checks whether the state has already been visited. If it is, the working set is set to the empty set, ensuring that the explorer will backtrack in the next phase. If the state is new, the working set is populated with the enabled set and the state matcher is updated such that it will match the new state.

4. Backtracking. Backtracking only happens if the working set is empty and if there are states to backtrack to. The working set can be empty for two reasons: (i) either the state is a revisited state or (ii) the state is an endstate. In both cases, the explorer backtracks. A backtrack operation pops off the top state from the backtrack state. This is repeated until a state is found on the backtrack stack that has a non-empty working set.

5. Next transition decision. The next transition decision phase can only be entered when the working set is non-empty, as a transition from the working set has to be chosen to forward. The working set is empty if all states in the state space have been explored. MOONWALKER will then stop exploration and quit. Otherwise, a transition is chosen from the working set and the explorer jumps back to the forwarding phase. Currently, the choice of a transition is based on the ordering of threads. The enabled thread with the lowest thread identifier is chosen first.

Several hooks (see the hook comments in algorithm 1) have been added to the exploration algorithm. They are used for error tracing, stateful dynamic partial order reduction (see Section 6) and logging exploration statistics. These hooks separate the source code of these features from the exploration code. This improves the readability and the overview of their respective sources.

3.3 Heap Symmetry

The heap is the part of the (active) state where dynamic allocations are stored. It is used often, since many modern (object-oriented) programming languages like C# use dynamic allocations (objects) for everything more complex than a simple number. MOONWALKER's explorer needs to be able to check two heaps for equivalence (i.e., phase 3 in Alg. 1). This is a bit cumbersome due to the highly dynamic nature of the heap. As a solution, MOONWALKER uses the same heap symmetry reduction technique as implemented in both JPF and XRT: when a new allocation is created on the heap for the first time, we remember where it has been put. The next time we create the *same* allocation, we put the object in the same place. In this way, the order in which the allocations are created is not longer relevant and always yields the same heap. *Heap symmetry reduction* is then achieved by canonicalisation of the heap, and this canonicalised representation is used for storage and matching.

To date, two variants of heap symmetry reduction are known to be effective. The technique of Iosif [27] traverses the full heap graph and creates a canonical array of objects out of it. This canonical array is stored in the hashtable. Upon state matching, the state to be matched is canonicalised and then the canonicalised arrays are matched. The technique of Lerda and Visser [30] maintains a canonicalised array, instead creating one when necessary. The latter is employed by MOONWALKER. Both techniques rely on the garbage collection algorithm to function. A heap graph traversal is needed for purging unreferencable, i.e., garbage, objects. This stems from an important observation by Iosif and Sisto that garbage objects may differ between states that have different paths leading to them, but are equivalent when canonicalised [29].

3.4 Partial Order Reduction

Partial order reduction (POR) may reduce the state space to be explored, while maintaining correctness of the verified specification. As mentioned before, MOONWALKER already performs a limited form of POR by merging thread-safe instructions as one step. We have implemented two much more effective POR techniques in MOONWALKER, namely POR using object escape analysis [19,30,14] and dynamic POR [17]. Our approach is based on the principles outlined in those papers. This section briefly summarizes both techniques.

A typical state space contains many paths that are semantically equivalent with respect to the specification. Yet, algorithm 1 explores all these paths anyway. The idea behind POR is to detect the semantically equivalent paths and then explore only one of them. POR is therefore also described as model checking using representatives. The notion of dependency and independency is central to POR. Concurrent commutative *independent* transitions lead to the same state when executed in different orders. Semantically equivalent paths are then paths which only differ in the total order of independent transitions. Transitions that are not independent are *dependent*. In practice, dependency is approximated using static analysis.

With POR, instead of exploring all transitions of the enabled set, only a subset of this enabled set is explored: a persistent set. A subset T of $\mathit{Enabled}(s)$ is called *persistent* in s if all transitions not in T that are enabled in s, or in a state reachable from s through transitions *not* in T, are independent with all transitions in T [19]. Of course, the key element of POR is the algorithm for computing persistent sets.

Object Escape Analysis. In MOONWALKER, transitions in the state space are ensured to be independent if they access an object that is only reachable by one thread in the object graph. Such objects are called thread-unshared objects. An object escape analysis algorithm determines which objects 'escape' their thread-local context, and therefore become thread-shared objects. It does this by setting an attribute for each object, that can hold either `UNMARKED`, a thread identifier or `SHARED`. The algorithm itself consists of two phases.

In the first phase, all objects are traversed. First, each object is initialised to `UNMARKED`, indicating it is unreachable from the callstacks. Then the callstacks are traversed, setting every object with the thread identifier associated with the callstack referenced from. This also indicates that the object is reachable. In case an object is referenced from two different callstacks, it is promoted to the `SHARED` status.

In the second phase of the algorithm, the objects are recursively traversed and the attributes are propagated. A child object is promoted to `SHARED` in case the propagated attribute is different than its own. This happens if the propagated attribute is a different thread identifier than the currently set thread identifier, or the current object is not yet shared, but its parent is.

After the algorithm has finished, objects that are attributed with `UNMARKED` are unreachable and can be garbage collected, objects that are attributed `SHARED` are thread-shared objects and objects that are attributed by a thread identifier are thread-unshared objects. Note that an assumption is made here that escaped objects are automatically assumed thread-shared. This is a coarse, but safe, assumption. After exploration of the full state space, it could turn out that all accesses to a thread-shared object were after all made by only one thread.

Once all objects are marked either thread-shared or thread-unshared, we use that information to calculate the persistent set. If the enabled set contains any transition that accesses a thread-unshared object, that transition is ensured to be independent with all other transitions in the enabled set and can therefore be used as the singleton persistent set. If no such independent transition is found, the whole enabled set is used to populate the persistent set. This for ensuring correctness.

Dynamic POR. A POR technique that differs from POR using object escape analysis is called POR using dynamically tracked dependencies, also called dynamic POR (DPOR). It is a more recent POR technique by Flanagan & Godefroid [17]. This original description of DPOR worked only correctly for stateless exploration.

The POR approach with object escape analysis is not always precise. For nearly all models, the object escape analysis makes too much assumptions. An often wrong assumption, but made for correctness, is that all child objects of a thread-shared object are also thread-shared. The dynamic variant of POR does not use object escape analysis, but analyses transitions on the backtrack stack for dependencies. These dependencies can be made at the level of field entities, making the dependency relation much more precise.

DPOR assumes that at each newly visited state, the current optimal persistent set is a singleton. The singleton can be any arbitrary transition from the enabled set. This singleton is then explored. Upon further exploration of a state, dependencies between transitions in subsequent enabled sets and explored transitions on the backtrack stack are determined. Two transitions are dependent if they access the same field entity and if no intermediate transition accesses it as well. Using this dependency relation, the backtrack stack is traversed to inject dependent transitions in the working sets. Thus, contrary to POR using object escape analysis, persistent sets are constructed afterwards and not beforehand.

More formally, at the visit of a newly visited state s, the associated enabled set is traversed. Each transition $t \in \mathit{Enabled}(s)$ is checked for its dependency with a transition t' from state s' on the backtrack stack by matching the field entity they access. If they are found dependent, a transition $t'' \in \mathit{Enabled}(s')$ by the same thread as t is injected into the $\mathit{Working}(s')$. If $\mathit{Thread}(t)$ is not enabled in s', then the whole enabled set is added to the $\mathit{Working}(s')$.

POR using object escape analysis and DPOR can be combined. In case the object escape analysis does not reveal a singleton persistent set, the whole enabled set is then used as the working set. If DPOR is activated as well, the DPOR clears the working set and only adds one transition from the enabled set to it. This transition is then traversed. Dependent transitions determined from the state space below may update the working set by adding elements from the enabled set to it.

3.5 State Storage

All states that are visited by the explorer are stored. The storage must be efficient in order to deal with the infamous state space explosion. Software model checkers additionally need to cope with the large state vectors as well, as individual VM states can be become big as well. To reduce the size of the states that are being stored, MOONWALKER uses a technique called *recursive indexing* (or *collapse compression*), which is also used in SPIN [23] and JPF.

The principle of recursive indexing is as follows. There is a data structure called *pool* that stores objects. Each stored object (O) is assigned a *unique* indexing number by the pool ($\mathcal{P}$). That is, the indexing number of two objects is the same if and only if the two objects are the same. Once an object is stored, it is never removed, and once assigned, the index number remains the same. Assume we have a part of the state that consists of several objects in a fixed order: $L_1 = [O_1, O_2, \ldots O_n]$. We store these n object in pool $\mathcal{P}$, and replace all objects by their respective index numbers in $\mathcal{P}$, giving us the list of numbers $C_1 = [\mathcal{P}(O_1), \mathcal{P}(O_2), \ldots, \mathcal{P}(O_n)]$. We call this translation *collapsing*. There is no loss of identity, since every object has a unique number. Figure 3 shows the state storage organization of MOONWALKER. The left-hand side of the figure shows the 'base' scenario which uses the collapsing method.

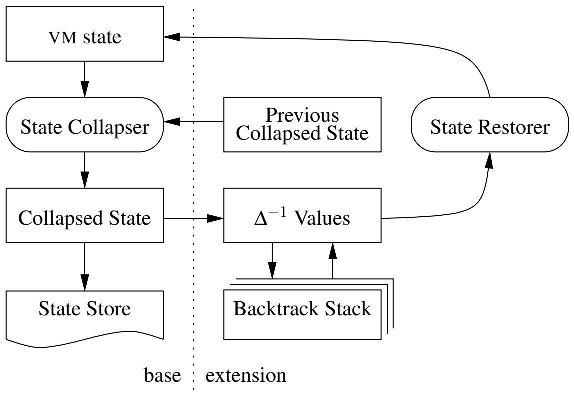


Figure 3. State Storage.

Although the collapse method is quite good at compressing the state there is still an aspect that is not optimal. Every time we have to check whether we have seen a state, the state first has to be collapsed in order to compare it to other states seen. This is mostly redundant work, because two consecutive states S_1 and S_2 do not differ that much. In other words, many of the objects that S_1 and S_2 consists of are the same. The solution to this problem is straightforward. We keep a copy of the previous collapsed state S_1 and only collapse the parts of S_2 that differ from S_1, i.e. those that have actually changed. The right-hand side of Fig. 3 shows this extension of the data flow. Note that although conceptually simple, this involves keeping track of all changes in the active state as code is being executed by MOONWALKER's virtual machine.

This *delta* between collapsed states proved also useful and effective in other parts of the explorer. As MOONWALKER uses a DFS strategy to visit all reachable states, it has to *backtrack* when it encounters a state it has already visited. Instead of storing all (collapsed) states on the backtrack stack, MOONWALKER follows SPIN's (and JPF's) approach and stores the reverse transition between two consecutive states on the stack. Given a state S_2, this reverse transition can be used to rebuild the previous state S_1. This reverse transition is called *reverse delta*, depicted by Δ^{-1}. The right-hand side of Fig. 3 shows how the Δ^{-1} values are computed and used by the explorer when backtracking.

4 Experimental Evaluation

The techniques described in this paper were evaluated using experiments. This section describes the used models, the applied parameters and the configuration of the hardware. Additionally, at the end of this paper, we present a comparative analysis of MOONWALKER 1.0 and JPF's trunk from 10 October 2007. Our comparison originally included BANDERA's verification times using BOGOR as a backend, but was left out of the comparison in this paper. The rationale behind this is discussed in section 8. The remainder of this section describes the experimental setup.

4.1 Bandera's Examples

Initially, we took three models from Bandera's repository of examples, namely Pipeline, SleepingBarbers and BoundedBuffer and manually porting them to C#. However, after running these academic examples through MOONWALKER, we found them unsuitable for our comparison. The techniques presented in this paper would only show their advantage on model with big heaps and long verification times. The three small examples have either short verification times (around a second) and/or very small heaps. We also purposely did not create our own benchmarks, of which the results may be interpreted with bias.

4.2 Java Grande Forum Benchmarks

Since Bandera's examples did not suffice due to their small and academic nature, we diverted to a different source for representable benchmarks that forces a software model checker to cope with the large state spaces. We chose to use an existing benchmark suite developed for the scientific community called the Java Grande Forum Benchmarks (JGF benchmarks) [41]. One part of this suite are the parallel benchmarks, which are medium-sized multithreaded applications for evaluating emerging parallel programming paradigms in Java and to expose their weaknesses. There are three parallel benchmarks, of which we used two:

- MOLDYN "is an $O((N*(N-1))/2)$ N-body code modelling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The computationally intense component of the benchmark is the force calculation, which calculates the force on a particle in a pair wise manner. This involves an outer loop over all particles in the system and an inner loop ranging from the current particle number to the total number of particles. The outer loop has been parallelised by dividing the range of the iterations of the outer loop between processors, in a cyclic manner to avoid load imbalance." [41].
- RAYTRACER "measures the performance of a 3D ray tracer. The scene contains 64 spheres and is rendered at a resolution of $N \times N$ pixels. The outermost loop (over rows of pixels) has been parallelised using a cyclic distribution for load balancing." [41].

The third benchmark is the MONTECARLO benchmark. It however uses file input/output, which is not (yet) supported by MOONWALKER and hence is left out for the comparison.

The benchmark are parametrised with two parameters: the number of threads and the data size. Throughout this paper, a configuration of these parameters is denoted by t-d, where t is the number of threads and d is the datasize. For MOLDYN, the datasize means the number of particles that is simulated. For RAYTRACER it means the number of pixels in both width and height that is rendered. An increased t and an increased d will lead to a larger state space. The largest explored state space we encountered (using RAYTRACER 3-2) was nearly 33

Metric	MOLDYN	RAYTRACER
#Lines of code	965	1540
#Classes	9	17
#Methods	28	71
#Statements	433	421
#Source code size in Kb.	26	49

Table 2. Metrics of the MOLDYN and RAYTRACER benchmarks.

million states. Additionally, to get an idea of the models' size and complexity, its source code metrics are shown in table 4.2.

As the benchmarks are written in Java, we had to convert them to C#. This was done automatically using Microsoft's Java Language Conversion Assistant 3.0, which is included with Microsoft Visual Studio 2005. The conversion was nearly complete and self-contained. The only two things that were not automatically converted were `assert` statements and final field attributes. The first was fixed by manually converting the assert statement to a `System.Diagnostics.Debug.Assert` statement in the resulting C# code. The second was fixed by adding the `readonly` attribute to fields which are marked final in the Java code.

While running initial runs, MOONWALKER found an assertion violation in both models due to a datarace. The datarace is on the correctness property, so the race does not affect behaviour of the model. The data races in the Java Grande Forum Benchmarks have also been detected by [16]. While the datarace can be fixed by proper synchronisation on the variables read by the correctness property, we purposely did not do that. We wanted to keep the benchmarks as pure as possible, and secondly, the datarace only increases the state space, so the only thing that happens is that the model checker has to do more work.

4.3 Setup

All benchmark runs were performed on identical 2.4 GHz computers with 2 GB memory installed. Each computer ran an identical Windows XP installation with an installation of Sun's Java 1.6 and Microsoft's .NET 3.0. Verification runs that exceeded 10 hours were automatically terminated. Also, verification runs that exceeded 1.5 Gb use of memory were also automatically terminated. The models checkers were configured for detection of deadlocks and assertion violations. Also, they were configured not to stop at the detection of an error, but to continue to explore the state space.

The .NET port of the Java Grande Benchmarks were compiled with MONO 1.1.8. The `DEBUG` constant was enabled such that the assertion check would be compiled into the CIL assembly. All MONO compiler optimisations were also enabled. The Java version of the Java Grande Benchmarks were compiled with Java SDK 1.6 with the compiler options `-target 1.4` and `-source 1.4`. This ensures that Java 1.4 compatible JVM bytecodes are generated, which is the highest JVM bytecodes version Bandera could process.

Our initial benchmark runs showed that JPF explored a much smaller state space than MOONWALKER. We found that this was caused by a POR bug in JPF. It assumed that accesses to arrays were always independent, which is not true. Accesses to arrays are only independent if (and only if) the accessed array is thread-unshared. We fixed this bug in our own version of JPF and reran all experiments again. The bug is fixed in the current development branch of JPF.

4.4 Usage

The aforementioned benchmark models and setup are used to evaluate the effectiveness of all the techniques described in this paper. The same setup was also used for the comparative analysis between MOONWALKER and JPF. To avoid recapitulation, the benchmark data presented in the subsequent sections are obtained from this experimental setup.

5 Structured Exception Handling

Exception handling is an important programming language construct to handle the occurrence of exceptions, i.e., special conditions that change the normal flow of program execution. Examples are a division by zero or a sudden network disconnect. They are raised by the program itself using a special statement or they are raised by the runtime environment, like the virtual machine. It is up to the programmer to write exception handlers to deal with these special conditions. As exceptions intrinsically break the normal flow of program execution, writing clean exception handling code is difficult. For this reason, modern runtime environments and their accompanied programming languages like JVM/Java and CLI/C# provide a broad scale of exception handling constructs.

Exception handling schemes have various degrees of simplicity and expresiveness. The most simple ones stop processing and directly run a designated exception handler. When this is put in a multi-threaded context, the invokation of the exception handler is nothing more than just a function call. The most advanced schemes, like the *structured exception handling* (SEH) scheme [31] used for .NET and the Windows architecture in general, allow for hierarchies of self-defined exceptions. On top of that, it furthermore allows for self-defined exception matchers that are responsible for looking up the correct exception handler. When this is put in a multi-threaded context, other threads intertwine during this form of exception handling and can potentially affect the choice of exception handler.

For MOONWALKER, we accounted for the full semantics of SEH, including the possible effects of intertwining threads on exception handling. This required an utmost cautious design and implementation, and hence, ours is the most advanced one to date in a model checker. In this section, we show how we implemented this involved scheme in MOONWALKER.

5.1 Java vs. CLI's Exception Handling

We assume familiarity of Java's exception handling [3] here and use that familiarity to introduce structured exception hand-

```
Public Sub ExceptionTestWithUserFilter()
  Try
    ...
  Catch ex As MyException
    When (ex.Code = 1 && CheckBounds(ex))
    ...
  Catch ex as OverflowException
    ...
  End Try
End Sub
```

Figure 4. Expressing filter-handlers in VisualBasic.NET.

```
public void exceptionTestWithUserFilter() {
  try {
    ...
  } catch (Exception ex) {
    if (ex instanceof MyException &&
        (MyException) ex).Code == 1 &&
        checkBounds((MyException) ex)) {
      ...
    } else if (ex instanceof OverflowException) {
      ...
    } else {
      throw;
    }
  }
}
```

Figure 5. Expressing filter-handlers in Java.

ling. In Java, exceptions are initiated using the `throw` statement. It takes an exception object as an argument, indicating the nature of the exception and may contain additional information related to it. This object is constructed as any ordinary object. Parts of the Java code can then be guarded with `try` clauses. Herein exceptions are thrown. If this occurs, an appropriate exception handler is looked up by matching the type of the exception object against the type expressed in the handler. The program flow then jumps to the body of that handler. When the `try`-block is left without an exception or when the end of the exception handler is reached, the program flow jumps to the code guarded by a `finally` clause. The programmer perform finalising operations in part in order to clean up the program state, like closing network sockets or releasing the locks.

The exception handling mechanism defined for the CLI has its roots from the Windows architecture. It was designed to deal with both hardware and software exceptions. It differs from Java's on three points.

(i) In the CLI, all exceptions are unchecked. This in contrast to Java, where nearly all exceptions are checked. This means that at the source level, the methods do not catch all exceptions possibly raised by their bodies. Instead, upon a thrown exception, the CLI requires the runtime environment to traverse the thread's callstack in search for a method that has a matching handler. If none are found, the thread in which the exception occurs stops.

(ii) In the CLI, the programmer can specify its own exception handler matching. Unlike Java, where an appropriate exception handler is found by matching the type of the exception object using the `catch` clauses, the CLI allows in addition filter-handlers. A filter-handler consists of two parts, the filter and the handler. The filter is a block of statements defined by the programmer that returns true or false. Upon true, the accompanied handler is invoked for handling the exception. Upon false, the CLI further traverses the callstack for possible filter- and catch-handlers. Filter-based and catch-handlers can be mixed. An example is shown in Figure 4, which is a snippet of VisualBasic.NET code. It is possible to express filter-handlers in Java using a catch-all, and have that handler to do the actual exception handling. This approach is less elegant with the equivalent in CLI, as can be seen in Figure 5.

(iii) The CLI supports, in addition to `finally`-handlers, also `fault`-handlers. They are similar, but differ in that fault-handlers are only invoked when an exception was or will be handled.

5.2 Design

The major challenge of implementing structured exception handling in a software model checking lies in the possibility that threads can interleave while fault-, finally, catch and filter-handlers are invoked in other threads. The CIL instructions that drive the disruptions of control flow due to these handlers are `throw`, `rethrow`, `endfilter`, `leave`, and `endfinally`. We shall show what operations are needed upon interpretation of these instructions in order to deal with SEH.

The global overview of SEH is shown in Figure 6. Three global phases are distinguished, namely the entrance to the try block, the exception handling mode and the finalising mode. They are discussed in the next subsections.

5.2.1 Throwing Exceptions

In CIL bytecode, a try block covers CIL instructions by the `try { ... }` construct. The block is entered via the normal program flow. Upon a special condition for which an exception needs to thrown, the exception object has to be constructed first. This is done as any ordinary object, thus it may lead to several method calls and pushing method states upon the callstack. When the exception object has been constructed, it can be thrown using the `throw` instruction. Upon interpretation of that instruction, MoonWalker will enter exception handling mode (process 2 in Figure 6). The normal program flow will be disrupted.

In case no exception is thrown, the end of the try block will be reached. This is indicated by the `leave` instruction. MoonWalker sets a `left`-flag when finalising mode (process 3 in Figure 6) is entered. This flag is used to distinguish the case that finalising mode is entered after exception handling and the case of successfully leaving a try-block. In finalising mode, the `left`-flag is used to determine whether only finally-handlers should be invoked or both finally- and fault-handlers.

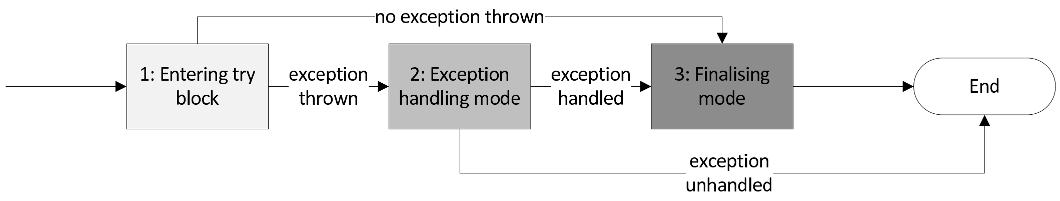


Figure 6. Global overview of structured exception handling.

5.2.2 Exception Handling Mode

In exception handling mode, *all* appropriate handlers must be invoked. The main difficulty to do this right is the possible interweaving of filter-handlers with catch-handlers and that in addition many fault- and finally-handlers might be invoked due to the unchecked nature of exceptions in structured exception handling. These invocations occur in a structured manner, as shown in Figure 7.

First the appropriate filter- or catch handler needs to be found (process 2.1 in Figure 7). For that, the callstack is traversed from top to bottom to search for the program counter of the first catch- or filter-handler within the scopes of the current program counters (line 3 in Algorithm 2). While traversing, the evaluation stacks of the traversed method states are purged (line 2 in Algorithm 2). This indicates that these method states could not handle the exception. When the end of the callstack is reached and no suitable exception handler was found, the callstack is cleared. The exception will be left unhandled and the thread will stop (line 17 in Algorithm 2).

Algorithm 2: HandlerLookup()

```
   Data: Callstack of current thread C
   Data: Reference to exception object e
1  foreach method state m ∈ C do
2      clear m's evaluation stack
3      eh ← next filter- or catch handler by type of e
4      if eh is a catch handler then
5          m's program counter ← begin of handler eh
6          push e on m's evaluation stack
7          pc ← FinallyOrFaultLookup()
8          return pc
9      else if eh is a filter handler then
10         push e on m's evaluation stack
11         clone m to $m_{clone}$
12         m's program counter ← begin of handler eh
13         pc ← begin of the filter of eh
14         push $m_{clone}$ on C
15         return pc
16 // This is only reached when no exception handler was found
17 clear C
```

However, if a matching catch-handler is found in a method, then the program counter of that method state is set to the beginning of the handler (see line 5 in Algorithm 2). Additionally, the reference to the exception object is pushed upon the cleared evaluation stack. This denotes that that method state is ready to handle the exception (see line 6 in Algorithm 2). Before handling the exception, all finally- and fault-handlers occurring in the method states above the method state that handles the exception have to be invoked first (see process 2.2 in Figure 7 and line 7 in Algorithm 2). This is done by traversing the callstack, do the necessary finally- and fault-handler invocations, until the method state is reached with a non-empty evaluation stack (i.e., containing the reference to the exception object because that method state contains the appropriate catch-handler). See also Algorithm 3.

Algorithm 3: FinallyOrFaultLookup()

```
   Data: Callstack of current thread C
1  m ← peek of C
2  pc ← m's program counter
3  while C is not empty $\wedge$ m's evaluation stack is empty do
4      m ← popped method from C
5      eh ← next filter- or catch handler by type of e
6      if eh $\neq$ null then
7          pc ← begin of handler eh
8          break
9  return pc
```

If however a filter is found first, the situation becomes more involved. The filter block needs to be invoked first to determine whether it should handle the exception. Moreover, it is possible that the filter occurs in a method state below the one from which the exception was thrown. It is not possible to pop off all method states above the one containing the filter, as these methods in between might contain finally- and fault-handlers that need to be invoked if (and only if) the filter returns true. Our solution is to clone the method state containing the filter and push the clone on the call stack (see line 11 in Algorithm 2). Its program counter is set to the beginning of the filter (see line 13 in Algorithm 2). The program counter of the original method state is set to the beginning of the filter-handler (see line 12 in Algorithm 2). The explorer then processes the filter as if it were any block of instructions.

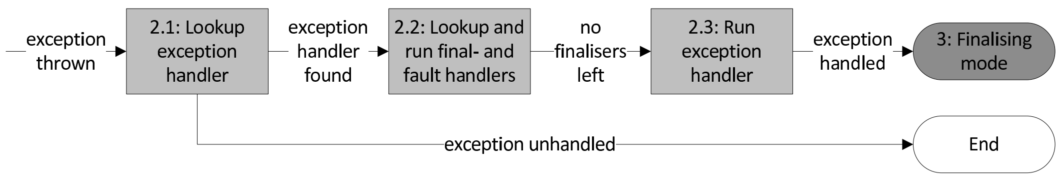


Figure 7. Flow diagram of exception handling mode (see process 2 in Figure 6).

The last instruction in a filter is always `endfilter`. Upon this instruction, a check is made whether the filter returned true. If so, process 2.2 kicks in, thus the finally- and fault-handlers between the top method state and the one that contains the filter are invoked. If the filter returns false, the callstack is further traversed for lookup of other possible matching catch- or filter-handlers.

5.2.3 Finalising Mode

The inner details of finalising mode (process 3 in Figure 6) is depicted in Figure 8. It handles the last invocation to the finally- or fault-handler in the same scope as the try-, filter- or catch-block. There are two different flows entering the finalising flow, depending on whether an exception was thrown and handled or not. The `left`-flag (set in process 1) is used for this.

In case the `left`-flag is set to true, indicating the try-block was left without any exceptions, only the next finally handler is looked up and invoked. If the `left`-flag is still false, then the next finally- or fault handler is looked up and invoked.

6 Summarising Interleaving Information

Two research groups independently investigated suitable statefulness extensions to dynamic partial order reduction [38,45]. Initially, a naive, but flawed approach shall be discussed in the following subsection, followed by a correct, but expensive, approach using interleaving information (II). It is an follow-up to the brief introduction to dynamic partial order reduction in Section 3. An optimisation upon this by [45], using summaries of interleaved information, is discussed thereafter. Despite the summarisation, the approach by [45] still incurs a high memory overhead. We propose a solution to reduce this by applying concepts borrowed from structured state collapsing and call it *Collapsed Summarised Interleaving Information*. Its effectiveness for stateful dynamic partial order reduction (SDPOR) is evaluated by the benchmark setup from section 4.

6.1 Statefulness to Dynamic Partial Order Reduction

A naive stateful adaptation of dynamic partial order reduction would be backtracking upon exploration of revisited state. This is incorrect, and is shown by the following counter-example: Consider an exploration to state s_n by the path $\pi_0 = s_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{n-1}} s_n$, and later on, the model checker reaches s_n again by a different path $\pi_1 = s_0 \xrightarrow{t'_0} \ldots \xrightarrow{t'_{n-2}} s'_{n-1} \xrightarrow{t'_{n-1}} s_n$. If, during exploration of path π_1, the model checker would simply backtrack upon the revisit of s_n , the set $Working(s'_{n-1})$ could be an incomplete persistent set, since the dependencies between transitions on the backtrack stack, i.e. $t'_0, \ldots, t'_{n-1}$, and any transition in the subspace below s_n would be unaccounted for. As dependent transitions would then not be injected into $Working(s'_{n-1})$, the result would be an over-aggresive reduction of the state space. Both [38] and [45] independently observed this. Their solutions are discussed hereafter.

6.1.1 Interleaving Information

Both [38] and [45] propose similar solutions to overcome the over-aggressive reduction of naive backtracking. Their idea is to mimic a stateful search upon a revisit by recalling all necessary information about the state space below the revisited state. In [38], each stored state is therefore associated with a set that contains all transitions that occur in the state space after the stored state. This set is called the *interleaving information.* Upon a revisit of a state, its associating interleaving information is requested, after which mutual dependencies between the transitions on the DFS stack and stored interleaving information is calculated, and appropriate transitions are injected in the working sets. This approach is however very memory-intensive. The interleaving information of states grows when one comes closer to the initial state. At the end of exploration, the initial state will hold all transitions in the state space. Nevertheless this drawback, [38] provide empiric evidence that their stateful extension to dynamic partial order reduction reduces the state space even more.

6.1.2 Summarised Interleaving Information

The flaw behind interleaving information is that it captures too much data, most of which is unnecessary to perform the correct dynamic partial order reduction. Hence, a summarised variant on the interleaving information was proposed by [45]. The idea is to omit transitions in the interleaving information set that have a predecessor transition (in the same set) accessing the same field entity.

The correctness proof behind this intuition refers to the notion of *path-independency*. Consider path $\pi = s_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{i-1}} s_i \xrightarrow{t_i} \ldots \xrightarrow{t_{n-1}} s_n$. The interleaving information of s_i would at least contain t_i and t_{n-1}. Assume that t_{i-1}, t_i and t_{n-1} access the same memory location m, then one should find that

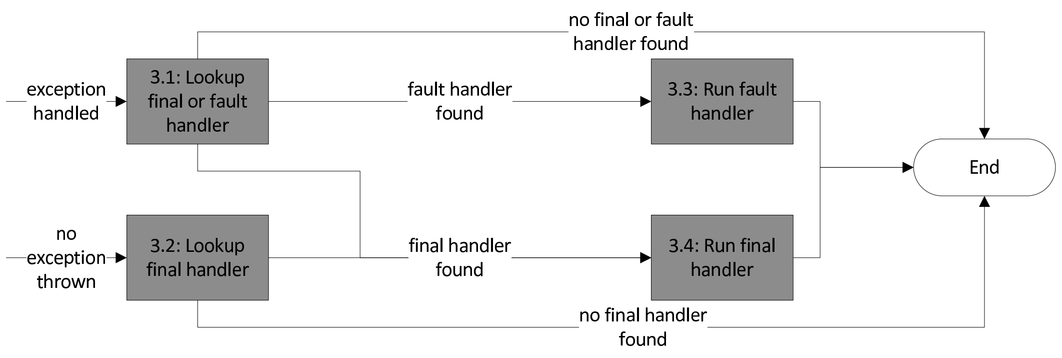


Figure 8. Flow diagram of finalising handling mode (see process 3 in Figure 6).

t_i is dependent on t_{i-1} and that t_{i-1} is independent with t_{n-1}, even though t_{n-1} accesses the same field. That is because t_{i-1} and t_{n-1} are independent in state s_i given path π, also called path-independent. Path-independent transitions in a state s can be determined beforehand, as path-independent transitions always have an intermediate transition that accesses the same object field (as defined in section 3.4). By leaving path-independent transitions out of the interleaving information, we get a summarised version of it. The transitions in the summarised interleaving information (SII) are called *minimum-indexed transitions*.

Path-independency is different than independency, as the latter is a global independency relation on the whole state space. It is possible that two transitions are path-independent on one path from state s, and on another path from s, they are path-dependent. Both transitions should then be included in the summarised interleaving information. This reveals another issue: the summarised interleaving information of a state s is not only constructed from one path from s, but for all paths from s. This is however problematic, as a typical state space contains exponentially many paths from a state. Instead of maintaining all paths, we devised a different approach by propagating the minimum-index transitions upwards in the state space. This exploits the observation that most of the minimum-indexed transitions of a state are the same as those of its successor states. We can simply merge the SII's of the successor states to get the SII for the parent state. The only pairs not valid during merging are those transitions that directly follow from s. An exception for them have to be made during merging. A second exception is also made for transitions that access fields of objects instantiated after state s. Such transitions are never dependent with transitions on any path to s. They too can be left out for consideration of merging. This consideration involved a check whether a field entity exists in state s.

A SII for a state s is incomplete until all the successor SII's are merged. We know this when the working set for s is empty, as then all transitions from s are explored and backtracked from.

6.2 *Structured State Collapsing*

State collapsing has been extended to software model checking in JPF by [30]. In software model checking, states are dynamic in size and are built up structurally. The latter eases state collapsing. States of software systems tend to be composed of objects, which are further composed of fields, a type definition and a locking wait queue. Hence its structure is clearly defined and the identifiable components in a state can represent parts to be collapsed. Upon backtracking, a state is easily restored by following the references to the collapsed part.

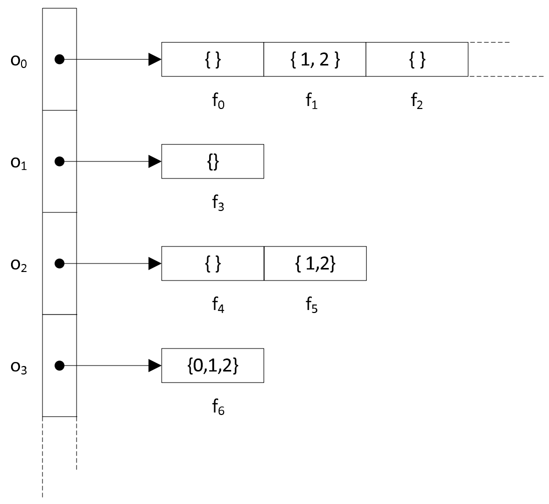
Structure state collapsing has proved to be very effective in reducing memory usage. The level of reduction is dependent on the collapsing scheme used. For JPF, a ten to twenty-fold reduction in memory use has been measured. Furthermore, collapsed states are faster matched because they only need to match the references to the collapsed parts, thereby prevent matching the substructures. collapsing however incurs an overhead, but this overhead is overly compensated by the performance increase incurred by only matching references [30].

6.3 *Collapsing Summarised Interleaving Information*

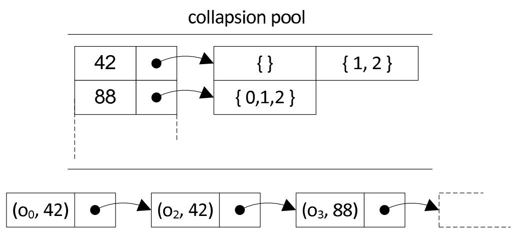
A SII of a state does not need to hold all details of the minimum-indexed transitions. It suffices to store only pairs of the field identifier accessed by the transition and the thread identifier involved. We considered a set-based approach of storing these tuples and a hierarchical indexed-based approach. The latter was devised because of its higher performance. The hierarchical indexed-based approach might incur a higher space-complexity, but we will show that a careful design, via a notion of canonicalisation and compression through collapsing, has better running time and memory characteristics.

6.3.1 Datastructures Design Considerations

The set-based approach is the most straightforward approach: the SII is a set of field-thread pairs. A fast implementation of a set is a hash set, which has $O(1)$ armortised operations. Our initial implementation of stateful DPOR used this approach. Initial runs revealed that the performance decreased so much,



(a) Organisation of the dynamic area in the SII.



(b) Collapsed representation of the above.

Figure 9. SII organisation and collapsing.

that it was even slower than POR disabled. Initially, we found this strange, because of the favourable, but amortised, time-complexity of a hash set. Yet, upon a closer look using a profiler, we observed that the merging of SII's upon backtracking took lots of time. The problem lies in the nature of an implementation of a set: before an element can be added, the set has to verify whether that element has not been added yet. This involves a call to the hash function and a lookup in the array. Although these operations are constant in time, the constant costs are quite high, especially when lots of field identifier-thread pairs were propagated between states.

After these observations we devised the indexed-based approach that fits closely to the heap-hierarchy. It mimics the organisation of a state and hence its field entities. In MOONWALKER, a state consists of three parts: the dynamic area (i.e., the heap), the static area (for static classes) and the thread pool. The first two contain fields. The dynamic area is composed as an array of objects. Each object is composed as an array of fields. The static area is composed as an array of classes. Each class is composed as an array of static fields. A SII differs slightly from this representation in that fields do not hold a value, but a set of threads, namely those in the minimum-indexed transition pairs. See figure 9(a).

Operations in the hierarchical indexed-based approach are also in $O(1)$. The advantage is that hashing is not used for looking up a field entity-thread pair, but that indexing is used instead. The latter has lower constant costs. The drawback of the hierarchical SII approach is that all fields in a state are modelled and thus they take space. Collapsing, as explained in the next subsection, is applied to alleviate that.

6.3.2 Collapsing Scheme

Collapsing compresses the SII by exploiting the well-known observation that SII's of states do not change much between successive states. This allows us to employ the same collapsing scheme as we use for collapsing states. In MOONWALKER this means that objects and classes are collapsed to a reference. See Figure 9(b). For further reducing the memory and performance overhead, we use canonicalisation to further optimise the representation of the collapsed SII.

Canonicalisation exploits the notion that interleaving information of two objects can be similar, even though their sizes differ. For example, in figure 9(b), assume object o_0 has a size of n field entities, where $n > 2$. Furthermore, assume that field entities after the f_1 were unaccessed, and thus no threads are stored in them. In terms of interleaving information, o_0 is similar to o_2, but o_2 uses less memory. For this reason, o_2 is stored instead. Whenever o_0 is collapsed, it is collapsed to the same reference used to for collapsing o_2, thereby saving memory.

Exploiting the sparsity of the collapsed SII is the second idea. A typical SII includes objects that are never accessed after a revisited state. These can be left out in the collapsed SII. We did this by modelling the collapsed SII as a linked list of pairs, where a pair consists of an object location and the collapsed reference. This is shown in figure 9(b), where the unaccessed object o_1 is not included in the collapsed SII.

Given these optimisations, the last remaining issue is when to collapse and to decollapse. Collapsing should occur when all paths from s have been explored. This is certain when the explorer backtracks the first time from state s. Decollapsing is necessary when the explorer stumbles upon an already seen state s' in the hashtable.

6.4 Evaluation

We used the JGF benchmarks (see §4) for evaluating the effectiveness of this collapsing scheme. All configurations were ran with collapsing enabled and after that, a second run with collapsing disabled. Disabling the collapser means that the SII is still stored in the hashtable, but in a uncollapsed form (like in Figure 9(a)).

Table 3 shows that for MOLDYN configurations 2-1, 2-2 and 3-1, the SII collapser enables over two times more states stored in the same amount of memory. This is also visible in the RAYTRACER results (see table 4), where the same amount of memory reduction is visible in configurations 2-2, 2-3, 3-1 and 3-2. In all other configurations, the reduction is not visible for either two reasons. The foremost reason is seen by the amount of backtracks. In case this in zero, the SII's were still on the DFS stack before the exploration ran out

config.	collapser	time (sec)	memory (Mb.)	states ($\cdot 10^3$)	revisits ($\cdot 10^3$)	stored states ($\cdot 10^3$)	backtracks ($\cdot 10^3$)	states/sec	stored states/Mb.
2-1	C	458	1470	1482	1063	1482	2544	5560	1008
	¬C	482	o.m.	1014	722	731	1727	3602	488
2-2	C	1553	o.m.	1926	788	977	2524	1748	651
	¬C	390	o.m.	552	203	393	563	1936	262
2-3	C	72	o.m.	249	0	249	0	3475	166
	¬C	79	o.m.	247	0	247	0	3112	165
3-1	C	1038	o.m.	2724	3018	1662	5677	5531	1108
	¬C	440	o.m.	1401	1294	727	2630	6127	485
3-2	C	98	o.m.	327	0	327	0	3324	218
	¬C	99	o.m.	326	0	326	0	3296	217
3-3	C	68	o.m.	151	0	151	0	2238	101
	¬C	70	o.m.	152	0	152	0	2174	101

Table 3. MOLDYN results with collapsing of SII's enabled (C) and disabled (¬C).

config.	collapser	time (sec)	memory (Mb.)	states	revisits	stored states	backtracks	states/sec	stored states/Mb.
2-1	C	1	36	844	579	844	1422	1198	23
	¬C	1	36	844	579	844	1422	1124	23
2-2	C	113	655	65923	53264	65923	119186	1055	101
	¬C	111	1256	65923	53264	65923	119186	1070	52
2-3	C	o.t.	1373	97233	24289	97233	97163	3	71
	¬C	o.t.	1501	97154	24269	58941	97084	3	39
3-1	C	68	475	53631	71076	53631	124706	1842	113
	¬C	74	1038	53631	71076	53631	124706	1684	52
3-2	C	o.t.	1572	30093872	246229	185707	30339192	843	118
	¬C	o.t.	1600	25803859	128121	83086	25929137	720	52
3-3	C	32	o.m.	43323	0	43323	0	1347	29
	¬C	38	o.m.	42713	0	42713	0	1136	28

Table 4. RAYTRACER results with collapsing of SII's enabled (C) and disabled (¬C).

of memory, and thus never were backtracked to, and as such, never collapsed to the hashtable. The second reason is reflected in RAYTRACER configuration 2-1. Here the amount of states is so small that the SII collapser cannot make difference.

The performance overhead of collapsing SII's is visible for MOLDYN configurations 2-2 and 3-1 and RAYTRACER configuration 2-2. Here, the slowdown of the SII collapser (in terms of states per second) is up to 10%. This is caused by the ex post facto statement merger (see Section 3), as reflected in the amount of states stored in those configurations. This statement merging technique causes purged revisited states to be reexplored. Its reexploration incurs an overhead that is visible in the amount of states per second. For most configurations however, the collapsing SII's improves performance. This is because the merging of collapsed SII's is faster than uncollapsed SII's as its collapsed counterpart is modelled as a sparse array and thus is faster to traverse. Furthermore, collapsing SII's frees up memory which can be used to store more states in memory, and hence, allows the explorer to detect more revisited states.

7 Reducing Time on Garbage Collection

As garbage collection is performed as part of heap symmetry reduction, it is an important means to deal with the state space explosion during exploration. It is however also an expensive algorithm. Using a profiler, we observed that the software model checkers typically spend around half of the time on garbage collection (see Figure 10 for an illustrating example). This does not come as a surprise as after most transitions the heap has to be cleaned from garbage. To lower the stake of garbage collection and therefore reduce the time needed for verification, we developed a new algorithm called the Memoised Garbage Collector (MGC). This algorithm is inspired by incremental graph updates from graph grammars and routing [37], and has a more favourable time-complexity compared to the often used Mark&Sweep (M&S) algorithm [32]. The key idea here is that instead of calculating reachability of a vertex (like M&S does), we

track the depths of the objects efficiently and purge objects whose depth becomes infinite, i.e., became unreachable.

We shall show how to use this idea to exploit the well-known observation in model checking that changes between successive states are usually small. This means that the changes to the heap graph are also small. By tracking these small changes to the heap and have them drive the garbage collection algorithm, we can save time for especially large heaps.

7.1 Notation

In this paper, a directed graph G with a source vertex is defined as $G = (V, v_0, E)$, where V is the set of vertices, E the set of edges, and $v_0 \in V$ is the initial, root vertex. The direct predecessors of a vertex u in a graph G is defined as the set $Pred(G,u)$. The set of direct successors of a vertex u are defined as $Succ(G,u)$.

7.2 Incremental Depth-Labelling Algorithms

Traditionally, depths of vertices are computed all at once using Dijkstra's algorithm, stored and used when necessary. We use $depth(u)$ to indicate the stored depth of vertex u. When the graph changes from G to G', the real depths of the vertices may change. This is however not reflected in the stored depths. Thus usually, upon a change to the graph, Dijkstra's algorithm is called to globally recompute the stored depths. For large graphs, it is more efficient to recompute only the stored depths of vertices whose real depths have changed. To date however, there is no method to determine efficiently and precisely this set of vertices.

Ramalingam and Reps devised a method to traverse efficiently an over-approximation of that set (see Algorithm 4). Their method can be viewed as an incremental version of Dijkstra's shortest path algorithm [13]. The over-approximation can be traversed using the notion of inconsistency and a top-down traversal order. The former is defined as follows:

> Given the stored depth mapping *depth*, a graph $G' = (V', v'_0, E')$ and the right-handside function $rhs(G',u) = \min_{v \in Pred(G',u)} depth(v) + 1$. A vertex $u \in V'$ is *inconsistent* if and only if $rhs(G',u) \neq depth(u)$.

Inconsistent vertices are spotted cheaply by monitoring the changes to the predecessor transitions upon graph changes, as shown later in Section 7.3. Then, the inconsistent vertices are traversed top-down according to their key, which is defined as the minimum of the *rhs* and the stored *depth*:

$$key(G',u) = \min(rhs(G',u), depth(u))$$

Algorithm 4: RamalingamReps()

```
   Data: graph $G' = (V', v'_0, E')$
1  while $G'$ contains inconsistent vertices do
2      $u \leftarrow$ the vertex with the lowest key
3      if $rhs(G',u) < depth(u)$ then
4          $depth(u) \leftarrow rhs(G',u)$
5      else if $depth(u) < rhs(G',u)$ then
6          $depth(u) \leftarrow \infty$
```

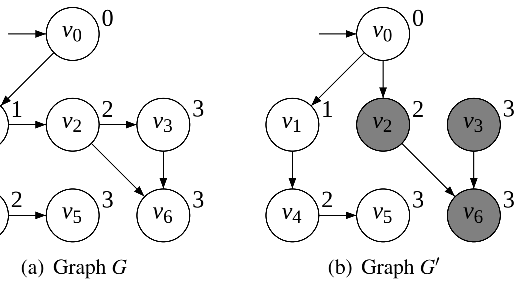


Figure 11. Graph G was changed to graph G', but the stored depths (the labels upper-right from the vertex) were not recomputed. Because of this, vertices v_2, v_3 and v_6 are inconsistent, as indicated by the gray fill in graph G'.

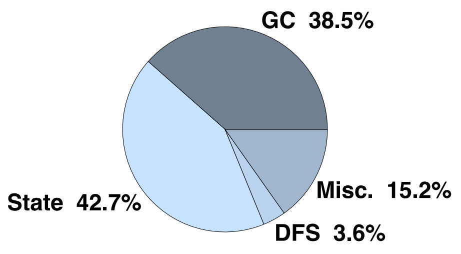
The vertex u with the lowest key is processed first, see line 2 of Algorithm 4. It is the inconsistent vertex closest to the root. If there are multiple vertices with the same lowest key, one is selected non-deterministically. In case its *rhs* is smaller than its stored depth, we know that the changes to the graph moved u closer to the root. We can assign its *rhs* value to *depth* to make it consistent (line 3-4). This could cause its successors, $Succ(G',u)$, to become inconsistent, and they will be processed when their key is the lowest. On line 5-6, we deal with the case that $rhs(G',u)$ is greater than $depth(u)$, thus it moved farther from the root. We assign its stored depth with infinity (∞). This ensures vertex u's key is purely determined by the *rhs* and if it is the lowest, it will be processed again. The cause-and-effect behaviour of making vertices consistent and triggering its successors become inconsistent is guaranteed to reach a fixpoint because of the traversal order by the lowest key. A proof of correctness is provided in [37].

7.2.1 Intuition by Walkthrough of an Example

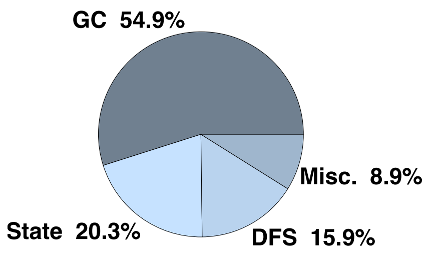
A walkthrough of this algorithm is shown in Figure 12. It outlines the steps of Ramalingam and Reps's algorithm on graph G' from Figure 11. It shows intuitively that this algorithm determines a subgraph of vertices for which the stored depths reflect the real depths. This is ensured for consistent vertices whose stored depth is smaller or equal to the vertex with the smallest key value. Based on this subgraph, the inconsistent vertex closest to this subgraph is made consistent. This enlarges the subgraph. This is recursively done until all inconsistent vertices are traversed and the subgraph is equal to the graph.

7.2.2 Applications of Incremental Shortest-Path

The foremost application of Ramalingam and Reps's algorithm is in routing. Routers need to recalculate shortest paths to neighbouring routers when the connections change. Whereas Dijkstra's algorithm recalculates all shortest paths, this algorithm only recalculates shortest paths that have actually changed. For large networks, this algorithm reduces time. In this paper, we show how the idea behind this algorithm improves garbage collection in software model checking.

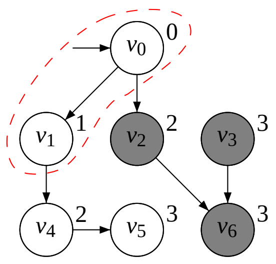


(a) Java PathFinder

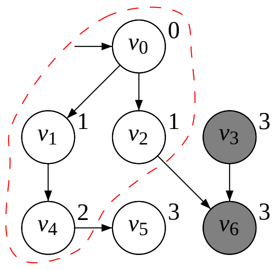


(b) MoonWalker

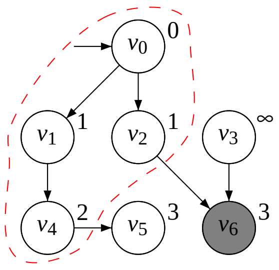
Figure 10. Profiler data from Raytracer 3-1 benchmark (see Section 4) describing the stakes of garbage collection (GC), state storage (State), exploration (DFS) and all remaining functionality (Misc.).



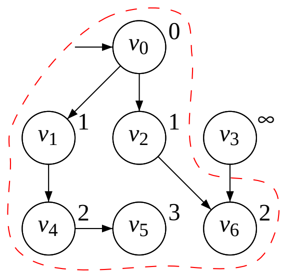
(a) Initial situation. Vertex v_2 is selected as the inconsistent vertex with the lowest key.



(b) Vertex v_2 was made consistent by assigning its *rhs* to its stored depth. Vertex v_3 is next.



(c) $depth(v_3)$ becomes infinity because it has no predecessors. Vertex v_6 is next.



(d) $rhs(G', v_6)$ is based on v_2 and therefore its stored depth becomes two. No inconsistent vertices left.

Figure 12. Walkthrough of Ramalingam and Reps's algorithm on graph G' of Figure 11. The vertices in the dashed subgraph are ensured consistent. In the last phase, all vertices with depth ∞ are unreachable and can be purged. Vertices v_3 and v_4 did not need to be traversed.

7.3 *Memoised Garbage Collection*

Ramalingam and Reps's algorithm works on graphs in general. To make it applicable for garbage collection in software model checking, we need to show how heaps can be represented as single-source directed graphs and how to implement graph (i.e., heap) operations efficiently.

7.3.1 Heaps as Single-Source Directed Graphs

A heap does not have a single root object, but multiple, namely the objects referenced from the call stacks of the program threads (see Figure 13). To make a heap graph a single-root graph, we introduce a fictive root v_0 whose successors are the objects referenced from the call stacks. Each reference counts as a distance of one. Given these semantic additions, the resulting graph can be processed by Ramalingam and Reps's algorithm. When the algorithm terminates, objects with an infinite depth are unreachable and can be garbage collected.

Due to the dynamic nature of object oriented software, the algorithm must also deal with newly instantiated objects, as they change the heap graph. To ensure that the new object will be seen as reachable from the fictive root, the stored depth of that object must be initialised with infinity. It will then be seen as inconsistent upon the first next run of the algorithm,

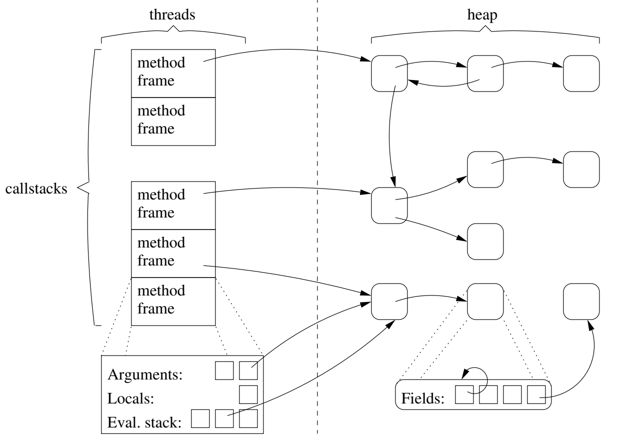


Figure 13. Organisation of a state in a software model checker. It consists of two threads, each having a callstack. The three objects on the left are root objects, as they are referenced directly from the call stacks.

and as such, it will be made consistent by assigning it with a consistent depth.

7.3.2 Implementing Graph Operations Efficiently

An implementation of the algorithm has three main issues to consider, namely (i) how inconsistent vertices are determined,

(ii) how an inconsistent vertex with the least key is found and (iii) how predecessors of an object are determined. Algorithm 5 is an implementation of Algorithm 4 for which these issues have been resolved.

Algorithm 5: MemoisedGC(s, s')

```
   Data: priority queue $Q$
1  $G = (V, E, v_0) \leftarrow$ the heap associated with state $s$
2  $G' = (V', E', v_0) \leftarrow$ the heap associated with state $s'$, the
   successor of $s$
3  foreach object $o \in V'$ do
4      if $Pred(G', o)$ is dirty $\vee$ $Pred(G', o) = \emptyset$ then
5          if $rhs(G', o) \neq depth(o)$ then
6              add $o$ to $Q$ with order $key(G', o)$
7  while $Q \neq \emptyset$ do
8      $u \leftarrow$ dequeue element from $Q$ with smallest order
9      if $rhs(G', u) < depth(u)$ then
10         $depth(u) \leftarrow rhs(G', u)$
11         $Affected \leftarrow Succ(G', u)$
12     else if $depth(u) < rhs(G', u)$ then
13         $depth(u) \leftarrow \infty$
14         $Affected \leftarrow Succ(G', u) \cup \{u\}$
15     foreach $o \in Affected$ do
16         if $rhs(G', o) \neq depth(o)$ then
17             if $o \in Q$ then
18                 adjust $o$ on $Q$ with order $key(G', o)$
19             else
20                 add $o$ to $Q$ with order $key(G', o)$
21         else if $o \in Q$ then
22             remove $o$ from $Q$
```

(i) The first issue is tackled by lines 3-6 and lines 15-21. Initially, a vertex could be detected on inconsistency by computing its *rhs* and compare it against its stored depth. Computing the *rhs* is expensive as it requires the traversal of all the predecessors. Therefore, we first use a safe indication, which we call "dirtyness". A predecessor set is dirty whenever it was modified, like additions and removal of predecessor objects, from the previous predecessor set, $Pred(G, o)$. Also, it is possible that during one transition, an object is created, used and discarded. Those objects have an empty predecessor set and also have to be considered for inconsistency. When an object passes these tests, then ultimately its *rhs* is calculated and compared against its depth. Between lines 15-21, successor objects are traversed that could have become inconsistent because their common parent has become consistent. The inconsistent childs are added to the priority queue Q so that they will be made consistent.

(ii) The second issue is determining the object with the least key. Inconsistent objects added to Q are sorted by their key. Due to this order, the object with the least key can be extracted in constant time. In case the key changes of an inconsistent object's that is already in Q (because its predecessor has a changed depth), then this change is reflected by an update to the queue, as done in line 17.

(iii) The third issue relates to function *rhs* and Algorithm 5. The heap only stores the successor relation explicitly. The predecessor relation can be derived implicitly from this. However, to speed up the algorithm, we maintain an explicit predecessor relation. In MOONWALKER, this relation has to be updated in the following situations:

- Upon interpretation of the `stfld` instruction, if an object reference is stored into an object's field.
- Upon interpretation of the `stelem` instruction, if an object reference is stored into an array element.
- Upon a `System.Array.ArrayCopy` internal call, when object references are copied to the destination array.
- When an object reference is pushed on the call stack; then the referenced object becomes a child of the fictive root.
- When an object reference is popped from the call stack; then the referenced object is removed as child of the fictive root.

When state collapsing [43] is applied, the predecessor relation also has to be updated similarly upon restoring an object, array and callstack. Furthermore, the predecessor relation has to be stored as a bag (i.e., a counting set). It is possible that an object references another object multiple times by holding the same object reference in multiple fields. If one of these references is removed, then the predecessor relation still holds. The predecessor relation between objects is discarded when all references to the successor object are removed.

7.4 Evaluation

In the following subsections, we explain the time-complexity of MGC and how this bound is reflected in the experiments with the JGF benchmark.

7.4.1 Time Complexity

Whereas the time complexity of M&S is linear to the size of the heap, the time-complexity of MGC is expressed in other terms. The main term is that of an affected object, which is an object whose stored depth has changed during one run of the algorithm. The extended size of an affected object o is $|Pred(o)|$. Given these terms, [37] showed that the worst-case time-complexity of algorithm 5 is $O(N \cdot (log(N) + M))$, where N is the sum of extended sizes of affected objects plus the amount of affected objects, and M the cost to calculate *rhs*.

7.4.2 Experiments

All benchmarks where run with the Memoised Garbage Collector enabled, and another series were run with the Mark & Sweep garbage collector enabled.

Table 5 show the results with MolDyn. MGC is faster (in terms of states/sec) for configurations 2-1, 2-2, 3-1 and 3-2, with respectively 5%, 7%, 14% and 8% performance increase. The average performance increase with MGC on these configuration is 9%. Table 6 shows that MGC is faster for all configurations except configuration 2-3. The increases

config.	gc.	heap size (#obj.)	time (sec)	memory (Mb.)	states ($\cdot 10^3$)	revisits ($\cdot 10^3$)	states stored ($\cdot 10^3$)	states stored/Mb	states/sec
2-1	MGC	45	434	1470	1482	1063	1482	1008	5863
	M&S		458	1470	1482	1063	1482	1008	5560
2-2	MGC	101	1447	o.m.	1928	790	978	652	1878
	M&S		1553	o.m.	1926	788	977	651	1748
2-3	MGC	253	78	o.m.	246	0	246	164	3163
	M&S		72	o.m.	249	0	249	166	3475
3-1	MGC	60	913	o.m.	2726	3022	1664	1109	6296
	M&S		1038	o.m.	2724	3018	1662	1108	5531
3-2	MGC	144	91	o.m.	328	0	328	218	3591
	M&S		98	o.m.	327	0	327	218	3324
3-3	MGC	372	153	o.m.	152	0	152	101	993
	M&S		68	o.m.	151	0	151	101	2238

Table 5. MolDyn results with the Memoised garbage collector (MGC) and the Mark & Sweep garbage collector (M&S).

config.	gc.	heap size (#obj.)	time (sec)	memory (Mb.)	states	revisits	states stored	states stored/Mb	states/sec
2-1	MGC	935	1	37	844	579	844	23	1231
	M&S		1	36	844	579	844	23	1198
2-2	MGC	940	109	664	65923	53264	65923	99	1091
	M&S		113	655	65923	53264	65923	101	1055
2-3	MGC	3254	o.t.	1151	79673	19899	79673	69	3
	M&S		o.t.	1373	97233	24289	97233	71	3
3-1	MGC	1368	38	483	53631	71076	53631	111	3278
	M&S		68	475	53631	71076	53631	113	1842
3-2	MGC	1368	o.t.	1571	32520383	248967	187623	119	910
	M&S		o.t.	1572	30093872	246229	185707	118	843
3-3	MGC	1384	23	o.m.	43330	0	43330	29	1890
	M&S		32	o.m.	43323	0	43323	29	1347

Table 6. Raytracer results with the Memoised garbage collector (MGC) and the Mark & Sweep garbage collector (M&S).

are respectively, 3% for both configurations 2-1 and 2-2, 78% for configuration 3-1, 8% for configuration 3-2 and 40% for configuration 3-3. The average increase of these configurations is 26%.

We hypothesised that the increase of performance correlates with the heap size. This is partially true. We saw that RayTracer configurations have bigger heaps, and as such the performance increase is generally higher than those of the MolDyn benchmarks. The latter configurations however revealed a surprising result, namely a huge decline in performance for configuration 3-3 and a moderate decline in performance for configuration 2-3. We investigated this using a profiler and observed that an assumption we hold true does not always hold. We assumed that the heap does not change much between successive states. This depends however on the heap property that is being measured. The heap shape does not change much, but we did observe that the depth labelling changes much for MolDyn configurations 2-3 and 3-3. As object references are popped and pushed upon the callstacks, the children of the fictive root change, and thus, also the heap. Also, these affected objects can cause a chain reaction of changed depth labelling of subsequent child objects. The MGC bases object reachability on this depth labelling. Furthermore, the profiler revealed an overhead in the maintenance of parent lists. These list are updated upon every change to the heap. The changes are especially heavy when a collapsed state is restored, where it is not uncommon that many are object change. Note that these observations also depend on the model that is being verified. The RayTracer model is less suspective to massive depth-labelling changes between successive states, thereby benefiting more from MGC.

When it comes to memory overhead (in terms of states stored per Mb.), we see that there is no significant difference between MGC and M&S. This means that the memory overhead for maintaining parentlists is neglectible.

8 MoonWalker vs. JPF

The comparison of tools, especially when done by empirical evidence, is a worthwhile effort. Each tool has a unique mix of techniques that were developed for them. It is interesting to learn the strengths and pitfalls of them. These results can be used to cross-fertilise techniques among tools.

For this paper, we compared MOONWALKER against JPF using empirical data from benchmarks. The benchmarks included BANDERA (using BOGOR as the backend) as well, but was finally left out of the comparison. Its meager performance of a few states per second was evidently outperformed by MOONWALKER and JPF, which easily explore thousands of states per second. This is however explainable. BANDERA's development paused since 2006 and is not updated with the latest developments. Hence, from here on, only MOONWALKER and JPF are compared. Readers interested in the benchmark data of BANDERA are referred to [34].

Feature-wise, MOONWALKER is equipped with several features, like Memoised Garbage Collection (see Section 7) and Stateful Dynamic Partial Order Reduction (see Section 6) which are not present in JPF. Despite these differences, a rough comparison is still possible because the tools are similar in many ways:

- The input is a program that is also executable.
- The program under verification is implemented in a programming language which is in wide-spread use (i.e. Java and C#)
- The program is verified by interpreting its representation in bytecode.
- Their input languages are based on the same object-oriented model, and thus have similar semantics and expressiveness.
- They use the same model checking approach and techniques: DFS search, statefulness, POR, deadlock detection and assertion violations.
- They are publicly available.

Other software model checkers were considered, but not taken into the comparative analysis because they did not fulfilled the above. For example, the SLAM model checker [5], is different in that it interprets C code (and thus not object-oriented) and applies heavy abstraction techniques, which might be refined during the same verification run. This model checking approach is called Counter-Example Guided Abstraction Refinement (CEGAR). The BLAST model checker [22,6] is similar, but uses a different abstraction approach. The software model checker Zing [2] uses a specialised object-oriented input language. This language is not, as far as we know, used for building software, as it lacks the expressiveness for that, like inheritance. XRT [20] is, like MOONWALKER, also a software model checker that interprets CIL bytecode. It converts CIL bytecode to an abstraction called XIL, which is then verified. Its featureset is similar to JPF's. XRT is however not publicly available, and therefore could not be taken with the benchmarks.

8.1 Results

The verification runs with MOONWALKER were run with both stateful dynamic POR and static POR enabled, collapsing of SII's was enabled and the use of the memoised garbage collector. The verifications with JPF were run with POR using object escape analysis. Heuristic search and symbolic extension were not enabled.

The results from these runs are shown in table 7 and table 8. We organise our analysis of MOONWALKER against JPF from three perspectives: speed, reduction of the state space and memory utilisation.

8.1.1 State Space Reduction

MOONWALKER and JPF are both able to fully verify MOLDYN configuration 2-1 and RAYTRACER configurations 2-1, 2-3 and 3-1. In addition to that, JPF is also able to fully verify RAYTRACER configuration 3-2. This observation was at first surprising, as we expected that MOONWALKER would be better at reducing the state space than JPF because it employs stateful dynamic POR. We noticed that this only holds for the fully explorable RAYTRACER configurations. Our investigation led two several factors that have an major influence on the resulting state space size.

(i) Stateful dynamic POR is not much effective on MOLDYN. We ran separate runs with MOONWALKER where we evaluated the effectiveness of stateful dynamic POR versus POR using object escape analysis in isolation. We observed that the state spaces of MOLDYN are very deep and that POR using object escape analysis already does a pretty good job. Therefore adding stateful dynamic POR upon POR using object escape analysis only leads to an overhead for this particular model.

(ii) MOONWALKER's object escape analysis is more rudimentary than JPF's. JPF also considers locking information to detect thread-unshared objects. This is especially beneficial to the RAYTRACER benchmark, where locks are used to rendez-vous barriers.

(iii) There is a small semantic difference between Java and .NET. For example, locking semantics of .NET cause an additional state explosion. In .NET, contrary to Java's, every lock operation (like wait or exit of a monitor) is preceded by a test that checks whether the thread actually holds the lock. This check is thread-unsafe, and adds another scheduling point that results in the creation of a state. Also, the semantics of CIL and JVM bytecodes differ slightly, and this might be reflected in the resulting compiled bytecode. The respective compilers might perform optimisations which the other does not. We did inspect numerous parts (the whole is too big to study within the available time) of the bytecode of the .NET and Java versions of the Java Grande Benchmarks, and while we did not detect any significant differences, it is not unimaginable that other parts of the bytecodes are different.

8.1.2 Speed

The foremost indicator of speed is the amount of states per second. The table of the MOLDYN results shows that MOONWALKER outperforms JPF in terms of states per second on configurations (config.) 2-3, 3-1, 3-2 and 3-3. JPF is faster on configurations 2-1 and 2-2. As states under (i) of §8.1.1, the latter is explained due to the overhead of stateful dynamic

config.	model checker	time (sec)	memory (Mb.)	states	revisits	max. depth	states/sec
2-1	MoonWalker	434	1470	1481603	1062794	27885	5863
	JPF	411	1488	1377258	1319071	26686	6560
2-2	MoonWalker	1447	o.m.	1928282	789872	195892	1878
	JPF	28618	o.m.	65681631	65270139	194710	4576
2-3	MoonWalker	78	o.m.	245878	0	245878	3163
	JPF	508	o.m.	567965	0	567964	1118
3-1	MoonWalker	913	o.m.	2725664	3021684	65706	6296
	JPF	31442	o.m.	65681631	128783286	63307	6185
3-2	MoonWalker	91	o.m.	327597	0	327597	3591
	JPF	19610	o.m.	7864076	11395825	565215	982
3-3	MoonWalker	153	o.m.	151782	0	151782	993
	JPF	529	o.m.	457719	0	457718	865

Table 7. Results of the MolDyn benchmark comparing MoonWalker and JPF.

config.	model checker	time (sec)	memory (Mb.)	states	revisits	max. depth	states/sec
2-1	MoonWalker	1	37	844	579	73	1231
	JPF	14	1488	2205	1978	214	299
2-2	MoonWalker	109	664	65923	53264	3173	1091
	JPF	112	1488	67511	64180	3318	1176
2-3	MoonWalker	o.t.	1151	79673	19899	19972	3
	JPF	o.t.	1488	66352	33063	33286	3
3-1	MoonWalker	38	483	53631	71076	145	3278
	JPF	172	1488	63207	117031	425	1048
3-2	MoonWalker	o.t.	1571	32520383	248967	3245	910
	JPF	9944	1488	3590219	6938176	3528	1059
3-3	MoonWalker	23	o.m.	43330	0	43330	1890
	JPF	379	o.m.	214594	0	214593	566

Table 8. Results of the RayTracer benchmark comparing MoonWalker and JPF.

POR on MolDyn. When MoonWalker was run with only POR using object escape analysis, it outperformed JPF by far (21246 states per second on configuration 2-1 and 6536 states per second on configuration 2-2). For the RayTracer benchmark, MoonWalker outperforms JPF on configurations 2-1, 3-1 and 3-3. For all remaining configurations, it is either on par or nearly on par with JPF.

8.1.3 Memory Consumption

JPF always reports that it used all the memory up to nearly 1.5 Gb, even for small state spaces. This probably unlikely. It is likelier that the measured memory usages are the allocated heap size and the not true peek usage of the heap. However, there is an indication that JPF utilised memory more efficiently than MoonWalker by looking at the pace at which MoonWalker runs out of memory. For all verifications that ran out of memory, MoonWalker explores less states in the same amount of memory than JPF. This is due to the use of stateful dynamic POR, which incurs a relative high memory overhead (see Section 6).

Disclaimer. In general, there are too many factors involved with conducting experiments on two different platforms (.Net and Java) with two different tools (JPF and MoonWalker) and two different representations of the model (the Java version and the .Net version) in order to draw hard conclusions. That is why we have not stated concrete numbers on the relative performance difference between JPF and MoonWalker.

9 Summary and Future Work

The dominant part of this paper was devoted to three novel techniques: (i) structured exception handling, (ii) dynamic partial order reduction using summarised interleaving information, and (iii) memoised garbage collection. We have shown that these techniques indeed improve the performance of MoonWalker. We believe that other virtual machine based model checkers could profit from these techniques as well. In this section, we summarise the results and pinpoint directions for future work.

SEH. SEH is one of the most sophisticated and fine-grained exception handling mechanisms for application platforms. We have implemented it fully in MoonWalker. Several difficulties had to be overcome to make this possible. To our knowledge, its implementation is the most sophisticated in a model checker to date. Its architecture can be used as a reference for future model checkers.

SII. The idea of collapsing interleaving information came from [45] and [38]. They observed that stateful dynamic POR uses lots of memory and suggested as future work to compress the interleaving information used for stateful dynamic POR. Our solution is to compress the interleaving information by canonicalisation followed by collapsing it. Experiments with it show that it reduces the memory use by a factor of two, allowing more states to be stored in the same amount of memory and thus, allowing larger state spaces to be explored.

In order to cope with bigger state spaces, we suggest to further investigate the use of POR. MOONWALKER currently employs POR using object escape analysis and stateful DPOR. The first can be improved by also considering locking information, as described in [14]. Stateful DPOR can be further improved by distinguishing more fine-grained independencies, like distinguishing read-write dependencies and read-read independencies. MOONWALKER can be further improved by using static POR pioneered by [38]. They use the Indus analyser to extract independencies. The same can also be applied for MOONWALKER if a similar analysis tool is developed for .NET. A fourth POR technique, sleep sets [19], can be used to reduce the number of transitions, and hence the amount of revisits. Besides POR, we believe that the addition of program slicing will reduce the state space significantly. This too requires an analysis tool for .NET, which development could be combined with the analyser for static POR.

There are still several ways to further improve the modelling, calculation and storage of SII's. First, two kinds of SII's need to be distinguished, namely those associated with states that are still on the stack, and those remaining in the hashtable. The SII's on the stack are unfinished, and successor SII's still have to be merged with it, and thus the datastructure is still in a volatile state. Also, in order to have the merging process to be fast, it needs to look up field entities fast. Hashtable SII's are used differently. The information they hold are only read, and also, when it is read, all its data is read. It is desired that hashtable SII's use memory more efficiently, because the number of SII's grows corresponding with the amount of stored states.

MGC. Software model checkers spend around half of their time on garbage collection using the Mark&Sweep algorithm. To optimise this, we described the Memoised Garbage Collecter, which has a better time-complexity than M&S. In particular, we show how depth-information can be used to determine reachability, how an incremental shortest-path algorithm can be applied to track the depths efficiently, how this algorithm drives the MGC, how this garbage collection algorithm can be implemented and finally an experimental evaluation of it on real-life benchmark models. The performance gain observed from our benchmarks is up to 78% percent, depending on the model and configuration.

Profiler measurements revealed that MGC decreases the stake of garbage collection from 55% to 24% on RAYTRACER 3-1. Yet, during development of MGC, we identified several opportunities for further optimising the garbage collection process.

Implementation. The implementation can be further improved by using a HOT queue [7] instead of currently used the Interval Heap for Q in Algorithm 5. The HOT queue has a better time-complexity for monotone increasing keys, which holds for this algorithm. Also the calculation of the *rhs* function can be done in constant-time using the improved algorithm by [37]. Both are more difficult to implement efficiently and for this reason, it is deferred as future work.

No garbage or lots of garbage. While studying the effect of MGC, we observed that lots of calls to the garbage collector do not result in the collection of garbage objects and that some calls result in the collection of lots of objects. The first especially happens when the callstack of a thread grows by successive method calls. If one would develop a method to detect this beforehand and disable the garbage collector, time is saved, especially when combined with M&S. The latter, collection of lots of garbage, occurs when exploration comes closer to the end state. By switching from MGC to M&S, thus a hybrid-approach, would benefit here.

Other incremental shortest path algorithms. The incremental shortest path algorithm by Ramalingam and Reps set off an active field of study on incremental shortest path calculation. Since its publication, hundreds of publications describing refinements and specialisations have emerged. It is well possible that improvements have been developed that are also applicable to MGC.

Incremental cycle detection with reference counting. The improvements mentioned above are merely to improve the MGC. Our study also gave us an idea for a more fundamental improvement. MGC uses the depth of a vertex as a property to determine reachability. In the end, it is all about the latter, not about the depth. Other properties of a graph might be used instead. For example, a fundamental different approach is to combine reference counting with a form of incremental cycle detection. The incremental cycle detector exploits the changes in a transition by incrementally maintaining the list of cycles in a heap. The reference garbage collector only has to check whether the change causes the cycle to become unreachable from the object graph, and if so, collect the cycle.

Applications of incremental computation. The incremental nature of the MGC is also applicable to other algorithms. For instance, [33] describes an incremental heap canonicalisation algorithm based on Iosif's canonicalisation algorithm [27]. They use the shortest path to achieve this, and, as they suggest themselves, can be calculated incrementally. This can be further extended to gain an incremental k-BOTS algorithm, such that thread symmetries [28] can be detected incrementally.

MOONWALKER. In the beginning of this paper we thoroughly discussed MOONWALKER 1.0, a model-checker for CIL bytecode programs. Due to several refactorings, the design and implementation of MOONWALKER is clear, readable and extensible.

We feel that MOONWALKER is already an useful platform in an academic environment where ease of experimentation with different implementations is an important virtue. To extend the usability of MOONWALKER even further, several improvements are planned:

– Further improvements to partial order reduction and state compression;
– Optimisations to the memoised garbage collector;
– Research on alternative incremental garbage collection schemes;
– Mixing of symbolic and concrete data;
– Multi-threaded and distributed version of MOONWALKER;
– Case studies with other programming languages than C#;
– Support for C# 3.0 and .NET 3.5;
– Approximate model checking capabilities using bitstate hashing and hash compaction techniques. In combination with the successful SWARM approach [25] this could considerably increase the coverage of the model checking process.

References

1. N. H. M. Aan de Brugh, V. Y. Nguyen, and T. Ruys. MoonWalker: Verification of .NET Programs. In S. Kowalewski and A. Philippou, editors, *Proc. of the 15th Int. Conf. on TACAS 2009, York, UK, March, 2009*, LNCS 5505, pages 170–173. Springer, 2009.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A Model Checker for Concurrent Software. In R. Alur and D. Peled, editors, *Proc. of 16th Int. Conf. on Computer Aided Verification (CAV 2004), Boston, MA, USA, July 2004*, LNCS 3114, pages 484–487. Springer, 2004.
3. K. Arnold, J. Gosling, and D. Holmes. *Java Language Specification*. Prentice Hall, third edition edition, 2005.
4. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, New York, 2008.
5. T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software Via Static Analysis. In *Proc. of the 29th Symp. on Principles of Programming Languages (POPL 2002)*, pages 1–3, 2002.
6. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST: Applications to Software Engineering. *Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.
7. B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. In M. Saks, editor, *Proc. of the 8th ACM-SIAM Symposium On Discrete Algorithms (SODA'97), New Orleans, Louisiana, USA*, pages 83–92. Society for Industrial and Applied Mathematics, 1997.
8. E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Proc. of the 10th Int. Conf. on TACAS 2004, Barcelona, ES, 2004*, LNCS 2988, pages 168–176. Springer, 2004.
9. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry Reductions in Model Checking. In A. J. Hu and M. Y. Vardi, editors, *Proc. of the 10th Int. Conf. on Computer Aided Verification (CAV 1998)*, LNCS 1427, pages 147–158. Springer, 1998.
10. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
11. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State models from Java Source Code. In *Proc. of 22nd Int. Conf. on Software Engineering (ICSE 2000), Limerick, Ireland, June, 2000*, pages 439–448. ACM, 2000.
12. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Bandera: A Source-Level Interface for Model Checking Java Programs. In *Proc. of 22nd Int. Conf. on Software Engineering (ICSE 2000), Limerick, Ireland, June, 2000*, pages 762–765. ACM, 2000.
13. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
14. M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.
15. ECMA International. Standard ECMA-335: Common Language Infrastructure (CLI), June 2005. http://www.ecma-international.org/publications/standards/Ecma-335.htm.
16. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Proc. of First combined Int. Workshops on Formal Approaches to Software Testing (FATES 2006) and Runtime Verification (RV 2006), Seattle, WA, USA, August 2006*, LNCS 4262, pages 193–208. Springer, 2006.
17. C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In J. Palsberg and M. Abadi, editors, *Proc. of POPL 2005*, pages 110–121. ACM, 2005.
18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
19. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD in Computer Science, University of Liege, Nov. 1994. A revised version has been published as LNCS 1032, Springer-Verlag (1996).
20. W. Grieskamp, N. Tillmann, and W. Schulte. XRT: Exploring Runtime for .NET - Architecture and Applications. *Electr. Notes Theor. Comput. Sci. (ENTCS)*, 144(3):3–26, 2006.
21. K. Havelund and T. Pressburger. Model Checking JAVA Programs Using JAVA PathFinder. *Software Tools for Technology Transfer (STTT)*, 2(4):366–381, Apr. 2000.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In T. Ball and S. K. Rajamani, editors, *Proc. of the 10th Int. SPIN Workshop (SPIN 2003), Portland, OR, USA, May 9-10, 2003*, volume 2648 of *LNCS 2648*, pages 235–239. Springer, 2003.
23. G. J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proc. of the 3th International SPIN Workshop (SPIN 1997), University of Twente, Enschede, The Netherlands*, 1997.
24. G. J. Holzmann. *The* SPIN *Model Checker – Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2004.
25. G. J. Holzmann, R. Joshi, and A. Groce. Tackling Large Verification Problems with the Swarm Tool. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Proc. of the 15th Int. SPIN Workshop (SPIN 2008), Los Angeles, CA, USA, August, 2008*, LNCS 5126, pages 134–143. Springer, 2008.
26. G. J. Holzmann and M. H. Smith. Software Model Checking. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Proc. of FORTE/PSTV 1999*, volume 156 of *IFIP Conference Proceedings*, pages 481–497. Kluwer, 1999.

27. R. Iosif. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *Proc. of the 16th Int. Conf. on Automated Software Engineer (ASE 2001), San Diego, USA, November, 2001*, pages 254–261. IEEE Computer Society, 2001.
28. R. Iosif. Symmetry Reductions for Model Checking of Concurrent Dynamic Software. *Software Tools for Technology Transfer (STTT)*, 6(4):302–319, 2004.
29. R. Iosif and R. Sisto. Using Garbage Collection in Model Checking. In K. Havelund, J. Penix, and W. Visser, editors, *Proc. of the 7th. Int. SPIN Workshop (SPIN 2000), Stanford, CA, USA, August, 2000*, LNCS 1885, pages 20–33. Springer, 2000.
30. F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In M. B. Dwyer, editor, *Proc. of the 8th. Int. SPIN Workshop (SPIN 2001), Toronto, Canada, May, 2001*, LNCS 2057, pages 80–102. Springer, 2001.
31. S. Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.
32. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
33. M. Musuvathi and D. L. Dill. An Incremental Heap Canonicalization Algorithm. In P. Godefroid, editor, *Proc. of the 12th Int. SPIN Workshop (SPIN 2005), San Francisco, CA, US, August 2005*, LNCS 3639, pages 28–42. Springer, 2005.
34. V. Y. Nguyen. Optimising Techniques for Model Checkers. Master's thesis, University of Twente, Enschede, The Netherlands, December 2007.
35. V. Y. Nguyen and T. C. Ruys. Memoised Garbage Collection for Software Model Checking. In S. Kowalewski and A. Philippou, editors, *Proc. of the 15th Int. Conf. on TACAS 2009, York, UK, March, 2009*, LNCS 5505, pages 201–214. Springer, 2009.
36. S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In N. Halbwachs and L. D. Zuck, editors, *Proc. of the 11th Int. Conf. on TACAS 2005, Edinburgh, UK, April, 2005*, LNCS 3440, pages 93–107. Springer, 2005.
37. G. Ramalingam and T. W. Reps. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal Algorithms*, 21(2):267–305, 1996.
38. V. P. Ranganath, J. Hatcliff, and Robby. Enabling Efficient Partial Order Reductions for Model Checking Object-Oriented Programs Using Static Calculation of Program Dependencies. Technical Report SAnToS-TR2007-2, SAnToS Laboratory, CIS Department, Kansas State University, 2007.
39. Robby, M. B. Dwyer, and J. Hatcliff. BOGOR: An Extensible and Highly-modular Software Model Checking Framework. In *Proc. of ESEC/SIGSOFT FSE*, pages 267–276, New York, NY, USA, 2003. ACM Press.
40. T. C. Ruys and N. H. M. Aan de Brugh. MMC: the Mono Model Checker. *ENTCS*, 190(1):149–160, 2007. Proc. of Bytecode 2007, Braga, Portugal.
41. L. A. Smith, J. M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *Proc. of the 2001 ACM/IEEE Conference on Supercomputing (SC 2001)*, pages 8–8. ACM, 2001.
42. N. Tillmann and J. D. Halleux. PEX: White Box Test Generation for .NET. In B. Beckert and R. Haehnle, editors, *Proc. of the 2nd Int. Conf. on TAP 2008, Prato, IT, 2008*, LNCS 4966, pages 134–153. Springer, 2008.
43. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering (ASE)*, 10(2):203–232, April 2003.
44. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A Logical Encoding of the π-calculus: Model Checking Mobile Processes Using Tabled Resolution. *Software Tools for Technology Transfer (STTT)*, 6(1):38–66, 2004.
45. X. Yi, J. Wang, and X. Yang. Stateful Dynamic Partial-Order Reduction. In Z. Liu and J. He, editors, *Proc. 8th Int. Conf. on Formal Engineering Methods (ICFEM 2006)*, LNCS 4260, pages 149–167. Springer, 2006.
46. Bandera. http://bandera.projects.cis.ksu.edu/.
47. BLAST: Berkeley Lazy Abstraction Software Verification Tool. http://mtc.epfl.ch/software-tools/blast/.
48. Bogor. http://bogor.projects.cis.ksu.edu/.
49. Java PathFinder. http://javapathfinder.sourceforge.net/.
50. The Mono Project. http://www.mono-project.com.
51. MOONWALKER. http://code.google.com/p/moonwalker/.
52. .NET Languages. http://www.dotnetlanguages.net/.
53. SLAM. http://research.microsoft.com/en-us/projects/slam/.

Glossary

CIL	Common Intermediate Language (p. 3)
CLI	Common Language Infrastructure (p. 3)
config.	Configuration (p. 20)
DFS	Depth-First Search (p. 5)
DPOR	Dynamic Partial Order Reduction (p. 7)
IE	Instruction Executor (p. 5)
II	Interleaving Information (p. 12)
JGF	Java Grande Forum (p. 8)
JPF	Java PathFinder (p. 3)
JVM	Java Virtual Machine (p. 3)
LTL	Linear Temporal Logic (p. 1)
MGC	Memoised Garbage Collection (p. 15)
M&S	Mark & Sweep (p. 15)
POR	Partial Order Reduction (p. 6)
SDPOR	Stateful Dynamic Partial Order Reduction (p. 12)
SEH	Structured Exception Handling (p. 9)
SII	Summarised Interleaving Information (p. 13)
VES	Virtual Execution System (p. 4)
VM	Virtual Machine (p. 4)
XIL	abstraction of the CIL instruction language (p. 3)