Faculty of Electrical Engineering, Mathematics and Computer Science,
Formal Methods and Tools (FMT),
University of Twente

Master's thesis

Optimising Techniques for Model Checkers

by
Viet Yen Nguyen

December, 2007

Committee:
dr. ir. T.C. Ruys (Principal supervisor)
dr. ir. A. Rensink
prof. dr. ir. J.P. Katoen

Abstract

The Mono Model Checker (MMC) is a software model checker that can verify whether a .NET program contains assertion violations or deadlocks. It was developed as part of a Master's project by [1]. Much of its design was inspired by Java PathFinder, a software model checker for Java programs. This thesis is the result of the follow-up Master's project on MMC. The goal during this project was to improve MMC's ability to verify models with larger state spaces.

Enhancements have been added in all areas. For improving MMC's performance, both partial order reduction (POR) using object escape analysis and stateful dynamic POR have been added. These techniques reduce the size of a model's state space, and hence reduce the time and memory needed for its verification. For improving MMC's usefulness, .NET's exception handling has been fully implemented, more instructions have been added for increased .NET compliance and a comprehensive testing framework has been created. The latter employs Microsoft's own .NET virtual machine testing suite and has revealed numerous bugs. For improving MMC's usability, an error tracer has been added. It shows the sequence of instructions leading to the assertion violation or deadlock. This improves the user's understanding of detected errors.

Besides improving MMC with known techniques, during the course of this Master's project, three new techniques were developed that effectively decrease both time and memory needed for verifying a model. To decrease memory use, a collapsing scheme has been developed for collapsing the metadata used by a stateful dynamic POR. The reduction of memory is more than a factor of two. To decrease the verification time, a Memoised Garbage Collector 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 between 9% and 26%, depending on the model that is verified. The third technique is called incremental hashing, which also has a lower time-complexity. The key notion in this hashing scheme is that a hashcode of an array is recalculated using the old hashcode and the changes to the array. Though this technique was originally developed for MMC, we implemented in Spin, because the stake of hashing in MMC is near zero.

Experiments with the BEEM benchmarks showed up to 20% reduction in time when compared against Jenkins's hash function, which is an often used hash function in model checking due to its good uniformity.

We also benchmarked MMC against JPF and Bandera using the Java Grande Benchmark models. The results indicate that MMC is faster in terms of states per second, but JPF is more effective in reducing the state space. Bandera is on all fronts no match for MMC and JPF, as it is outperformed in every benchmark.

Preface

My interest in model checking began during the course "Concurrent and Distributed Programming", where I learnt about formal modal logic. I found its application in model checking fascinating, because the act of model checking forces one to resort to such logic. The enormous size of a model's state space is just not comprehensible without it. Besides this, model checking is also interesting for its usefulness. It helps us to spot errors in systems on which we are dependent for our well-being, like flight-controllers, energy-transportation, etc. It is important that these systems never malfunction, especially not from errors that are preventable, like design or implementation errors.

I learnt most of the principles of model checking from Theo Ruys. His enthusiasm for it is infectious. Theo also lectured me on many other computer science topics. His attitude for valuing technical merit has always suited me. Theo also showed me the value of good external support. Around the beginning of this project, a heavy injury deprived me of my drive. Theo nonetheless forced me to make initial progress on MMC. The hopeful results from it gave me the energy to speed up my recovery and to make more progress on MMC. I am grateful for his support and stimulating guidance at a time I needed it most.

I also thank Boudewijn Haverkort and Matthias Kuntz from the DACS research group for providing me access to their Linux cluster, which speeded up my experiments greatly. I also thank Henk van de Zandschulp from the EWI system administration group for his flexibility of running a second batch of experiments on a Windows cluster too many times. Also, I thank YourKit and Red Gate Software, for providing academic licenses of their .NET profilers, which helped me improve the speed of MMC a lot. I also thank Niels Aan de Brugh for sparring initial ideas on MMC. Last, but not least, I also thank Choong Wei Tjeng for the little moral support during our daily early morning chats.

Viet Yen Nguyen
Enschede, December 2007

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

abbreviation	word or phrase
API	Application Programming Interface
BEEM	BEnchmarks for Explicit Model checkers
BVT	Base Verification Tests
CIL	Common Instruction Language
CLI	Common Language Infrastructure
coll.	collision
collrate.	collision rate
config.	configuration
GC	Garbage Collecter
GF	Galois Field
II	Interleaving Information
inc.	incremental
Jen.	Jenkins
JGF	Java Grande Forum
JPF	Java PathFinder
LFCS	Labelled Formal Concurrent System
LTS	Labelled Transition System
MMC	Mono Model Checker
MolDyn	Molecular Dynamics
POR	Partial Order Reduction
P-E	Partial Order Reduction using object Escape analysis
P-D	Dynamic Partial Order Reduction
P-C	Combined Partial Order Reduction
OBDD	Ordered Binary Decision Diagram
obj.	object
rhs-value	right hand side value
SEH	Structured Exception Handling
SII	Summarised Interleaving Information
sv.	state vector
VM	Virtual Machine
VS2005	Microsoft Visual Studio 2005

Chapter 1

Introduction

1.1 Motivation

We heavily depend on automated systems for our most basic needs, like the energy-supply, monetary infrastructures and transportation safety systems. It is of great importance to ensure these systems keep running correctly. Any failure should be prevented beforehand. This requires extensive testing. Traditionally, systems are tested using sample testing. A sample test describes a scenario and then the system is checked whether it behaves correctly in that scenario. This kind of testing is simple, but is often incomplete. This is especially true for complex systems which may have many behaviours in many scenario's. Creating tests for them all is tedious. That is why software verification methods are preferred.

Contrary to sample testing, software verification methods are employed to verify whether a system works under all specified circumstances. This is especially challenging with concurrent systems, which prevalence is increasing as the current trend is to increase computer power by increasing concurrency. However, concurrent systems are also notoriously difficult to design well. The difficulty lies in the amount of behaviours of a concurrent system. Processes that comprise a concurrent program occur in parallel, and these processes influence and interact with each other. This causes a combinatorial explosion of its possible behaviours. For concurrent programs performing crucial functions, we want to be sure that all possible behaviours are those we desire.

An important part of the verification process is to express what is desired behaviour. This is captured by defining its correctness properties. An example property is that the program should never crash, or that it is always operational within 10 seconds after a power-failure. The collection of correctness properties is called the specification. To ensure that a concurrent program lives up to its specification, we want to verify each possible behaviour against the specification.

There are two approaches to do this, namely theorem proving and model checking. Theorem proving expresses both the specification and the concurrent program in formula. Logic is then applied to deduce whether the formula of the concurrent program are always equivalent to the formula in the specification. Theorem proving is often done manually and is therefore prone to errors. Furthermore, the process is slow and does not scale well with increasing system complexity.

Model checking on the other hand systematically explores all states a concurrent program can be in. This is often implemented as an automated process. The behaviour is defined as the whole of reachable states, called the state space. The model checker then verifies each state against the specification for correctness. Model checkers have been successfully used for the verification of industrial sized systems. For example, the Spin model checker has been used to verify the correctness of an important Dutch flood control barrier [59]. The same model checker has also been used for the verification of an enterprise-scaled phone switch by Lucent Technologies [34]. Another model checker, Java PathFinder, has been used to verify a prototype of NASA's Mars Rover [62]. In all these cases, errors have been uncovered which were unnoticed by traditional testing methods. The main limit of model checking is the size of the state space, which correlates with the complexity of the program to be verified. The complexity of a program increases with the amount of components in a concurrent program and the size of each component. It is this increasing complexity that makes effective model checking challenging.

This challenge is the main motivation for this Master's thesis. Concretely, the research goal as set during this Master's final project is as follows:

> **Research goal**: *Engineer techniques that make model checkers significantly more effective in verifying larger concurrent systems*

The result are three new techniques, namely Memoised Garbage Collection (see §5), Collapsion of Interleaving Information (see §4) and Incremental Hashing (see §6). Experiments conducted with these techniques support our findings and their usefulness. Besides these techniques, several existing techniques were implemented for improving our testbed model checker, the Mono Model Checker (see §3).

1.2 Model Checking

This section introduces the background of model checking. It first starts with the classical model checking approach, followed by a list of techniques for effective model checking and concludes with an introduction to software model checking.

1.2.1 Classical Model Checking

The practise of classic model checking is a three-phased process consisting of modelling, specification and verification:

Modelling Traditionally, systems have been difficult to verify due to limitations on time and memory. Therefore a model is derived from the system as a representative that is verified instead. The derivation can be done automatically or manually. A good model abstracts from irrelevant or unimportant details of the system.

A variety of model specification languages have been created that all have mutual advantages and disadvantages. It is out of the scope of this thesis to further discuss this in detail. In general, all existing model specification languages support a form of composition of concurrent components and means for interaction between those components. Yet, the expressiveness of a model specification language is limited when compared to ordinary programming languages because their semantics are closely tied to the mathematical models used by the verification algorithms [14].

Well-known model specification languages are Promela [33], famed for protocol verification, SMV [46], suitable for specifying hardware models and BIR [55], suitable for specifying object-oriented software.

Specification The specification states the correctness properties that a system must satisfy. There are two types of correctness properties, namely safety and liveness properties. Safety properties must always be true. Examples are "the system never deadlocks" or "the variable x is always between 0 and 42". Liveness properties must eventually be true (where the present is included is eventually) [8], like for example "on a stoplight, a red green is eventually followed by the yellow light".

For correctness properties to be applicable for automatic verification, they ought to be expressed in an unambiguous way rather than in natural language like in the provided examples. Notable approaches for expressing them are by assertions, like "$0 \leq x \leq 42$" or by more powerful temporal logic formulae. The latter can express a particular ordering of events without explicitly introducing time. Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are the most prominent temporal logics. The interested reader is referred to [35] for an in-depth treatment on LTL and CTL.

Verification Several verification methods exist for checking whether the model satisfies the specification. The preferred method is by using automatic verification algorithms, as they require little to no manual interference. A global overview of such an algorithm is shown in algorithm 1. This algorithm starts with the initial state s_0, and then recursively explores its successors states by following the transitions possible by the system in that state s, i.e.,

the *Enabled*(s). States that have been visited are matched by M and do not have to be explored further. The result of this algorithm is a graph, called the state space, which is a representation of the behaviour of the verified system.

Algorithm 1: Explore(s_0)

```
Data: backtrack stack B, state matcher M
push s_0 on B
while B is not empty do
    s ← popped element from B
    if s is not matched by M then
        update M such that it matches s
        foreach transition t from Enabled(s) do
            s' ← the successor state of s by exploring t
            push s on B
```

Two methods exist for encoding the state space, namely symbolically or explicitly. A state in symbolic state spaces represents a set of states. The symbolic representation tends to work out well for hardware circuits and protocols, as they often contain regularities which can be nicely captured symbolically [19]. Our focus is on explicit state model checking. In explicit state spaces, each state is represented individually. The drawback of purely explicit state model checking is that explicit states spaces explode even more in open systems, i.e., systems with interaction with the environment. Hybrid symbolic-explicit approaches have recently emerged in software model checking as the symbolic representation better suits the information retrieved from and sent to the environment [50].

The state space can be traversed in several ways. Depth-first search is usually the traversal order of choice because state spaces tend to be broad and deep, on which depth-first search is a resource-efficient traversal order. Algorithm 1 does a depth-first search as well. However, in some cases breadth-first search might be more effective because correctness properties might be dissatisfied "early" in the state space. Another traversal approach is the directed one. Directed strategies determine the order of traversal by heuristics meant to find dissatisfied correctness properties sooner [27].

When the search leads to a dissatisfied correctness property, a proof of it aids the user to locate the error. In model checking, such a proof is an error trace. An error trace is one run of the model that leads to the dissatisfied correctness property condition.

Tools for verification are Spin [33], an explicit state model checker that uses the Promela language, NuSMV [11], a symbolic model checker that uses the SMV language and Bogor [55], a hybrid model checker that uses

the BIR language, Java PathFinder, a model checker for Java and Groove, a graph transformation tool that is also usable as a model checker for model checking models represented as graphs and graph transformations[41].

1.2.2 Techniques for Effective Model Checking

Early model checkers were only capable of verifying small systems due to limitations of available computing resources at that time. This drove researchers to develop techniques allowing model checkers to verify larger systems without added computing resources. These techniques are categorised into performance improvements, reduction methods, approximative methods. Many of these techniques can be applied orthogonal on each other.

Performance Improvements The category of performance improvements contain techniques that improve the performance of the model checker by employing more efficient algorithms that either or both reduce time and memory usage.

In a typical search, the explorer (see algorithm 1) encounters states that it has seen before. This is because different runs of a concurrent system may lead to the same state. Exploration of its successors is not necessary, as these have already been visited. A state matcher is used to check whether a state has already been visited or not [23]. An usually used datastructure for a state matcher is a hashtable of visited states. Another approach to match states is using a state recogniser, like the minimised automata approach by [32]. Instead of storing a state individually, an automaton is built that recognises the seen states. Upon exploration of a new state, the automaton is updated such that it becomes recognisable.

Related to state matching using hashtables is state caching. It is a memory management technique that stores states to a certain threshold, like the available amount of memory. If that threshold is reached, older states, like those not on the backtrack stack, are freed from the cache. With state caching, the search risks to revisit already visited states, however the chance is less compared to statelessness [23].

State compression encompasses all techniques applied to reduce the size of individual states. Huffman encoding is for example used by [29] to reduce the size of a state represented by a bitvector. Another example that is combinable with Huffman encoding is state collapsion. This compression approach exploits the observation that a single transition only results to a small change between successive states. A greater part of the state is left unchanged. Hence, in a state space, states tend to share large parts of values. This observation can be put to an advantage by collapsing those parts, and store a reference to the collapsed part instead. This way, shared components are only stored once, thereby reducing memory [29].

Concurrent verification techniques exploit multi-processor systems by having the state space explored by concurrent explorers [31]. The best results have been delivered when each explorer explores a non-overlapped partition of the state space [22]. Similar to concurrent verification is distributed verification. Whereas concurrent verification techniques share the same memory, hence also the same state storage in case state matching is used, distributed techniques explore the state space over processing nodes which each have their non-shared memory. In practise, distributed techniques are applied over computers connected by a computer network.

Reduction Methods Reduction methods reduce the size of model's state space while ensuring that the verification results remain formally correct. Reduction methods also lead to increased performance.

Partial order reduction is the primary optimisation technique for model checking. Central to partial order reduction is the notion of dependent and independent actions (more about this in §3.3.2, 3.3.3 and §4). Independent actions are actions that do not affect other concurrent processes. Yet they incur interleavings (i.e., paths) in the state space which are not interesting (with respect to the specification) to explore. Partial order reduction techniques detect independent actions and defer their exploration until all dependent actions are explored, thereby reducing the size of the state space.

Thread symmetry reduction (also known as process symmetry reduction) techniques detect threads that are different by their process identifier and not by their semantics. Not all actions induced by symmetric threads in a particular state need to be explored. Only one of them suffices. The degree of exploitation of thread symmetries is inherent to the model [54]. The dining philosophers model is academically well known to be massively reducible by thread symmetry reduction.

Heap symmetry reduction techniques detect heaps that are semantically equivalent. They are useful in verification of dynamic memory systems, where a heap (or some other dynamic memory structure) is used by the model. In case of pointerless object-oriented systems, the reduction is possible because the index of an object on the heap does not matter, but it matters how objects are referenced. By mapping out all object references, one gets an object graph and heaps with the same object graph shape can be considered semantically equivalent, even though objects are differently indexed in the array. Heap symmetry reduction is achieved by canonicalisation of the heap, and use the canonicalised representation for storage and matching [54, 43].

Program slicing techniques reduces the model and therefore its state space. This exploits the observation that parts of the model may not be interesting for the behaviour to be verified. They can be sliced away. The resulting model is then verified instead. Usually the specification is used for determining the slicable parts [14].

Approximative Methods Contrary to exhaustive methods of model checking, approximative methods speed up the verification process by applying techniques that may cause the model checker to miss parts of the state space. Two well known approximative techniques are bitstate hashing and hash compaction.

Bitstate hashing is a form of state matching where states are not explicitly stored, but their position in the hashtable is flagged by a boolean. Upon a collision, the assumption is made that the current state is already seen. This is not certain, as the current state cannot be matched using byte-for-byte equivalence. That is why bitstate hashing is approximative. An optimisation of bitstate hashing is k-fold bitstate hashing, where k independent hashfunctions are used to flag the bits in k hashtables. A state collides if its hashcodes all return indices that are already flagged [30].

Hash compaction [65] is a technique that is closely related to bitstate hashing. Two independent hashfunctions are used. One is used for calculation of the position in the hashtable. The second is used to derive a compacted hash that is representative for the state in question. It is then stored at the position determined using the first hash. A state is matched if both the position and the compacted hashes are equivalent. This technique is approximative because while the compacted hashes may be equivalent, their states from which they are derived might not [42].

1.2.3 Software Model Checking

Model checking was deemed inadequate for verification of software applications due to the high level of detail found it. In the past decade, the development of effective techniques as described in §1.2.2 and advances in hardware led industry and academia to study and develop software model checkers, where the software is the model itself.

The foremost advantage of software model checkers is that there is no need to manually create a model of the system. Faults not existing in the system tend to slip in easily through the abstraction process. Also, creating a model that contains all interested behaviour of the system has proved to be difficult. Thirdly, a correctly verified model does not implicitly mean that the system satisfies the specification. That is why classic model checkers has proved to verify designs with success, whereas software model checkers can be employed to verify implementations.

A disadvantage of software model checkers is the on average larger state size due to the high level of detail found in systems. Also, software model checkers need to cope with dynamic allocations and complex datastructures. These are abstracted away in classic model checking. Another disadvantage is that most of the actions that systems can do are independent. Partial order reduction is therefore a necessity to deal with state space explosion. A third disadvantage is that systems are rarely closed systems. A software

model checker either deals with that, or limits the set of verifiable systems to the set of closed systems.

Most techniques outline in §1.2.2 are effective and, with some modification, applicable to software model checkers. Besides those, through observations of the early software model checkers, researchers identified issues and optimisation opportunities that are specific to software model checkers. Heap symmetry reduction is for example one of them. The Java PathFinder team identified that for their heap canonicalisation technique, garbage collection is needed. They also developed backtracking optimisations by creating a delta (like a patch) between collapsed states on the backtrack, and upon backtracking, only restore the delta instead of the whole collapsed state [43]. Both [18] and [43] observed the use of object escape analysis for detecting independent actions for driving partial order reduction. In [21], a partial order reduction method called dynamic partial order reduction is described that further reduces the state space upon previous partial order reduction methods.

To name a handful of existing software model checkers, like Java PathFinder [61], for the verification of Java programs, Bandera [14], which can also verify Java programs and StEAM [44], for the verification of C and C++ programs.

1.3 Mono Model Checker

The first version of the Mono Model Checker (MMC) was developed as part of a Master's thesis at the University of Twente. Its development was initiated to gain experience with designing and implementing a software model checker and to provide an inhouse sandbox for further research. The result was a competitive software model checker that is performance wise on par with today's software model checkers [1]. We further improved MMC and used it as testbed for the techniques developed for this Master's thesis.

MMC verifies Common Intermediate Language (CIL) assemblies, better known as .NET programs. The CIL and its semantics are the core of the Common Language Infrastructure (CLI), better known as .NET. The CLI is standardised under ECMA 335 and ISO/IEC 23271:2006. It is designed to be language-agnostic and today, many variants on many languages as C, C#, C++, Java, Visual Basic, Prolog, Python and Ruby have been created that compile to CIL. Principally, MMC can model check them all, although only C# programs have been used in tests so far.

CLI is creation of Microsoft, but its specification is open and others are free to implement it themselves. A group of open source programmers organised the Mono project to provide an free and open source implementation of CLI [15]. Although the CIL is designed as such that CIL programs are exchangeable with different implementations of CIL, the internals of Mono's

virtual machine differ from Microsoft's implementation in such a way that their class libraries are completely different, although they share same the API. MMC mimics the semantics of Mono's implementation because their internal calls semantics were easily determined by reading the source code.

To prove that the CLI is platform-independent, Microsoft provides its own implementation of the CLI for FreeBSD under the shared source license[1]. This implementation is called the Rotor distribution [58], and shares a part of code with the current Windows implementation of .NET. We attempted to support Rotor, and hopefully the current Windows implementations as well with MMC by mimicking their internal call semantics. This failed because of the lack of documentation, up to date source code and support from Microsoft. However, we were able to port MMC to Microsoft's .NET platform, without much effort. In order to run MMC under a Microsoft VM, one has to set the MONO_HOME environment variable to the Mono directory in order to ensure that MMC uses Mono's class library to link the to be verified program to.

From the perspective of model checking techniques and architecture, MMC has a close resemblance with JPF. Many techniques used in JPF are also included in MMC, like statefulness, state collapsion, a form of partial order reduction, heap symmetry reduction, deadlock checking, checking of assertion violations and backtracking by delta's. More on MMC is described in [1].

1.4 Java Grande Benchmarks

The techniques described in this thesis are assessed using experiments. Instead of using small benchmarks that synthesise a small scenario, we used benchmarks that resemble real life applications. This is more in line with the research goal, because such applications have a higher complexity and their state space is therefore larger. Also, we purposely did not create our own benchmarks, of which the results may be interpreted with bias. We used an existing benchmark suite developed for the scientific community called the Java Grande Forum Benchmarks (JGF benchmarks) [57].

One part of this suite are the parallel benchmarks, which are 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

[1] The shared source license is not an open source license.

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 1.1: Metrics of the MolDyn en RayTracer benchmarks.

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." [57].

- **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 balance." [57].

The third benchmark is the MonteCarlo benchmark. It however uses file input/output, which is not (yet) supported by both MMC and Bandera, which is why we left it out for comparison.

The benchmark are parametrised with two parameters: the number of threads and the data size. Throughout this thesis, 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. Additionally, to get an idea of the models's size and complexity, its metrics are shown in table 1.4.

As the benchmarks are written in Java, we had to convert them to C#. This was done 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, MMC 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. Data races in the Java Grande Benchmarks have also been detected by [20]. 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.

1.5 Organisation of this Thesis

Chapter 2 describes the formalisms used throughout this thesis. A formal definition of a model, the Labelled Formal Concurrent System is presented. Its semantics is formalised as a Labelled Transition System. The memory semantics of a state is formalised by an object graph.

Chapter 3 describes several improvements to MMC, namely a redesigned explorer, dynamic partial order reduction, partial order reduction using object escape analysis, exception handling, testing framework, error tracing, ex post facto transition merger and performance improvements by profiling.

Chapter 4 describes a compression technique that is used for collapsing interleaving information. This information is collected and used for the stateful variant of dynamic partial order reduction algorithm. The technique is evaluated with experiments on the Java Grande Benchmarks.

Chapter 5 describes the Memoised Garbage Collection algorithm, which is designed for use in software model checkers. This technique has been evaluated with experiments of the Java Grande Benchmarks as well and the results and its discussion are included.

Chapter 6 describes the incremental hashing scheme. It is a hashing scheme suited for model checkers in which the hash function has a big stake in the total running time. Its effectiveness is supported by results from experiments with a modified version of the Spin model checker.

Chapter 7 describes results from benchmarking two competitive model checkers against the Mono Model Checker, namely Java PathFinder and Bandera. The Java Grande Benchmark suite is used as input models.

Chapter 8 summarised this work and highlights the most promising directions for future work.

Chapter 2

Concurrent Software Semantics

This chapter presents a formalism for modelling concurrent software systems (i.e., model), a formalism of its behaviour (i.e., state space) and a formalism to model its memory semantics (i.e., object graph). These formalisms shall be used throughout this thesis.

2.1 Model Definition

Within the object-oriented software domain, sequential processes in a concurrent software systems are abstracted as threads. Threads are assumed to be finite-state and deterministic. Threads communicate by accessing objects and interact by synchronisation mechanisms on objects. This formal model used throughout this thesis is a Labelled Formal Concurrent System (LFCS), which is inspired by [23]:

Definition 2.1.1. A labelled formal concurrent systems, further referred to as a system, is a tuple $\langle P, O, T, \lambda, s_0 \rangle$ where

- P is a finite set of threads.
- O is a finite set of object entities.
- T is a finite set of transitions.
- $\lambda : T \mapsto \Sigma$ is a labelling function associating a label from an alphabet of instructions Σ with each transition of T.
- s_0 is the initial state of the system.

A thread $TH \in P$ is represented as a finite nonempty set of program counters, $TH = \{p_1, \ldots, p_n\}$. The set denotes the possible local states the thread can be in. Threads are pairwise disjoint.

An object entity $OE = \{o_1, \ldots, o_n\}$, $OE \in O$, is a finite nonempty set of object states which it may take during its lifetime. Object entities are

pairwise disjoint. An object entity can change state because of a transition. In the LFCS, O captures only global objects entities. Local object entities like local variables and method arguments are abstracted away. Also, for ease, we simply refer an object state as an object.

A state of the LFCS is an element in the Cartesian product of threads and objects: $S \subseteq TH_1 \times \ldots \times TH_n \times OE_1 \times \ldots \times OE_m$ where $n = |P|$, $m = |O|$, $TH_i \in P$ and $OE_j \in O$, $0 \leq i \leq n$ and $0 \leq j \leq m$. The initial state s_0 is an element of S.

A transition $t \in T$ models the execution by one thread. It is a tuple $\langle p, g, c, p' \rangle$. Program counters p and p' are elements of a thread $TH \in P$. The predicate $Thread(t)$ is used to get thread TH from t. The program counter p' is the incremented program counter upon the transition by that thread. A transition is enabled in state s if p matches with the thread's current program counter and if the guard g is true, which is a conjunction of boolean conditions upon the objects in s. The command function $c(o_1 \times \ldots \times o_i \times \ldots \times o_n) = (o_1 \times \ldots \times o'_i \times \ldots \times o_n)$ models a change to one object entity $OE_i \in O$ from object state o_i to o'_i, where both o_i and o'_i are in OE_i. It is possible that the function c does not change any object entity, in case transition t is a no operation instruction (NOP). Since at most one object entity is involved in a transition, we shall denote this object entity by $Object(t)$.

To the reader familiar with the nomenclature in [1], the semantic link between a LFCS and a program in the .NET model checking domain is as follows. Object entities in O model both heap- and static object entities. The definition of O also includes thread-unshared objects (discussed further in §3.3.2). A transition is the equivalent of a CIL instruction. Note the definition of a LFCS abstracts from dynamic thread spawning, dynamic memory allocation and implicit object wait queues; including their notions in the definition does not add value throughout this thesis.

2.2 Model Semantics

The behaviours of a LFCS are modelled as a directed graph called the state space. Successive states s and s' in the state space are linked by the transition t that led s to s' and is notated by $s \xrightarrow{t} s'$. Formally, a state space is a labelled transition system (LTS):

Definition 2.2.1. The behaviour of a LFCS $\langle P, O, T, \lambda, s_0 \rangle$ is represented by LTS $A_G = \langle \Sigma, S, \Delta, s_0 \rangle$ where:

- Σ is the alphabet from λ.
- S is the set of states.
- $\Delta \subseteq S \times \Sigma \times S$ is the transition relation defined as:

$$\langle s, a, s' \rangle \in \Delta \Leftrightarrow \exists t \in T \bullet s \xrightarrow{t} s' \wedge a = \lambda(t)$$

- s_0 is the initial state of the LFCS.

A transition in Δ corresponds to the execution of a transition $t \in T$. To avoid confusion, transitions in Δ are from now on referred to as global transitions while transitions in T are referred to as transitions. The global transition relation is bound to the deterministic nature of threads. If there exists a transition $t = \langle p, g, c, p' \rangle$ enabled in state s, then another transition $t' \neq t$ having the same p cannot be enabled in s. A consequence of this, is that enabled transitions in s, denoted by $Enabled(s)$, can also be uniquely identified by the threads involved in the enabled transitions. This notion is used throughout this thesis if it eases the context.

A path π in A_G is the sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} s_n$ such that for $0 < i < n$ holds $\langle s_i, \lambda(t_i), s_{i+1} \rangle \in \Delta$. The transitions involved in the path is a transition sequence $t_1 \ldots t_{n-1} \in T^*$. The transition sequence can be mapped by the labelling function λ to the word $a_1 \ldots a_{n-1}$. Also, similar to the transition sequence, we define a state sequence $s_1 \ldots s_{n-1} \in S^*$ from π. Paths, words, transition- and state sequence are mutually deducible given A_G, hence each notation is used interchangeable in this thesis when suitable within the context. The notation $s \overset{w}{\Rightarrow} s'$ means that s' is reachable from s via a finite transition sequence or word w.

The state space is constructed by exploring all states that are reachable from the initial state. This algorithm is discussed in §3.2.

2.3 Object Graph Semantics

In anticipation of a discussion of the Memoised Garbage Collection in §5, this section introduces an object graph notation and its semantics. The object graph[1] models a part of a state from the state space by mapping the relations between the values held in the objects.

The LFCS abstracted peculiar details of an object entity, thus also that of an object. More specifically, an object entity is a composition of a finite amount of field entities. A field entity F is a set of field values that it may held. For ease, a field value is just called a field. An object is therefore a tuple of fields $\langle f_1, \ldots, f_n \rangle$, where n is the amount of field entities, also referred to as the size of an object.

In §2.1, a transition only changes the state of one object entity. In fact, a transition actually only changes the state of one field entity, and therefore the state of the object entity. The field involved in a transition t is denoted by $Field(t)$.

A field f_i may hold a reference to another object. In an object graph, these references between objects are mapped as a directed graph.

[1] The object graph is also called heap graph in literature.

Definition 2.3.1. An object graph $\langle V, E, v_0 \rangle$ of a state $s = \langle \ldots, o_1, \ldots, o_m \rangle$ derived from LFCS $\langle P, O, T, \lambda, s_0 \rangle$ is a directed graph where:

- $V = \{v_0, o_1, \ldots, o_m\}$ represents objects.
- $E \subseteq V \times V$ represents object references.
- v_0 represents the fictive root.

Given a state s, the object graph is built up as follows. The callstacks of the threads are traversed for references to objects. Any object referenced in there is a child of the fictive root. Then, for every object $o \in V$ its fields are read, and the objects referenced by o become its children. This relation is captured by E, where a pair $\langle o, o' \rangle \in E$ states that o' is a child of o, and o is the parent of o'. The easier readable notation $o \rightarrow o'$ is equivalent to $\langle o, o' \rangle \in E$.

An object $o_i \in V$ is said to be reachable from $o_j \in V$ iff $o_i \rightarrow o_j$ or if there is a reference chain $o_i \rightarrow \ldots \rightarrow o_j$. The shorter notation $o_i \Rightarrow o_j$ is also used to express this notion of reachability. The predicate $Parents(o)$ is the set of parent objects of o. The predicate $Childs(o)$ is the set of child objects of o.

Chapter 3

Improving the Mono Model Checker

This chapter describes the engineering efforts on MMC that have advanced its usefulness. The result is an improved version of MMC, which we versioned 1.0. The improvements are built upon MMC 0.5, which is described by [1]. The reader is recommended to read it first, because we shall build upon the concepts and terminology introduced there.

3.1 Overview of Improvements

This section sets out the metrics of MMC 0.5 against our improved MMC, MMC 1.0 (see table 3.1). This is followed by a list of MMC 0.5's features, which is extended by the list of improvements upon it by MMC 1.0.

Metric	**MMC 0.5**	**MMC 1.0**
#Lines of code	15834	17052
#Classes	284	274
#Methods	2701	1907
#Statements	4348	5034
#Source size in Kb.	451	475
Implemented CIL instructions	58 out of 83	74 out of 83
Supported internal calls	13	33

Table 3.1: Metrics of MMC 0.5 set against MMC 1.0.

MMC 0.5's list of noteworthy features is as follows:

- Verification of deadlocks and assertion violations.
- Structured state collapsion.
- Backtracking by delta's.

- Heap canonicalisation.
- Reference counting garbage collection.[1]
- Mark & Sweep garbage collection.
- Partial order reduction by distinguishing thread-safe and thread-unsafe instructions.

In MMC 1.0, the following improvements were added upon those above:

- Memoised garbage collection (see §5).
- Partial order reduction using object escape analysis (see §3.3.2).
- Stateful dynamic partial order reduction (see §3.3.3 and §4.2.1).
- Collapsion of Summarised Interleaving Information (see §4.2.3).
- Error tracer (see §3.6).
- Testing framework (see §3.5).
- A priori statement merging (see §3.3.2).
- Ex post facto statement merging[2] (see §3.8).

3.2 Redesigned Explorer

The design of the original explorer was not extensible enough for our purposes. Our first attempts of implementing partial order reduction (see §3.3.3 and §3.3.2) and error tracing (see §3.6) proved to be difficult and error-prone. Eventually, we decided to save our time by redesigning a new explorer from scratch that better suits our needs. The remainder of this section replaces §4.3.1 and §4.3.2 from [1].

3.2.1 Definitions

The following are additional definitions upon those mentioned in §2.

The working set, denoted by $Working(s)$, is the set of transitions that have not been explored yet in state s. The done set, denoted by $Done(s)$, is the set of transitions that have already been explored in state s. The working and done sets for a state s are always disjoint.

3.2.2 Algorithm

From a global point of view, the redesigned explorer works in five repeating phases:

1. Forwarding
2. Error checking
3. State matching

[1] In MMC 1.0, reference counting became defunct. It has not been prioritised to be fixed, as it lacks preciseness (see §5.2), thereby causing possible state space explosion.

[2] Ex post facto statement merging is combinable with a priori statement merging

4. Backtracking
5. Next transition decision

Our explorer that is based on this phased approach is shown in algorithm 2. It is a stateful depth-first search explorer. Additional features like heap symmetry reduction, state collapsion and backtracking by delta's [1] are left out of this description. They are however included and enabled by default in MMC. As this thesis does not further builds on them, their notions are excluded for the sake of overview.

Algorithm 2: Explorer()

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

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[3] instructions by the same thread. A forward step also ends if an endstate of a thread is reached.

After a forward step, a successor state is reached that becomes the current state. It is checked for assertion violations and deadlocks. The assertion violation check and the deadlock detection algorithm is the same as in §4.3.3 of [1]. Upon an error, MMC will by default stop exploration and pass control to the error tracer (see §3.6).

The state matching 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.

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: either the state is a revisited state or 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.

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. MMC 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 2) have been added to the exploration algorithm. They are used for error tracing (see §3.6), stateful dynamic partial order reduction (see §3.3.3) 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 Partial Order Reduction

Partial order reduction (POR) may reduce the state space to be explored, while maintaining correctness of the verified specification. MMC already performs a limited form of POR by merging thread-safe instructions as one step (see ExecuteStep in §3.2). We implemented two much more effective POR techniques in MMC, namely POR using object escape analysis [23, 43,

[3]Thread-safe instructions are called safe instructions in [1], yet we call them thread-safe to distinct it from safe and unsafe code as specified by the CLI.

18] and dynamic POR [21]. Our approach is based on the principles outlined in those papers. This section only focuses on implementation-specific details.

3.3.1 Definitions

A typical state space contains many paths that are semantically equivalent with respect to the specification. Yet, the explorer algorithm in §3.2 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. This principle is formalised by the following:

Definition 3.3.1. Given a LFCS $\langle P, O, T, \lambda, s_0 \rangle$. The relation $D \subseteq T \times T$ is a minimal dependency relation for the LFCS iff for all $t_1, t_2 \in T$, $\langle t_1, t_2 \rangle \notin D$ (t_1 and t_2 are independent) implies that the following holds for all states s in S of the associated LTS:

- Independent transitions can neither disable nor enable each other: if $s \xrightarrow{t_1} s'$, then t_2 is enabled in both s and s'
- There is a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$

Instead of exploring the full enabled set, the minimal D can be used to calculate a perfect persistent set. This is a subset of the enabled set that contains only mutually dependent transitions. The minimal D is however only determinable when the full state space is known. Calculating it during exploration would defy the purpose of POR.

To achieve reduction, the dependency relation does not have to be minimal. It suffices to determine a D' that is a superset of the minimal D. Such a D' can be determined using information other than the full state space, like dependency information inferred from a particular state or from a set of already explored states. Such a D' is useful for calculating supersets of the perfect persistent set, which we just call a persistent set[4].

For example, an other less perfect, but good, known way to determine dependencies is by tracking the nature of the transition and the object it accesses. If two transitions are enabled and they write to the same object, they can be assumed dependent. This assumption is false if both transitions write the same value. Nevertheless, it is a safe assumption to make. An other approach is to determine the objects that are only accessible by one thread for a part of the state space. Accesses to this object are ensured to be independent for that part of the state space. Both approaches towards the determination of D are used in MMC.

[4] A persistent set is also called an ample set

3.3.2 Object Escape Analysis

The calculation of persistent sets using object escape analysis in software model checking was pioneered by the JPF team [43] and was subsequently improved by [18] and [53]. Although these improvements work, we only implemented JPF's original partial order reduction technique in MMC. This is for two reasons. For one, it was relatively easy to implement once the explorer was redesigned (see §3.2). And two, it could be used to assess the impact of the more recent dynamic POR approach (see §3.3.3).

Dependency relation The first step is the calculation of the dependency relation D. In both JPF and MMC, 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. We use an object escape analysis algorithm to determine those objects. See algorithm 3.

The 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. It first initialises every object 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, the objects in *toRecurse* 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 call to algorithm 3, objects that are attributed 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 only unescaped objects are automatically assumed thread-unshared. 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. Even though this assumption does not always hold, we will see later in the experimental results that this POR technique reduces the state space quite effectively.

Persistent set Once all objects are marked either thread-shared or thread-unshared, we use that information to calculate the persistent set. This goes

Algorithm 3: MarkThreadSharedObjects

Data: Object graph $\langle V, E, D, v_0 \rangle$, LFCS $\langle P, O, T, \lambda, s_0 \rangle$, stack *toRecurse*

```
// Initialise every object to be unmarked
```
foreach *Object o in V* **do**
    $Attribute(o) \leftarrow$ UNMARKED
foreach *Thread p in P* **do**
    foreach *Object o referenced from the callstack of p* **do**
        SetAttribute(o, p)

```
// Recursively traverse childs
```
while *toRecurse is not empty* **do**
    $o \leftarrow$ pop *toRecurse*
    foreach *Object* o_c *in* $Childs(o)$ **do**
        SetAttribute(o_c, $Attribute(o)$)

Algorithm 4: SetAttribute(o, p)

Data: stack *toRecurse*

if $Attribute(o) =$ UNMARKED **then**
    $Attribute(o) \leftarrow p$
    push o on *toRecurse*
else if $Attribute(o) \neq p \wedge Attribute(o) \neq$ SHARED) **then**
    $Attribute(o) \leftarrow$ SHARED
    push o on *toRecurse*

as follows: If the enabled set contains any transition that accesses an 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.

A priori transition merging There is an opportunity for optimisation with this approach. In case a singleton persistent set is formed for a state, we know for sure that state has only one successor state. This allows us to merge that transition with its predecessor. This observation can be extended for multiple subsequent states that have singleton persistent sets. By merging those transitions, intermediate state canonicalisation, state collapsion and state storage is prevented, therefore decreasing running time. In MMC, this transition merging is implemented similar to the merging of thread-safe instructions. One ExecuteStep is called first. This is followed by object escape analysis, calculation of a persistent set, and if it is a singleton, call another ExecuteStep. The latter steps are repeated until a non-singleton persistent set is calculated.

There is one drawback of transition merging. For some models, it is possible that the POR merges an infinite sequence of ExecuteStep. The model checker will then livelock. This happens if the explored state space also contains a sequence of transitions that purely operates on thread-unshared (or even worse, local) objects. Such states spaces are associated with models that have threads that can starve because of the lack of fairness. This can be detected. A technique for this is described in the future work (see §3.9), as time-constraints prevented us from implementing it. Another available method to verify models with starvable threads in the absence of fairness is by disabling POR using object escape analyses.

Another issue with a priori transition merging is that we have to be sure that the dependency relation used is consistent, i.e., that objects marked thread-unshared are truly thread-unshared. This may deviate because of writes to object field instances and static class fields. More particularly, if an object reference to a thread-unshared object is written to a field whose associated object is thread-shared, then the thread-unshared object becomes thread-shared. In order to ensure the consistency of the dependency relation, algorithm 3 is triggered when (i) the written value is an object reference to a thread-unshared object, and (ii) the object associated to the written field is known as thread-shared.

Miscellaneous details Some last implementation notes: just as in JPF, object escape analysis (algorithm 3) in MMC piggybacks with the garbage collector, which is used for heap symmetry reduction. This saves time by traversing the object graph only once. Second, objects referenced from the

static area are considered as thread-shared objects. They are pushed on the recurse stack before the callstacks are traversed. Third, ExecutePORStep is implemented similar to ExecuteStep. At runtime, a parameter can be given to choose between either two. The explorer uses that parameter to make the right call.

3.3.3 Dynamically Tracked Dependencies

A POR technique that differs from POR using object escape analysis (see §3.3.2) is called POR using dynamically tracked dependencies, also called dynamic POR. It is a more recent POR technique by [21]. That description of dynamic POR worked only correctly for stateless exploration. [66] and [53] proposed a variant on dynamic POR that works for stateful exploration, dubbed stateful dynamic POR. We implemented the latter in MMC.

First, we recap the general idea behind stateless dynamic POR, followed by the pieces of algorithms that compose the stateless dynamic POR framework in MMC. The parts that extend this stateless algorithm for stateful dynamic POR is described in §4.2.1.

Reasons for dynamic POR The POR approach with object escape analysis is very imprecise. 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. Dynamic 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.

Dependency relation How does dynamic POR work? Dynamic POR 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

$Working(s')$. If $Thread(t)$ is not enabled in s', then the whole enabled set is added to the $Working(s')$. An outline of the algorithm is shown in algorithm 5, which is hooked to OnNewState in the explorer (see algorithm 2). Note that algorithm 5 calls ExpandSelectSet, which is described in algorithm 6.

Algorithm 5: OnNewState(s)

Data: backtrack B

```
1 Working(s) ← an arbitrary element from Enabled(s)
2 foreach transition t in Enabled(s) do
3     ExpandSelectedSet(Field(t), Thread(t))
```

Algorithm 6: ExpandSelectedSet(F, tid)

```
1 s' ← L(F)
2 if ∃t' ∈ Enabled(s') ∧ Thread(t') = tid then
3     if t' ∉ Done(s') then
4         add t' to Working(s')
5 else
6     add Enabled(s')\Done(s') to Working(s')
```

Line 1 of ExpandSelectedSet makes a call to L. That call retrieves the most recent state on the backtrack stack that was accessed by the field entity given as an argument. This is implemented by a hashtable that maps each field entity to a stack of working sets as the internal datastructure of the L. It is updated every time a new transition is chosen (see algorithm 7) and upon backtracking (see algorithm 8).

Algorithm 7: ThreadPicked(s, t)

Data: hashtable L

```
1 push Working(s) on L(Field(t))
```

C3 proviso The original dynamic POR technique works only correctly on verification of deadlocks and safety properties in acyclic state spaces. Preserving correctness of verifying safety properties on cyclic state spaces is little more problematic. The problem is referred to as the "ignoring problem". The solution is an additional proviso called the C3 proviso in [12]. Both [66] and [53] independently recognised that this same proviso must be applied for dynamic POR if safety properties are verified in a cyclic state space. The C3 proviso simply states that if a revisited state is still on the

Algorithm 8: Backtracked(s)

Data: hashtable L

```
1 t ← most recently explored transition from s
2 pop L(Field(t))
```

stack, i.e., a cycle, then the enabled set minus the done set should be added to its working set (see algorithm 9).

Algorithm 9: OnSeenState(s)

```
1 if s is on B then
2     Working(s) ← Enabled(s)\Done(s)
```

3.3.4 Combining the two POR techniques

POR using object escape analysis and (stateful) dynamic POR are combinable. 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 dynamic POR is activated as well, the dynamic POR 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.

3.3.5 Experimental Evaluation

To evaluate the effectiveness of the mentioned POR approaches, we conducted experiments with the Java Grande Benchmarks (see §1.4). We ran a series with POR completely disabled (¬P), a series with purely POR using object escape analysis (P-E), a series with purely stateful dynamic POR (P-D) and a series that uses a combined POR approach (P-C). All series were ran on a 2.4 GHz systems equipped with 2 GB memory. We set the maximal running time to 10 hours, the memory limit to 1.5 Gb and the ex post facto transition merger (see §3.8) memory threshold also to 1.5 Gb.

This experiment was planned to and initially ran on Mono's VM, but the results from it were unusable. Mono's GC crashed on many configurations of both MolDyn and RayTracer, making it impossible to run a complete series of trials. The crashes were caused by two known bugs in Mono's GC[5]. We decided to rerun all experiments under Windows XP installed with .NET 3.0, as Microsoft's VM does not have that bug. All subsequent experiments with the JGF benchmarks were run in the latter setup as well.

[5]These bugs are knows under bug #324318 and #325386 within the Mono project.

The results are shown in tables 3.2 and 3.3. The time column is the verification time in seconds. A verification that has run out of time is indicated by "o.t.". The memory column is the maximal memory used during verification in megabytes. A verification that has run out of memory is indicated by "o.m.". The states column is the amount of states in the state space. The revisits column is the amount of states revisited during verification. The states stored column is the amount stored in the hashtable. This may differ from the amount of states in the state space due to the ex post facto transition merger. The max. DFS stack column is the maximal depth of the state space reached during verification. Note that the latter four columns are represented in thousands for the results from MolDyn benchmarks (see table 3.2). The state stored/Mb. column gives an indication of the memory utilisation efficiency. The states/sec. column is the amount of states processed per second during verification. It is calculated by adding the amount of states with the revisits and have that divided by the verification time.

We shall first look at the amount of reduction achieved. The MolDyn results show (see table 3.2) that the absence of POR explodes the state space as every configuration runs out of memory with POR disabled. POR using object escape analysis improves the situation a lot, as the state space is enormously reduced for all configurations when it is enabled. It furthermore enables the full exploration of the first configuration. If stateful dynamic POR is used on its own, the first model also becomes fully verifiable, yet its achieved reduction is less than runs with POR using object escape analysis. The reason of this is the a priori transition merger that is driven by POR using object escape analysis. It is extremely effective in cutting down the amount of states. This is reflected in the max. DFS stack size, which is less when POR using object escape analysis is enabled. The same principle is not usable for dynamic POR, as every thread-safe instruction encountered is always considered a scheduling point at which state collapsion and garbage collection is run. When the combined POR approach is used, the state space is further reduced, but not much. This is however different when we look at the results with RayTracer (see table 3.3). Here, three configurations are fully verifiable, and the combined POR approach reduces the space space by nearly a half over POR using object escape analysis. This observation is in line with that of [53]. They observed as well that the effectiveness of the combined POR approach depends on the structure of the model.

When it comes to pure raw performance (in terms of states per second), we see in both tables 3.2 and 3.3 that verifications with POR using object escape analysis and verifications with POR disabled are significantly faster. This reflects the overhead of the stateful dynamic POR algorithm. We ran the profiler on this to understand this, and we observed that the overhead is mostly caused by maintaining the L map (see algorithm 6) and the handling and maintenance of metadata used by stateful dynamic POR (see §4.2.1).

When it comes to memory overhead, we also see in table 3.2 that the use

config.	POR type	time (sec)	memory (Mb.)	states ($\cdot 10^3$)	revisits ($\cdot 10^3$)	states stored ($\cdot 10^3$)	max. DFS stack ($\cdot 10^3$)	states stored/Mb.	states/sec
2-1	P-C	458	1470	1482	1063	1482	28	1008	5560
	P-E	142	643	1533	1474	1533	28	2386	21246
	P-D	1450	1535	7529	1092	2076	93	1352	5945
	¬P	403	o.m.	3799	3690	3735	93	2490	18564
2-2	P-C	1553	o.m.	1926	788	977	196	651	1748
	P-E	563	o.m.	1939	1741	1930	196	1286	6536
	P-D	120	o.m.	512	0	512	512	341	4251
	¬P	137	o.m.	754	0	754	754	503	5501
2-3	P-C	72	o.m.	249	0	249	249	166	3475
	P-E	145	o.m.	378	0	378	378	252	2613
	P-D	108	o.m.	254	0	254	254	169	2343
	¬P	99	o.m.	383	0	383	383	255	3862
3-1	P-C	1038	o.m.	2724	3018	1662	66	1108	5531
	P-E	590	o.m.	3359	6459	3357	66	2238	16652
	P-D	o.t.	1512	190546	51544	27049	140	17886	6725
	¬P	431	o.m.	3198	6011	3198	140	2132	21
3-2	P-C	98	o.m.	327	0	327	327	218	3324
	P-E	78	o.m.	453	0	453	453	302	5817
	P-D	53	o.m.	335	0	335	335	223	6264
	¬P	82	o.m.	458	0	458	458	305	5567
3-3	P-C	68	o.m.	151	0	151	151	101	2238
	P-E	73	o.m.	215	0	215	215	144	2948
	P-D	54	o.m.	168	0	168	168	112	3129
	¬P	73	o.m.	227	0	227	227	152	3102

Table 3.2: MolDyn results with no POR at all (¬P), purely POR using object escape analysis (P-E), purely dynamic POR (P-D) and the combined POR approach (P-C).

config.	POR type	time (sec)	memory (Mb.)	states	revisits	states stored	max. DFS stack	states stored/Mb.	states/sec
2-1	P-C	1	36	844	579	844	73	23	1198
	P-E	2	36	1505	1365	1505	73	41	1733
	P-D	593	1546	752107	6711	229767	6602	149	1280
	¬P	201	o.m.	364997	356998	362684	6602	242	3597
2-2	P-C	113	655	65923	53264	65923	3173	101	1055
	P-E	142	729	134910	128570	134910	3173	185	1857
	P-D	o.t.	1529	29029628	315742	471622	12897	308	815
	¬P	236	o.m.	354620	341712	350169	12897	233	2949
2-3	P-C	o.t.	1373	97233	24289	97233	24362	71	3
	P-E	o.t.	821	64867	32395	64867	32468	79	3
	P-D	80	o.m.	75931	0	75931	75931	51	954
	¬P	52	o.m.	121473	0	121473	121473	81	2336
3-1	P-C	68	475	53631	71076	53631	145	113	1842
	P-E	163	950	115113	215103	115113	148	121	2025
	P-D	516	o.m.	434336	308140	166874	10966	111	1438
	¬P	221	o.m.	173727	295279	173360	10966	116	2126
3-2	P-C	o.t.	1572	30093872	246229	185707	3245	118	843
	P-E	175	o.m.	188101	199539	184887	3245	123	2211
	P-D	o.t.	1561	29472967	246995	164672	17261	106	826
	¬P	172	o.m.	162085	144809	161473	17261	108	1788
3-3	P-C	32	o.m.	43323	0	43323	43323	29	1347
	P-E	42	o.m.	63035	0	63035	63035	42	1513
	P-D	31	o.m.	44554	0	44554	44554	30	1421
	¬P	44	o.m.	64583	0	64583	64583	43	1463

Table 3.3: RayTracer results with no POR at all (¬P), purely POR using object escape analysis (P-E), purely dynamic POR (P-D) and the combined POR approach (P-C).

of dynamic POR (on its own and when combined with POR using object escape analysis) decreases the amount of stored states per megabyte by a half when compared against purely POR using object escape analysis. This is observable in table 3.3 as well. One might expect that the combined POR would use less memory instead of more because a smaller state space is explored. The memory savings of traversing a smaller state space are however cancelled out by the storage of summarised interleaving information (see 4) necessary for stateful dynamic POR.

Conclusive, POR in general enables a significant reduction of the state space. This reduction is as such, that it enables full verification of configurations which would be otherwise not fully verifiable without POR. We furthermore observed that POR using object escape analysis is the most effective POR technique. It reduces the state space significantly without much performance and memory overhead. Stateful dynamic POR can further reduce the state space, but the degree of reduction depends on the structure of the model. This reduction comes at the cost of increased performance and memory overhead. These observations are in line with that of [43], [18] and [66] and [53].

3.4 Filter-based Exception Handling

The first version of MMC missed exception handling. The lack of it was reflected during testing (see §3.5), where many tests failed because exceptions, like arithmetic overflows or illegal array accesses were left unhandled. Also, the Java Grande Benchmarks (see §1.4) relies on proper exception handling for correctness. For these reasons, we designed and implemented exception handling in MMC.

The CLI 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. We have implemented it fully in MMC. 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.

The remainder of this section shall introduce CLI's exception handling mechanism followed by a description of the implementation in MMC.

3.4.1 Structured Exception Handling

CLI specifies an exception handling mechanism called Structured Exception Handling (SEH) [45]. It is similar to Java's exception handling mechanism, but differs substantially on three points.

For one, contrary to Java, all exceptions in .NET are unchecked. At source level, methods do not need to catch all exceptions possibly raised

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

Figure 3.1: 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 3.2: Expressing filter-handlers in Java.

by its body. For this instead, the CLI traverses the callstack upon a raised exception in search for a method that has a matching handler. If one is not found, the thread in which the exception occurred stops.

Another significant difference from Java's exception handling mechanism, is that SEH allows user-defined exception handlers. In Java, the appropriate exception handler is looked up by matching the exception's type with the types defined by the catch-handlers. SEH supports in addition to Java's type-matching handler-lookup a filter-mechanism. A filter-handler is composed 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, SEH further traverses the callstack for other possible filter- or type-handlers. Filter-based and type-based handlers can be mixed. An example of this is shown in figure 3.4.1, which is a piece of VisualBasic.NET code. It is possible to express filter-handlers in Java using a catch-all, and have that catch all to do the actual exception handling. This approach is less elegant with the equivalent in .NET, as can be seen in figure 3.4.1.

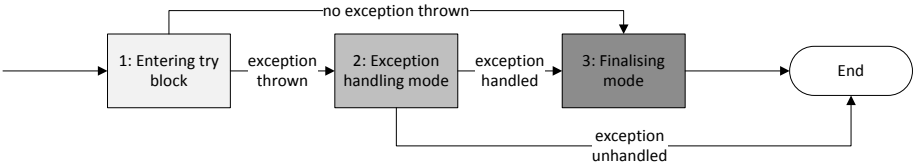


Figure 3.3: Global overview of structured exception handling.

A third point are fault-handlers. SEH supports it in addition to finally-handlers. Fault-handlers are similar to finally-handlers, but are different in that fault-handlers are only invoked when an exception was or will be handled.

3.4.2 Architecture

Figure 3.3 shows a global overview of exception handling. Three global phases are distinguished, namely the entrance to the try block, the exception handling mode and the finalising mode.

If an exception is thrown while the program counter is within the try block, exception handling (process 1 in figure 3.3) mode kicks in. Otherwise, it will eventually always leave the try block via a `leave` instruction. In both cases, finalising mode is entered.

Upon detection of an exceptional condition, the IsSourceException field of the current method state is set to true, indicating that the exceptional condition arose from that method. Next is the construction of the exception object, which is constructed as any object. Upon completion of construction, the program counter returns to the method state from where the exception was thrown. This method state is indicated by the IsSourceException field. The instruction `throw` is used to throw the reference to the exception object, thereby entering exception handling mode (process 2 in 3.3).

Figure 3.4 shows the flow diagram of the exception handling mode (process 2 in 3.3). The algorithm is shown in figure 10. The first step is to find a suitable exception handler. For that, the callstack is traversed from top to bottom to search for the first catch- or filter-handler within the scopes of the current program counters (see process 2.1 in 3.4 and lines 2 to 16 in algorithm 10). While traversing, it also purges the evaluation stacks of the traversed method states (line 3 of algorithm 10). If no suitable exception handler is found, the call stack of the active thread is cleared, indicating that the exception is unhandled (line 18 of algorithm 10).

If a matching catch-handler is found first in a method, then its program counter is set to that catch-handler. The stack of that method is purged and ThreadState::ExceptionReference is pushed upon that. This denotes that that method is ready to handle the exception. See lines 5 to 7 in algorithm 10. However, before handling the exception, all finally- and fault-handlers

Algorithm 10: HandlerLookup()

Data: Callstack of current thread C, ObjectReference e

```
1  retval ← null
2  foreach m ∈ C do
3      clear m's evaluation stack
4      eh ← m.NextFilterOrCatchHandler(e)
5      if eh is a catch handler then
6          m.PC ← eh.HandlerStart
7          push e on m's evaluation stack
8          retval ← FinallyOrFaultLookup(m)
9          break
10     else if eh is a filter handler then
11         m.PC ← eh.HandlerStart
12         push e on m's evaluation stack
13         retval ← eh.FilterStart
14         m' ← m
15         push m' on C
16         break
17 if retval = null then
18     clear C
19 return retval
```

Algorithm 11: FinallyOrFaultLookup(m)

Data: Callstack of current thread C

```
1  m ← peek of C
2  retval ← m.PC
3  while C is not empty ∧ m's evaluation stack is empty do
4      m ← popped method from C
5      eh ← m.NextFinallyOrFaultHandler()
6      if eh ≠ null then
7          retval ← eh.HandlerStart
8          break
9  return retval
```

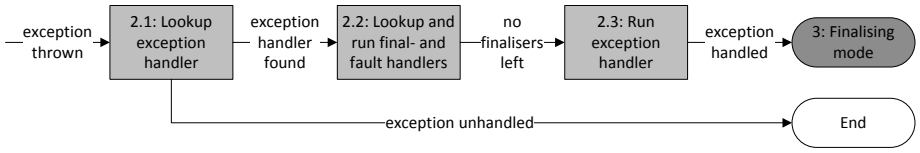


Figure 3.4: Flow diagram of exception handling mode.

occurring in method states above the one that will handle the exception have to be invoked first (see process 2.2 in 3.4 and line 8 in algorithm 10 and algorithm 11). This is done by traversing the call stack, do the necessary invocations, until the method state is reached that has the ObjectReference to the exception object on the evaluation stack (see algorithm 11).

If a filter is found first, things become more complicated. The filter block must be run first to determine whether it will handle the exception. It is possible that the filter occurs below the method from which the exception rose. It is not possible to pop off all methods above the one that has the filter, because the methods in between might contain finally- and fault-handlers that need to be invoked if (and only if) the filter returns true. We solved this by cloning the method state that contains the filter. The clone is pushed on the callstack. Its program counter is set the the beginning of the filter (see lines 10 to 16 in algorithm 10). If the filter returns true, the finally- and fault-handlers between the top method state and the one that contains the filter are invoked, similar to the invocation of finally- and fault-handlers upon a found catch-handler. If the filter returns false, the callstack is further traversed for lookup of other possible matching catch- or filter-handlers.

Finalising mode (process 3 in figure 3.3) is entered after the exception was handled by either a filter or catch-handler. It can also be entered after leaving the try-block. This can be seen by two entrance points in figure 3.5. A fault handler, if existing, is only invoked if it comes from exception handling mode. The latter case is distinguished by reading ThreadState::ExceptionReference, which will contains the ObjectReference to the exception object. This distinction is made before finalising mode is entered, because upon entering finalising mode, the ThreadState::ExceptionReference is set back to ObjectReference.Null.

Process 3.1, the "lookup final or fault handling" process, and the "lookup and run final handlers" (process 2.2) are similar. In fact, both call algorithm 11. The only difference is that in process 2.2, after invocation of a finally- or fault handler, subsequent finally- or fault-handlers are looked up and invoked, whereas in process 3.1 just ends after invocation. Here, the distinction is made by reading ThreadState::ExceptionReference. During process 2.2, ThreadState::ExceptionReference holds an ObjectReference to the exception object, whereas during process 3.1 and onwards, it holds ObjectReference.Null.

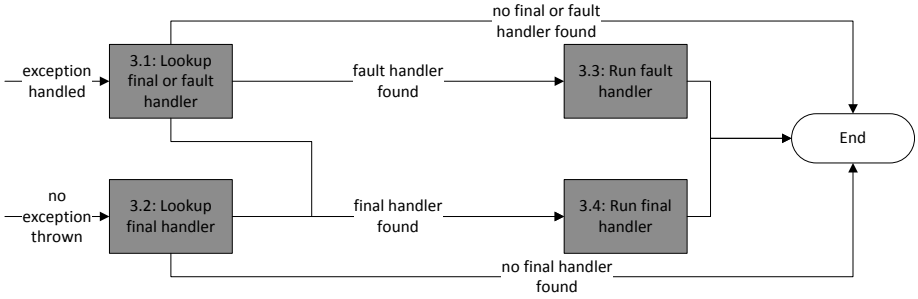


Figure 3.5: Flow diagram of finalising mode.

Between exception construction, throwing and filter-, catch-, final- and fault-invocation, other threads may interleave, causing other new states to be explored.

3.5 Testing Framework

Early in the second iteration of MMC's development, we set up a testing framework for two reasons:

- CLI conformance testing.
- Regression testing.

Conformance testing was the foremost objective. MMC was initially tested with C# programs only. We wished to support other .NET languages as well by conforming better to the CLI [56]. Conformance testing was also important to weed out bugs in MMC. Regression testing became more important after MMC evolved a lot and the chances of breaking something became bigger. The remainder of this section discusses the components of the testing framework and the results achieved with it.

Test suite A testing framework consists of two parts, (i) the test suite and (ii) the test driver. We started with looking at existing test suites. The CLI is a big specification and writing a test suite from scratch that covers it properly is time consuming. We considered JPF's and Mono's testsuites first. However, upon a closer look, we found out they were too unstructured, even lacking an unit test for each CIL bytecode instruction. The third test suite we investigated showed more promise. Microsoft's Rotor contains a highly structured and comprehensive test suite covering tests for the virtual machine, security mechanism, base class collection, JIT and platform independence. We took the virtual machine tests, which are called the Base Verification Tests (BVT). The BVT is composed of 328 unit tests that tests each CIL instruction for border equivalence classes and conditions.

Initial test results We designed a testdriver for running the BVT tests. Although the design was tailored for BVT tests, it should also work for other (self-made) test suites. The first testruns showed that MMC passed only 83 out of the 328 BVT tests. Quite a low score, but not surprising, as MMC was developed directly from the specification without such a structured testsuite.

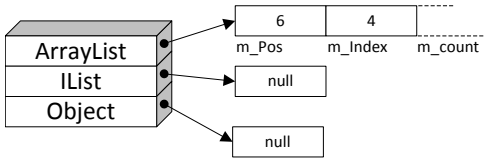
The tests failed for many reasons. A short list of reasons: bugs in wait-/notify semantics, bugs in default variable initialisation, bugs in static class initialisation in combination with the state collapser, bugs in handling virtual methods, bugs in accessing inherited fields, bugs with operations on 64-bits datatypes, bugs with operations on unsigned datatypes, bugs with casting, bugs with managed pointers, lack of overflow checking, lack of exception handling, lack of managed pointer arithmetic, lack of support for multidimensional arrays, lack of support for important internal calls, lack of support for reflection, lack of support for value types, etc. All of them were resolved except for the latter two. They are deferred as future work.

Unimplemented CIL instructions The parts where MMC lacked were mostly due to unimplemented CIL instructions. The first MMC release implemented 58 instructions. We added another 16 instructions upon it. These instructions varied from overflow checking, loading and operations on managed pointers, multidimensional arrays, exception handling, instructions on unsigned datatypes and casting. MMC in its current state still misses nine instructions, namely ARGLIST, CPBLK, INITBLK, LOCALLOC, CPOBJ, INITOBJ, MKREFANY, REFANYTYPE, REFANYVAL. Their implementation is deferred as future work.

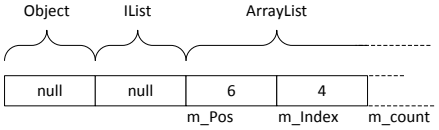
Initial tests runs with the MolDyn and RayTracer benchmarks unveiled the lack of many System.Math.* and System.Array.* internal calls. They were easily implemented in MMC, as the first MMC author anticipated on that.

Furthermore, the addition of exception handling as described in §3.4 improved conformance a lot. Exceptions play a big role in the BVT tests. The unit tests do not only test for border conditions, but also for semantics on violation of them. Exceptions are thrown in that case, which need to be caught, otherwise the test fails automatically. Now that exception handling was in place, many more tests could pass.

Another significant bug was that accesses to inherited fields did not work. MMC did not create space for field entities that were inherited from the class's supertypes. We considered two approaches to solve this. One, is to create a stack of field-arrays, where the stack represents the inheritance chain (see figure 3.6(a)). This approach would have overhauled too much of MMC, including the garbage collector and the state collapser. Therefore, we decided on a less invasive approach by creating one big array that hold all fields, including inherited ones. Upon initialisation of an object, the inheritance chain is traversed to calculate the necessary length. The



(a) By stack of field-arrays.



(b) By concatenated field-arrays.

Figure 3.6: Modelling inherited fields.

field-array is then initialised with that length. Upon access to a field, the corresponding offset in the field-array is calculated. See figure 3.6(b). This approach only changed the AllocatedObject constructor, `stfld` (store field) and `ldfld` (load field) instructions.

Final test results The testing framework was run repeatedly to fix the bugs and to fill in the missing parts. Currently, MMC passes 286 out of the 328 BVT tests. This is an improvement of 62% percent compared to the initial testrun. At this moment, most of the failures are due to the lack of pointer arithmetic on datatypes. We deliberately did not attempt to solve them. Code that contains pointer arithmetic is considered unsafe, and the CLI does not specify its semantics. Because its semantics are undefined, MMC cannot meaningfully implement them in a verifiable way. Besides that, Microsoft encourages the use of safe code over unsafe code. The current test results show that MMC covers lots of the semantics of safe code. This gives us more confidence that more .NET languages and more programs are verifiable with MMC.

3.6 Error Tracing

The first release of MMC prints out the current callstacks upon detection of an assertion violation or deadlock. This does not explain to the user how that error could have occurred. We extended MMC by providing a trace of CIL instructions. If the explorer detects an assertion violation or deadlock, the backtrack stack is traversed and a trace of the chosen thread-id's at the schedulingpoints is collected. Then the hashtable is purged of all states and the initial state is recreated. The trace is passed to the error tracer, and exploration is restarted from the initial state. The error tracer is simply a subclass of the explorer that only adds the thread-id from the trace to the

```
- thread: 0         0006 ret  on stack [3]
- thread: 0  _____________|
- thread: 0 |_ System.Void DataRace::Main(System.String[])
- thread: 0    0087 ldc.i4.6  on stack [3]
- thread: 0    0088 beq.s 0102 on stack [3, 6]
- thread: 0    0090 ldloc.0  on stack []
- thread: 0    0091 callvirt Cell::Get() on stack [Alloc(5)]
- thread: 0     |_ 0000 ldarg.0  on stack []
- thread: 0        0001 ldfld Cell::v on stack [Alloc(5)]
- thread: 0        0006 ret  on stack [3]
- thread: 0  _____________|
- thread: 0 |_ System.Void DataRace::Main(System.String[])
- thread: 0    0096 ldc.i4.s 12 on stack [3]
- thread: 0    0098 ceq  on stack [3, 12]
- thread: 0    0100 br.s 0103 on stack [0]
- thread: 0    0103 call Debug::Assert(Boolean) on stack [0]
10:26:33 [    Message] Assertion violation detected
```

Figure 3.7: Trace snippet to an assertion violation.

working set, not the full set of runnable threads. The error tracer also prints out the textual description of each executed instruction along with the the method's evaluation stack for each step. A sample trace snippet looks is shown in figure 3.7.

This error traces improves usability a lot, as it does not only provide a trace, thereby explaining the cause and effect relations towards the error, but also shows the nesting of methods as well, providing the user with an elegant overview of traced events.

3.7 Faster ChangingIntVector

During profiling MMC with the ANTS profiler[6], we observed a bottleneck with the ChangingIntVector. As described in [1], the ChangingIntVector is the datastructure used for representing collapsed states and maintaining the delta between successive states. The profiler showed that read accesses to the ChangingIntVector were responsible for a large stake of the total running time, roughly about 8.5% to 10% of the total running time. This is quite a lot for a rather simple datastructure. We gave it a thorough look and saw an opportunity to optimise.

The redesign involves an observation of an invariant. We observed that between successive states, if an index is written, it is only written once. With this invariant in mind, the ChangingIntVector was designed to contain one array holding all current values and a linked list of patches. Upon a

[6]From Red Gate Software Ltd: http://www.red-gate.com/

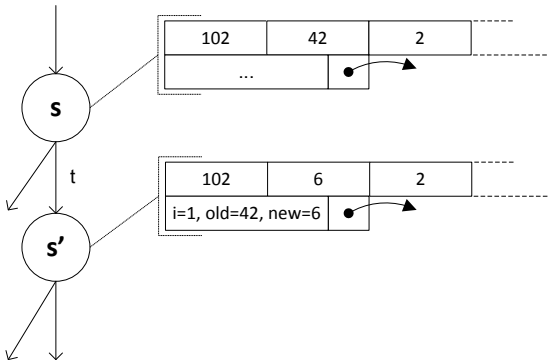


Figure 3.8: The linked list in the ChangingIntVector associated with state s' contains the changes between itself and state s.

read of the ChangingIntVector the array of current values was accessed and the desired value is returned. Upon a write, the new value was written and a patch was created that contains a triple of the index, the old value and the new value. The patch is added to the linked list of patches. See figure 3.8.

The state decollapser had to be slightly changed as well. Prior to the change, the state decollapser got a delta of the type ChangingIntVector. It was traversed to see whether any values had changed. Now, after overhaul, the state decollapser receives the linked list of patches as the delta. The changes are directly readable and do not have to be inferred from the ChangingIntVector.

This redesign increased performance significantly. The profiler backed this up by showing that the relative stake of accesses to the ChangingIntVector has been diminished to a near 0% percent stake. Such a percentage is what one expects more from a simple datastructure as the ChangingIntVector.

3.8 Resource Optimisation and Limitation

For extremely large models whose state space size is yet unknown, it is useful to control the time and memory available for exploration such that results can be retrieved from a partial exploration.

We added these limitations as runtime options to MMC. Two timers are used for that. The first timer triggers an event if the maximal time limit has been reached. Upon the event, the exploration algorithm jumps out the do-loop and stops exploration. The second timer is triggered every five seconds and is used to control and limit memory resources. At runtime, a maximal memory use limit in megabytes can be provided at which the exploration will stop. A second memory-related option is the ex post facto transition merger.

The ex post facto transition merger purges the hashtable from states that are, in retrospect, part of a mergable transition sequence. This is different than the a priori transition merger (see §3.3.2), as this transition merging technique occurs after the state is canonicalised, collapsed, stored and subsequently traversed. Such states can be safely purged, because upon a revisit of it, the exploration simply continues until an end state is reached or a non-mergable state is matched. As a state description in a software model checker is large, and the amount of states can be huge, this compaction technique saves memory at the cost for additional exploration time.

Initially, this compaction technique was implemented as such that it prevented the storage of states on a mergable transition sequence. We noticed that this increased exploration time quite much, especially on models whose paths have long sequences of mergeable transitions. Instead, we added a memory threshold parameter which can be provided as an argument upon running MMC. If this threshold is set, the ex post facto transition merger becomes enabled, and purging only occurs if that threshold is reached. By setting the memory threshold parameter close to the amount of available memory, the explorer holds as much states possible in memory, such that statefulness is as efficient as possible.

The effectiveness of this technique can be seen in table 3.2 and 3.3 by comparing the columns states and states stored. If the amount of states stored is less than the amount of states, then the ex post facto transition merger has purged states from the hashtable. If we look purely at the results of the combined POR approach, we see that twice the amount of states stored was explored of MolDyn configurations 2-2 and 3-1. Without the ex post facto transition merger, it would have run out memory twice at fast. The technique is particularly effective on a purely stateful dynamic POR exploration, where the amount of explored states is a multiple of what is stored. The reason for this effectiveness is that stateful dynamic POR lacks the a priori transition merger that is enabled with POR using object escape analysis. This also explains why the ex post facto transition merger is ineffective for verifications with only POR using object escape analysis enabled. Additionally, with stateful dynamic POR, it is well possible that the initial assumed singleton persistent set is the actual persistent set, and thus it is detected by the ex post facto transition merger as purgable when the memory threshold is reached.

3.9 Future Work

There are many promising techniques that further improve MMC. In this section, we shall highlight the ones we recommend to start with.

Partial order reduction The foremost technique that has substantially improved performance is POR. The current implemented POR techniques, object escape analysis and dynamic POR, reduce the state space a lot. Yet, there is still much room for improvement. Dependencies can be more fine-grained by distinguishing read-write dependencies from read-read independencies. Even better is to distinguish write-write independencies by looking up which value they write. These approaches still use information from the state space for the calculating dependencies. Instead of using the state space, one can also use the model itself. Such static analysis is more difficult on object-oriented software models due to their dynamically structured nature. Nevertheless this difficulty, [53] showed that their Indus analyser, which employs a collection of Java analysis techniques, could find many dependencies to drive POR. They showed that this POR on its own is on par with the POR approach using object escape analysis. A combination of the static POR, POR by object escape analysis and dynamic POR reduces the state space even more dramatically. A fourth POR technique which is worth implementing is sleep sets [23]. Sleep sets can reduce the number of transitions, thereby preventing calls to garbage collection, heap symmetry analysis and state collapsion for the revisited state. Case studies back this up: [66] combined dynamic POR with sleep sets and showed that this reduces the running time dramatically. As we see it, there is no reason why it could not be implemented along with the other three POR approaches.

Prevention of livelock As mentioned in §3.3.2, the merging of transitions may lead to a livelock. One approach to prevent this is by keeping track of the program counters on the backtrack stack and by keeping an incremental hash (see §6) of the dynamic- and static area. If the explorer jumps back to a program counter that is also on the stack, we know there is a loop. If the hash has not changes during that loop, we can assume the loop is a live lock. An another approach to solve the problem is by disabling transition merging, without disabling POR using object escape analysis. The advantage of transition merging would then also be disabled.

Program slicing We believe that program slicing will also improve performance a lot [43]. The idea with program slicing is to trim the CIL assembly such that only instructions remain that have an effect on the behaviour one is wishing to verify. Therefore by reducing the size of the model, the size of its state space is also reduced. Of course, only parts should be sliced out that have no effect on the checked assertion property or deadlock. Program slicing requires a static analysis, just as static POR as described above. Therefore, if a static analyser is written for either two of them, it is worth to design it for use of both techniques.

State compression The ex post facto transition merger technique (see §3.8) gave us another idea to further reduce memory. Instead of using full blown collapsed arrays, one can use partial arrays, where parts of the array are represented by a part from the parent state. This exploits the idea that only a part of the state is changed upon transition. Such a partial array could be composed of triples of ⟨ reference to array, offset, range ⟩. For values that have changed, a reference to a fresh state-unrelated array can be used. This works of course only well when a transition only incurs small changes. An issue is that at some point, referring to parts of arrays of parent states is not efficient, as too much patchwork may be needed to keep a partial state description complete. A simple solution to this might be to only use a full blown array for states at uneven depth, and use a partial array for states at even depth. This can in theory reduce the amount of used memory up to nearly 50%. This technique is also called difference compression [49].

Error tracing The current error tracer shows the sequence of CIL instructions from the initial state to the error state. This list of instructions is long and can be difficult to read. More interesting is to link the bytecode to the corresponding statements in the sourcecode, thus providing a more high-level trace. This is possible using the Debugging Interchange Format as defined in the fourth edition of the ECMA 335 specification. Additionally, it would be more helpful to show a sliced trace that only shows the statements that have a cause and effect relation with the detected error. This helps the user to understand the error faster.

Profiling Performance can also be improved by investigation of the bottlenecks. We have already optimised away the bottleneck with the ChangingIntVector, but there are more, less obvious, bottlenecks to optimise. Right now, the garbage collector tops at the profiler results. Any optimisation should start with that. We made a start by experimenting with the Memoised Garbage Collector. The reader is referred to §5 for more on this.

CLI conformance MMC can also be improved on CLI conformance. Even though conformance has improved much, the conformance is still incomplete. For instance, a few rarely used instructions have not been implemented yet. The support for value types is also lacking. If both become implemented, MMC would fully conform to .NET 1.0. For full conformance to .NET 2.0, support for generics is needed. Implementing generics into MMC is not a trivial task. Contrary to Java, generics are not syntactic sugar in .NET, but their notion is known and reasoned with at the level of the virtual machine. MMC therefore should do the same. We have not studied generics in depth to present implementation suggestions, but we are sure that is will overhaul much of MMC. However, prior to any conformance

improvement should be the creation of appropriate tests that can be used with the existing testing framework. This eases debugging and prevents possible regressions.

Testing other .NET languages We have only tested MMC with models expressed in C#. With the increased CLI conformance, it is likely that models expressed in other .NET languages are verifiable as well. More testing is however needed to claim compatibility with other languages.

3.10 Conclusions

A significant amount of time was spent on studying, designing, implementing and testing existing techniques for use in software model checking. The result is that MMC is much improved because of this. The explorer has been refactored, which made implementations of POR using object escape analysis and dynamic POR possible. Our experiments show that engineering these POR techniques in MMC were worth their time, as our experiments show that they improve performance a lot. The refactored explorer also eased the implementation of an error tracer. This gives the user better feedback on detected errors. Furthermore, on the performance front, we ran the profiler on MMC, and detected a bottleneck in a core datastructure, the ChangingIntVector. Read accesses to it topped the profiler results. We optimised the bottleneck away by redesigning it. Now accesses to the ChangingIntVector are at the end of the profiler results. Additionally, we implemented an ex post facto transition merger that reduces the memory used for large state spaces by purging states that are part of an mergeable transition sequence.

Also things were improved in the area of CLI conformance. A testing framework for MMC has been created that uses Microsoft's testsuite for conformance and regression testing. This made rigorous testing possible, and because of that, CLI conformance improved a lot. One part of the CLI that was challenging to implement is structured exception handling. The fined-grainedness of SEH introduces rather complicated issues, which we had to resolve. Our elegant approach towards SEH can be used as a reference for future software model checkers.

Even though MMC has improved greatly on all areas, there is still much room left for improvement. The future work in §3.9 provides some suggestions which we believe are the next logical step for further improving MMC.

Chapter 4

Collapsing Interleaving Information

This chapter presents a memory compression technique that collapses metadata, called the interleaving information, collected during a stateful dynamic POR search.

4.1 Related Work

This section introduces the stateful dynamic POR technique along with a brief description of state collapsion employed in software model checkers.

4.1.1 Stateful Dynamic Partial Order Reduction

The dynamic POR algorithm as described in §3.3.3 only works correctly with stateless exploration. The issue lies in the correct dynamic POR semantics upon a state revisit. A naive and incorrect stateful adaptation of dynamic POR would backtrack upon exploration of a revisited state. This is incorrect, because mutual dependencies between transitions in the state space below the revisited state and the current path to the revisited state would not be considered. This leads to over-aggressive reduction. Both [53] and [66] independently observed this, and proposed similar solutions. The idea is to mimic a stateless search upon a revisit by recalling all necessary information about the state space below the revisited state and inject the appropriate transitions in the working sets on the current DFS stack. In [53], each stored state is associated with a set that contains all transitions occurring after s. This set represents the interleaving information (II) after s, denoted as $II(s)$. Upon a revisit of s, mutual dependencies between transitions on the current DFS stack and $II(s)$ are 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 holds all transitions in the state space. The approach by [66] stores a more efficient representation of the interleaving information called the summarised interleaving information (SII). The SII only contains minimum-indexed transitions. This is discussed in detail in §4.2.1.

Both [53] and [66] provide empiric results of stateful DPOR, and show that stateful dynamic POR improves upon stateless dynamic POR in terms of speed. Both groups however noticed that while stateful DPOR reduces the state space even more, this increased reduction comes at the cost of increased memory. Both groups suggest that a form of memory compression on the (summarised) interleaving information to lower memory usage. However, they both left this issue open as future work. In this thesis, a compression technique in the form of collapse compression is proposed for compressing the SII's from a stateful dynamic POR exploration.

4.1.2 Structured State Collapsion

State collapsion has been extended to software model checking in JPF by [43]. In software model checking, states are dynamic in size and are built up structurally. The latter eases state collapsion. 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 collapsion has proved to be very effective in reducing memory usage. The level of reduction is dependent on the collapsion 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. Collapsion however incurs an overhead, but this overhead is overly compensated by the performance increase incurred by only matching references [43].

4.2 Implementation

This section describes the statefulness extension upon the dynamic POR implementation as described in §3.3.3. It first describes how to use the SII. The internal structure of SII is described afterwards, along with a scheme to collapse the SII to a CollapsedSII.

4.2.1 Statefulness Extension

In the stateless dynamic POR approach, ExpandSelectedSet is responsible for calculating dependencies with transitions in the current enabled set and the transitions on the DFS stack, and also injection of the appropriate transitions in the working sets. It does this based on the field entity and thread involved in the transition (of an enabled set). To mimic a stateless dynamic POR search by using SII, the SII of a state s should therefore contain pairs of $\langle field\ entity, thread \rangle$ that were involved in the transitions below s. Two hooks are needed for this, namely one into OnSeenState and one into Backtracked.

OnSeenState is triggered upon a state revisit. The extension (see the OnSeenState hook in algorithm 2) looks up the SII associated with the revisited state, requests the field entity-thread pairs and calls ExpandSelectedSet as if it were a stateless search.

Algorithm 12: OnSeenState(s)

```
1 foreach field entity-thread pair ⟨F, t⟩ in SII(s) do
2     ExpandSelectedSet(F, tid)
```

The next issue is how to collect the field entity-thread pairs such that it can be used by OnSeenState when it is necessary. To understand what is needed for a SII, one must understand the design flaw with a II.

The design flaw involves the notion of path-independency. Consider a path $\pi = s_0 \xrightarrow{t_0} \dots \xrightarrow{t_{i-1}} s_i \xrightarrow{t_i} \dots \xrightarrow{t_{n-1}} s_n$. The II 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 field entity F, then a stateful dynamic POR should find that 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 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 field entity. By leaving path-independent transitions out of the II, we get a Summarised II, called SII. The transitions in the SII are called minimum-indexed transitions.

Path-independency is different than independency, as the latter is an 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 would then be included in the SII. This reveals another issue with calculating the SII: The SII of a state s is not only constructed from one path from s, but for all paths from s. This introduces a problem: 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 in the SII's of successor states also hold for their parent state s. We can simply merge them to get the SII for s. The only pairs not valid 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. See algorithm 13.

Algorithm 13: Backtracked(s)

```
1 s' ← the state backtracked from
2 t ← transition explored from s to s'
3 foreach field entity-thread pair ⟨F, tid⟩ in SII(s') do
4     if F ≠ Field(t) ∧ F exists in s then
5         add ⟨F, tid⟩ to SII(s)
6 add ⟨Field(t), Thread(t)⟩ to SII(s)
```

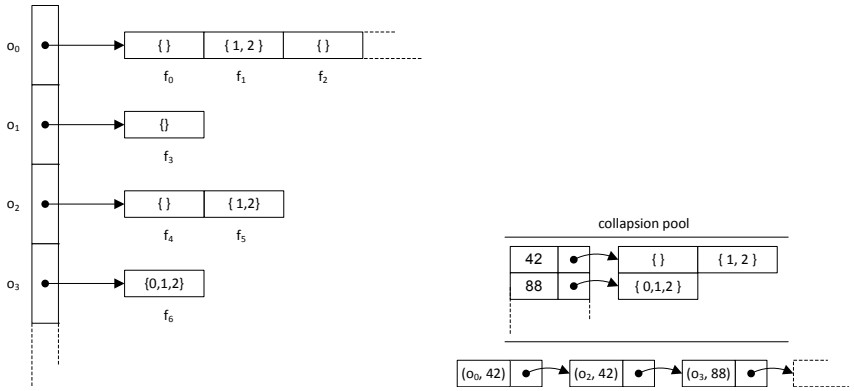
A SII for a state s is incomplete until all successor's SII are merged. We know this when the working set for s is empty, as then all transitions from s are explored and backtracked to.

4.2.2 SII organisation

So far, pairs of field identifier-thread were read and written to a SII. We have not described how the SII actually stores them. We have considered two approaches, of which one is implemented in MMC.

The first considered approach is the most straightforward approach: the SII is a set of field-identifier pairs. A fast implementation of a set is a hash set, which has $O(1)$ operations. Our initial implementation of stateful dynamic POR used this approach. Initial runs revealed that the performance decreased so much, that it was even slower than POR disabled. Initially, we found this strange, because of the favourable theoretical time-complexity of a hash set. Yet, upon a closer look using a profiler, we saw that the merging process (see algorithm 13) 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.

We devised a different approach that mimics the organisation of a state and hence its field entities. In MMC, a state consists of three parts: the dynamic area (i.e., the heap), the static area (for static classes) and the



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

(b) Collapsed representation of the left.

Figure 4.1: SII organisation and collapsion.

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. Whereas in a state the field entity holds the field value, the SII holds there a set of threads, namely those in the minimum-indexed transitions pairs. Thus, the SII follows the same hierarchical composition of a state. See figure 4.1(a).

Operations on such a hierarchical SII 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, they take space. Collapsion, as explained in the next subsection, is applied to solve that.

4.2.3 SII collapsion scheme

Collapsion compresses the SII by exploiting the notion that SII's of states do not change much between successive states. The collapsion scheme is rather obvious: we simply use the same state collapsion scheme for collapsing the SII. In MMC, this means that objects and classes are collapsed to a reference. See figure 4.1(b). We used two additional techniques to further reduce the size of the collapsed SII.

The first technique optimises by canonicalising the collapsed SII's. The canonicalisation process exploits the notion that interleaving information of two objects can be similar, even though their sizes differ. For example, in figure 4.1(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 unaccesed, and thus no threads are stored in them. In terms of interleaving information, o_0 is

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 4.1: MolDyn results with collapsing of SII's enabled (C) and disabled (¬C).

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.

The second technique exploits the observation that a collapsed SII is in fact a sparse array. A typical SII includes objects that are never accessed from s. 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 4.1(b), where the unaccessed object o_1 is not included in the collapsed SII.

An yet undiscussed issue of this collapsion scheme for SII's, is when to collapse and when to decollapse. Given a state s, the SII associated with state s should be collapsed 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, namely in OnSeenState.

4.3 Experimental Evaluation

We used the JGF benchmarks for evaluating the effectiveness of this collapsion scheme. The experimental setup is the same as those for POR (see §3.3). 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.

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 4.2: RayTracer results with collapsing of SII's enabled (C) and disabled (¬C).

The collapsion of SII's reduces the memory much. Table 4.1 shows that for 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 4.2), 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 0, the SII's were still on the DFS stack before the exploration ran out 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 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, 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 SII collapser improves performance. This is because the merging of collapsed SII's is faster as it is modelled as a sparse array. The full traversal of it is faster than a full blown array. Furthermore, collapsing of 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.

4.4 Future Work

There are 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.

Backtrack SII First, the current design of a backtrack SII (see figure 4.1(a)) fits the demands for which it is used. Most importantly, accesses to them are fast through indexing. They do consume quite a lot of memory, as it also reserves memory for field entities that are not accessed during exploration. For example a thread object has 40 fields, yet only five of them are accessed during exploration. The relative large size of the backtrack SII is not a problem as long as the state space is not too steep. Some configurations in the Java Grande Benchmarks have a depth of up to ten-thousands elements, and they were verifiable.

Yet, for models with problematic steep state spaces, other designs for a backtrack SII can be used. A hybrid approach is one way to reduce memory, while keeping the speed on par. A linkedlist-based SII is used for states whose interleaving information only contains n fields. If the threshold of n is crossed, the linkedlist-based SII is converted to an indexing SII. Combinations with other datastructures might also be effective. A sorted list, based for example on a binary tree, might also be effective. As a ordering scheme, one can order the fields $heap < static < lock$, then the location by their natural order, followed by the offset, also in their natural order. Even though the time-complexity of accesses to a sorted list is theoretically more expensive, it can be relatively cheap and fast when n is kept small.

When these solutions are not effective enough, then one might consider to store the backtrack SII in a collapsed form on the DFS stack. However, when they are needed again, they need to be decollapsed back. As this happens upon every backtrack to the state associated with the SII, this will certainly increase running time. Also, from an engineering point of view, one also has to consider to decanonicalise the decollapsed SII, as canonicalised objects in the SII may have a deviated size. We therefore think this is only necessary for verifying extremely steep state spaces.

Unstructured collapsing The above are merely suggestions that are have a likely chance of working out well. A more daring approach is to

let go of structured collapsing, and resort to unstructured collapsing, where for example blocks of n bytes are collapsed instead. Further study on the collapsion heuristics would be required in that case. We recommend those who pursuit this, to study the nature of SII, especially under which circumstances it is read, and when it is written, and experiment with datastructures that suit those circumstances.

4.5 Conclusions

Experiments with our proposed collapsion scheme enables MMC to store more states in the same amount of memory. The increased amount depends heavily on the size of the state space. As can be seen in the results of the RayTracer benchmark, the effect of SII collapsion is insignificant on small state spaces. For bigger state spaces, as seen in the results of the MolDyn benchmark, the memory is utilised more efficiently by allowing twice as much memory to be stored in the same amount of memory.

Even though our proposed collapsion scheme is already effective, it might be improved further, as can be read in the future work of §4.4.

Chapter 5

Memoised Garbage Collection

This chapter presents a new garbage collection algorithm, called Memoised Garbage Collection, for specific use in software model checkers.

5.1 Purpose of Garbage Collection

Garbage collection is usually, albeit not wholly, associated with virtual machines. It is the process of reclaiming memory allocations that will not be used in the future, thereby freeing up memory. Garbage collection is a rather expensive process, it usually requires the traversal of all memory allocation before it is decidable which allocations can be reclaimed. Within the context of software model checking, garbage collection is used for a slightly different purpose [43].

The scenario of a typical software model checker is as follows. Consider an object-oriented language like Java that disallows pointer arithmetic, like Java or C#. Objects used by a program are internally stored in an array. Yet, because pointer arithmetic's is disallowed, the index of an object (i.e., its address) has no semantic value. Objects can only be reached via referencing. When references between objects in an array are mapped, the resulting graph is an object graph (see §2.3). The shape of the object graph is of semantic value, because the references between objects are. Due to different interleavings of a program, a model checker can reach different states such that both have the same object graph shape, but the objects in question are permutated differently in the respective arrays. If states are matched by matching array-equivalence, the semantically equivalent heaps will be seen as different, thereby increase the state space unnecessary. Detection of semantically equivalent heaps is called *heap symmetry detection* [37, 43].

So far, two variants of heap symmetry reduction are known to be effective. The technique of [36] traverses the object 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 [43] maintains a canonicalised array, instead creating one when necessary. The latter is employed by MMC and details of it can be found in [1]. However, for both techniques, the array needs to be purged of garbage objects, i.e., objects that are not longer referenceable. Garbage objects may differ between states that have different paths leading to them, but are equivalent when canonicalised.

As for garbage collection in virtual machines, the time spent for garbage collection in software model checking is the most expensive algorithm in terms of its stake of the total running time (see §3.9) as it is run after each state visit. Its relative stake depends on the model and other enabled model checking techniques like POR. For example, for RayTracer configuration 2-1, we measured a 26% stake of the total running time with the Mark & Sweep garbage collection algorithm. In software model checking though, it is observable that changes between successive states are small. Hence, the changes to the object graph are also small. The garbage collection algorithm proposed in this thesis exploits this observation by remembering object graph shape information along with a state on the DFS stack, and propagates that information along with the changes due to the transition, to the successive state. Therefore, by reusing the information deduced from previous garbage collection analysis, time is saved on garbage collection.

5.2 Related Work

There are two fundamental different garbage collection techniques, namely Mark & Sweep and reference counting [64, 4].

The *Mark & Sweep* algorithm consists of two phases. The marking phase first marks all the roots of the object graph and then recursively marks all the children of the marked objects. The roots are allocations whose references occur in thread's callstacks and the static classes. The sweep phase traverses the array and deallocates all unmarked objects. The Mark & Sweep algorithm therefore traverses the complete object graph [64, 4].

The *reference counting approach* tracks the amount of references towards an object. This amount is updated when references are added, removed or updated. If the amount becomes zero, the object is ensured to be unreachable and therefore can be deallocated. There is a big disadvantage on the reference counting approach, namely that cyclic subgraphs unconnected to the fictive root cannot be detected with it. Consider the base case: an object with a self-reference. Such object, with a cycle to itself, has a reference count of at least one (unless the self-reference is removed). If that object becomes disconnected from the object graph, it becomes unreachable, but as it reference count has not reached 0, it is not detected as unconnected.

Reference counting algorithms used in practise (like virtual machines) therefore also employ a cycle detection algorithm, that is executed at particular intervals, and deallocates objects on disconnected cycles [64, 4].

For virtual machines, there are numerous variants on these two approaches. Stop-the-world variants only run the garbage collection algorithm when the program is paused. Afterwards, the heap is completely clean from garbage, and the program can be resumed. Another variant, namely precise garbage collection algorithms, identify all unreachable objects, whereas non-precise variants use assumptions to leave false-negatives on the heap. All algorithms need to ensure that they do not collect away false-positives [39]. In the context of software model checking, precise stop-the-world algorithms are used.

In this thesis, a completely different approach towards garbage collection is presented which is inspired by an *incremental shortest-path algorithm* originally intended for routers and context-free grammars. It is generalised for single-source directed graphs with positive weights in [52]. The basic idea is to track depths for each vertex. Upon changes to the graph, the tracked depths of the changed vertices become inconsistent, and their depths needs to be recalculated. It does this without traversing the whole graph. See the algorithm 14.

Algorithm 14: RamalingamReps()

Data: graph $G' = (V', v_0, E')$

```
1 while G' contains inconsistent vertices do
2     u ← inconsistent vertex with least key value
3     if rhs(u) < depth(u) then
4         depth(u) ← rhs(u)
5     else if depth(u) < rhs(u) then
6         depth(u) ← ∞
```

This algorithm is applicable as follows. Given a graph $G = (V, v_0, E)$ with a depth-labelling function *depth* that is consistent, i.e., the depths associated with the vertices from the fictive root v_0 are correct. Changes to the graph, like the removal or addition of edges or the removal or addition of vertices, lead to the successor graph $G' = (V', v_0, E')$. The depth-labelling *depth* is not changed accordingly, and thus for a subset vertices in V', the depth-labelling is inconsistent for graph G'. We call this the initial set of inconsistent vertices. For now we assume that we just know them, in §5.4 it is shown how these can be calculated. The algorithm is used to make the depth-labelling consistent by traversing inconsistent vertices from top to bottom. It first picks out an inconsistent vertex that has the lowest key. The key of vertex v is determined using $min(depth(v), rhs(v))$. The function rhs stands for right-hand-side function and calculates the depth of v based on a

parent from $Parent(v)$ that has the lowest depth-labelling. The rhs-value of an inconsistent vertex is always different than its depth-labelling. In case vertex v is underconsistent, i.e., $rhs(v) < depth(v)$, we make the v consistent by assigning $rhs(v)$ to $depth(v)$. In the case, it is certain that $rhs(v)$ is a consistent depth-labelling of v as the algorithm traverses the graph from top to bottom, and vertices between v and the fictive root v_0 are ensured to be consistent. In case vertex v is overconsistent, $depth(v) < rhs(v)$, we set the depth to infinity, to ensure it stays inconsistent until it becomes the vertex with the lowest key. Note that if a vertex is made consistent, it can cause child vertices to become inconsistent. They are made consistent too when their key is the lowest. The algorithm runs until no inconsistent vertex is left, i.e., the depth-labelling function $depth$ is consistent with G'. A proof of correctness is provided in [52].

From a more global view, the above algorithm determines a subgraph of consistent vertices from which neighbouring inconsistent vertices are made consistent and added to the subgraph of consistent vertices. Once an inconsistent vertex becomes consistent, it cannot become inconsistent in the same call of the algorithm. This is similar to Dijkstra's algorithm. The difference lies in the input. Whereas Dijkstra's algorithms starts from the fictive root, algorithm 14 starts with an inconsistent vertex closest to the root. It is made consistent and from that point, neighbouring inconsistent vertices are processed.

The foremost application of this incremental shortest path 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. This algorithm therefore reduces time.

Several other approaches to incremental computation of shortest paths have been proposed, an overview of them along with an experimental analysis is provided in [16]. They all have their unique distinctive features. Yet, the main reasons for basing the garbage collection algorithm in this paper on the above incremental shortest path algorithm is: (i), this algorithm is generally known to be correct, (ii), it is able to deal with deleted edges and (iii), it is relatively easy to implement as proof of concept. The use of other incremental shortest path algorithms for garbage collection is deferred as future work.

A final note on how our garbage collection algorithm compares against incremental garbage collection techniques [5]. Incremental garbage collection techniques are a class of imprecise garbage collection algorithms that use heuristics to determine which area of the heap contains collectable garbage objects. The work of an incremental garbage collector is then interleaved during the normal work to prevent observable pauses in program execution. Our algorithm is different as it is precise and as we view it, only applicable to model checking. It is memoised in the sense that is uses heap information remembered from the predecessor state to determine the garbage objects.

5.3 Algorithm

The Memoised Garbage Collection approach has algorithm 14 as its core. However, that algorithm is designed for any weighted direct graph in general, and not specifically for object graphs. To make it applicable for object graphs, we apply the following notions: the fictive root v_0 has always a depth of 0. The edges are always of weight 1, as the object graph is unweighted.

Given this specialisation, the object graph can be treated by algorithm 14 as any graph it would have as input. After that algorithm has run, objects that are reachable from the fictive root have a finite depth, while those that are unreachable have an infinite depth. The latter can be garbage collected.

Due to the dynamic nature of object oriented software, the algorithm should also deal with newly created objects. Consider a transition that creates a new object entity in the dynamic area. To ensure that it will be seen as reachable from the fictive root, the new object must be initialised with a depth of infinity. The object will then be seen as inconsistent upon the first next run of the Memoised Garbage Collector, and as such, it will be made consistent by assigning it with a consistent depth.

5.4 Implementation Details

Algorithm An implementation of the Memoised Garbage Collection algorithm has three issues to consider, namely (i) how are inconsistent vertices determined, (ii) how to find an inconsistent vertex with the least key and (iii) how to determine rhs-values of vertices. Algorithm 15 is an implementation of algorithm 14 for which these issues have been resolved.

Algorithm 15: MemoisedGC(s, s')

```
   Data: priority queue Q
 1 (V_s, E_s, v_0) ← the object graph associated with state s
 2 (V_s', E_s', v_0) ← the object graph associated with state s'
 3 foreach object o in V_s' do
 4     if Parents_s(o) ≠ Parents_s'(o) ∨ Parents_s'(o) is empty then
 5         insert o to Q with order key(o)
 6 while Q is not empty do
 7     u ← dequeue element from Q with smallest order
 8     if rhs(u) < depth(u) then
 9         depth(u) ← rhs(u)
10         CheckConsistency(Childs(u), s'))
11     else if depth(u) < rhs(u) then
12         depth(u) ← ∞
13         CheckConsistency(Childs(u) ∪ {u}, s')
```

Algorithm 16: CheckConsistency(U, s')

```
   Data: priority queue Q
1  foreach o ∈ U do
2      if rhs(o) ≠ depth(o) then
3          if o ∈ Q then
4              adjust o on Q with order key(o)
5          else
6              enqueue o to Q with order key(o)
7      else if o ∈ H then
8          remove o from Q
```

Let us tackle the first issue about how algorithm 15 and algorithm 16 combined solve the problem of determining inconsistent vertices. Initially, lines 3 to 5 initialise the algorithm by looking up vertices whose parents have changed. As the weights are always fixed to one, a vertex's depth is only changed if its parents have changed. Secondly, it is also possible that during one transition, an object is created, used and discarded again. Those objects have an empty parent set, and that is why they are also added as inconsistent vertices to Q. Lines 6 and 13 traverses Q in order by their key, and makes them consistent. When a vertex is made consistent, it could be possible that its children become inconsistent because of that. Therefore, all its children are traversed and checked for consistency by algorithm 16. Any inconsistent child is added to the priority queue Q so that it will be made consistent by algorithm 15

The second issue is determining the vertex with the least key. Inconsistent vertices added to Q are sorted by their key. Due to this order, the vertex with the least key can be extracted in constant time. Upon changes of an object's key due to changes of its parent, these changes are updated in the heap, as done in algorithm 15.

The third issue is about the determination of the rhs-value of an object. The rhs-value of an object is based on the depths of its parents. The parent with the least depth value is used for that. That value is then added with 1, i.e., the edge weight, to get the rhs-value. However, to make this possible, the set of parents of an object need to be maintained, which normally a software model checker does not.

Maintaining parent objects We approached this problem by looking how children of an object are maintained. For parents, this process is similar, but only the reverse edges are mapped out. Whenever a reference to a child changes, we also update the references to the parents. This happens in the following situations:

- Upon a `stfld`, if an object reference is stored into a field entity.
- Upon a `stelem`, 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. The referenced object then becomes a child of the fictive root.
- When an object reference is popped from the call stack. The referenced object is then removed as child of the fictive root.
- Upon the restore of a collapsed object, any object referenced from its fields become its childs.
- Upon the restore of a collapsed array, any object referenced from its elements become its parents.
- Upon the restore of a collapsed callstack, any object referenced from it becomes a child of the fictive root.

In order to maintain a correct set of parents of an object in the object graph, this set must be modelled as a bag[1]. It is possible that an object references another object multiple times by holding the same object reference is multiple fields. If one of these references is removed, then the parent-child relation still holds. The parent-child relation is discarded when all its references to the child object are removed.

Time complexity Let us first explain the used factors that is needed to express the time complexity of the Memoised GC. An object is affected if its depth has changed during one run of the algorithm. The extended size of an affected object is the amount of parents of that object. In [52], it is shown that the time-complexity of algorithm 15 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 costs to calculate the rhs-value. The reader is referred to [52] for a more thorough explanation.

5.5 Experimental Evaluation

We used the JGF benchmarks for evaluating the effectiveness of the Memoised Garbage Collection algorithm. The experimental setup is the same as for POR (see §3.3). All benchmarks where run with the Memoised Garbage Collector enabled, and another series were run with the Mark & Sweep garbage collector enabled.

Table 5.1 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

[1] A bag is also known as a counting set, a set that also keeps track how many times an element occurs in the set

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 5.1: 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 5.2: Raytracer results with the Memoised garbage collector (MGC) and the Mark & Sweep garbage collector (M&S).

MGC on these configuration is 9%. Table 5.2 shows that MGC is faster for all configurations except configuration 2-3. The increases 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 object graph. 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 object graph. 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.

5.6 Future Work

It is important to improve the garbage collector in a software model checker. Our profiler data of RayTracer configuration 2-1 shows that the Mark & Sweep algorithms takes 26% of the total running time. This is improved by the Memoised GC, which decreases the stake to 17%. Even though this is less, it is still significant. While developing and investigating the Memoised GC, we saw opportunities for further improving it. They are outlined below.

Priority queue Algorithm 15 is very amendable for improvements. For the priority queue, Q, we used the IntervalHeap [60] from the C5 .NET collection library. It has $O(logn)$ time-complexity for adding and removing

elements and $O(1)$ time-complexity for retrieving the vertex with the least key. Other datastructures may be more effective. It is important that operations for adding, removing and retrieve the vertex with least key are fast. A promising datastructure is the HOT queue [10]. It is designed for applications in which keys are monotone increasing, i.e., in one run of the algorithm, the keys added or updated are either equal or increasing. For such applications, the HOT queue has improved time complexities. A description of it, along with the elaborate time complexities are can be found in [10].

Right hand side value In our experiments we observed that the calculation of the rhs-value can be expensive, as it requires the traversal of all parents of the given child. This is of linear time complexity. This can be improved to constant time by using the improved algorithm by [52]. This improves the time-complexity, but it is more difficult to implement efficiently. For this reason, we have not implemented it ourselves, but put it as future work.

State decollapsion As shown in §5.5, MGC is not always improvement. The overhead incurred my the maintenance of parent sets is significant, and mostly caused on backtracking. Currently, on state decollapsion, changes to the heap are not accumulated, but committed per decollapsed object. Thus, successive decollapsion of the same object entity causes repeated calls made to the parent set maintainer. In the end, only the last change really matters. Thus, it might be effective to accumulate the changes, and calculate which changes are going to have effect, and only process those. Another possibility is to find a method to detect whether the state is going to change a lot and disable the parent set maintainer in that case. After the changes are over, a complete sweep over the heap can be made to recalculate all parent sets. An other approach is by storing parentlists associated with a state on the DFS stack, and restore them when the explorer backtracks to the state. This increases memory use, but saves a lot of parent lists maintenance necessary during state decollapsion.

No garbage We observed that lots of calls to the garbage collector do not result in the collection of garbage objects. It appears that most of the time, garbage collection is called on heaps that have no garbage. This happens especially when the callstack of a thread grows by successive method calls. If one develops a cheap method to determine such situations beforehand, it can be used to prevent a call garbage collector. The method does not have to be precise, it can have false-positives (where a positive indicates that the GC should be run), as long as the false-positive rate is low. Such a method will help the Memoised Garbage Collector a little, but might improve the performance of Mark & Sweep a lot.

Hybrid approaches We also observed that some calls to the garbage collector result in the collection of relative lots of garbage objects. This happens quite often when the exploration comes closer to the end state. Mark & Sweep is faster in such cases, and it would be beneficial to have the explorer dynamically switch to that algorithm from Memoised Garbage Collector. However, one has to develop a method that determines beforehand that likely (there can be false-positives) lots of objects are collected. Secondly, one has also to develop how the dynamic switch can be done efficiently.

Another way to combine the strengths of Mark & Sweep with Memoised GC is to partition the heap in two graphs. For one graph, Memoised GC is responsible. Mark & Sweep is responsible for the other. A partition criterion can be for example the number of fields in an object. Objects with lots of fields are the responsibility of the Memoised GC, the remaining objects are for Mark & Sweep GC.

Other incremental shortest path algorithms The incremental shortest path algorithm by [52], used as inspiration for Memoised Garbage Collection, has set off an active field of study on incremental shortest path calculation. Since its publication, refinements and specialisations have been proposed. We have not investigated the state of art in this area, because of the lack of time to study this, to us unfamiliar, field. It it well possible that improvements have been developed that are also applicable to Memoised Garbage Collection.

Incremental cycle detection with reference counting The improvements mentioned above are merely to improve the Memoised GC. Our study also gave us an idea for a more fundamental improvement. The Memoised GC uses the depth of a vertex as a property to determine connectedness. 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 to incrementally maintain the list of cycles in a heap. The reference garbage collector then 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 Memoised Garbage Collector is also applicable to other algorithms. For instance, [48] describes an incremental heap canonicalisation algorithm based on Iosif's canonicalisation algorithm [36]. 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 [54], such that thread symmetries can be detected incrementally.

5.7 Conclusions

The use of Memoised GC increases performance between 3% to 78%, depending on the model. The average performance increase for the MolDyn benchmark is 9% and for the RayTracer benchmark is 26%. This performance increase comes at an insignificant memory overhead. In few cases the Memoised GC performs worse, namely up to a two-fold performance decrease. This decreased performance is dependent on the model, and more particularly, its state space. A transition may trigger many changes to the object graph, which triggers the parent set maintainer, and eventually, triggers more objects as possible affected for traversal by the Memoised GC.

As the garbage collector is the most expensive algorithm in a software model checker, improving it can lead to much gain. Much of our time was spent investigating how to achieve that. While our proposed solution, the Memoised GC, works well for most configurations of the used Java Grande Benchmark models, it works only particularly well for configurations whose object graphs have depth labellings that change little during transitions. It is desired to devise a GC algorithm that works well for all models. Via our work on the Memoised GC, we generated ideas for future study (see future work in §5.6), which hopefully will eventually lead to a faster novel general purpose GC for software model checking.

Chapter 6

Incremental Hashing

The first application of incremental hashing scheme in a model checker is described in [47]. That incremental hashing scheme is only practicable for hashing stack and queues incrementally. We improved that hashing scheme by generalising it for hashing vector-based datastructure incrementally by using cyclic polynomials from [13]. Implementations in C and C# are provided as well, along with analysis followed from extensive benchmarking with Spin using the BEEM benchmark suite [51].

6.1 Use of Hashcodes in Model Checking

The hashtable is the cornerstone of stateful state space exploration. States encountered during exploration are stored in it. Upon exploration of each state, the hashtable is consulted to check whether that state has already been explored or not. Hashtables are also used for state collapsion, where shared components are mapped to a reference (usually an integer). Accesses to a hashtable are in amortised $O(1)$ time. Although this is a good worst-case time-complexity, the constant costs are high if a bad hash function is chosen.

The characteristics of a good hash function depends much on the anatomy of the hashtable. A hashtable constitutes of B indexed buckets. The element to be stored in a hashtable is called a key. Upon storing, the hashtable computes the hashcode of the key using a hash function, then maps that hashcode to an index of the hashtable and finally stores the key into the bucket associated with that index. In case another key is already mapped to that index, a collision occurred. The hashtable will need to perform collision-management, like closed address hashing or open address hashing, to resolve the collision [3].

Collision management is a time-expensive operation. Thus a good hash function should have a good uniformity, i.e., keys are spread along buckets as much as possible. A good hash function is also fast. In model checking,

states are usually represented as arrays. A good "traditional" hashcode of an array would at least require the traversal of the whole array.

But states can become large, and so do the arrays that represent them. The traversal of a large array becomes an expensive operation. That is why a considerable amount of running time of a model checker is spent on the computation of the hashcode. [47] observed that successive states do not change much, and if somehow the hashcode of the predecessor state can be reused to compute the hashcode of the successor state, a performance increase is gained. This is the underlying idea of incremental hashing as described in this thesis. Upon a small change of an array $k = v_1 \dots v_j \dots v_n$ resulting in the changed array $k' = v_1 \dots v'_j \dots v_n$, the hashcode of k' is calculated by using the hashcode of k and v_j, v'_j as input. This prevents traversal of all elements in array k'. This idea can also be extended to state collapsion, where shared components as stacks and object field vectors are also modelled as arrays and stored into hashtables.

6.2 Related Work

A well known hash function for hashing arrays is the rolling hash function [3]. Given a ring R, a radix $r \in R$ and a mapping function T that maps array elements to R, the rolling hash code for an array $a = a_0 \dots a_n$ is computed as follows:

$$H(a) = T(a_0) + rT(a_1) + \dots + r^n T(a_n) \tag{6.1}$$

$$= \sum_{i=0}^{n} r^i T(a_i) \tag{6.2}$$

A possible suitable ring R is $\mathbb{Z}/B$, where B is a prime and is also the amount of buckets in the hashtable. This specialisation of the rolling hash function is called hashing by prime integer division [13].

It is not difficult to see that the rolling hash function is prone to overflow, especially due to the power operations with the radix. Remedying overflow is costly. A recursive formulation of the rolling hashcode is less prone to overflow:

$$H(a_0) = T(a_0) \tag{6.3}$$

$$H(a_i) = rH(a_{i-1}) + T(a_i) \qquad 1 \leq i \leq n \tag{6.4}$$

Note that the radixes are reversely mapped to the array elements when compared to equation 6.2, and therefore hashcodes derived from the recursive formulation should not be matched against hashcodes derived from the non-recursive formulation.

[40] described an incremental recursive hash function for fast string pattern matching by using recursive hashing by prime integer division. Their method is as follows. Given a string $s = s_1 \dots s_n$ and a pattern $p = p_1 \dots p_m$:

1. Hashcode of pattern p is $H(p)$
2. Hashcode of the first m characters in s is $H(s_1 \dots s_m)$
3. Iterate for all $1 \leq i \leq n - m$:
   (a) If $H(p) = H(s_i \dots s_{i+m-1})$, then perform a character by character comparison on p and $s_i \dots s_{i+m-1}$
   (b) Else, calculate $H(s_{i+1} \dots s_{i+m})$ incrementally by:

$$H(s_{i+1} \dots s_{i+m}) = rH(s_i \dots s_{i+m-1}) + T(s_{i+m-1}) - r^{m-1}T(s_i)$$

The idea is to reuse the hashcode of the previous unmatched substring for the calculation of the shifted substring. In [13], this is generalised for matching of n-grams.

The rolling hash function is not only amenable for incremental recursive hashing, but also incremental linear hashing. The idea behind incremental linear hashing, is that the contribution of an array element is independent of the contributions of other array elements. In case of an array change, the influence of the old array element is known and thus can be removed, followed by adding the influence of the new array element [13]. In [47], this is expressed as follows. Consider an array $k = v_0 \dots v_i \dots v_n$ and its successor $k' = v_0 \dots v'_i \dots v_n$, then the hashcode of k' can be computed as follows:

$$H(k') = H(k) - r^i T(k_i) + r^i T(k'_i) \tag{6.5}$$

Depending on the ring chosen, the power operation with a large index i can easily lead to overflow. Thus using this hashing scheme for arbitrary modification of large arrays is impractical. For stacks and queues however, [47] describes a rewritten version of that formula for push and pop operations with the power operation removed. They tested it in their StEAM model checker, and got at least a speedup factor by 10 compared to non-incremental hashing. Note that this speedup was achieved with fixed-sized stacks of eight megabyte. It is logical to assume that the speedup factor will be much lower with with arbitrary sized stacks.

In this thesis, it is the goal to hash arbitrary sized arrays, not just stacks, without worrying about overflow issues, and still having a hash function that distributes uniformly. The inspiration for such a hash function came from [13]. That paper proposes the use of cyclic polynomials for the generalised n-gram matching algorithm. Operations over cyclic polynomials can be done using bitwise operations without overflowing. In this chapter, Cohen's recursive hashing approach using cyclic polynomials is combined with equation 6.5 to obtain a fast and overflowless linear recursive hashing algorithm for dynamically-sized arrays with arbitrary changes.

Finally, a section on hashing cannot leave hashing methods in cryptography unmentioned. Hashcodes in cryptography are used to determine authenticity of information. In [7], several incremental hashing algorithms

suited for cryptography are described. Though there are superficial similarities with the approach described in this thesis, there are fundamental differences. Hash functions in cryptography need to have the collision-free property, which contrary to one might assume, means that it should be computationally unfeasible to find two (or more) keys that hash to the same hashcode. The presented collision-free incremental hash functions suffer from overflow issues, which is less concerning in cryptography because hashcode sizes of 512 bits and even larger are more common, hence the increased uniformity. For a typical model checker, and for hashtables in general, hashcodes are restricted to 32 bits, and the collision-free property is of no concern. Speed on the other hand is, and as such, makes up the fundamental difference between the incremental hashing function there and the one presented here.

6.3 Incremental Hashing Function

This section presents the proof of the incremental property and the time-complexity of the incremental hash function. A few concepts, like polynomial rings, from algebra are used to express this proof. Readers unfamiliar with this may consult [13, Appendix A]. Implementors can skip this section and jump directly to §6.4.

Consider a Galois field (also known as a finite field) $R = GF(2)[x]/(x^w + 1)$, the ring consisting of polynomials in x whose coefficients are 0 or 1, reduced modulo the polynomial $x^w + 1$. Make sure that w matches the computer's word size, thus 32 for 32-bits words. The polynomials are represented by w-sized bitmasks by placing the coefficients of x^i at the i^{th} bit, creating an one-on-one correspondence between polynomials in R and the bitmasks.

As a radix, the polynomial $x^\delta \in R$ is chosen. By setting radix $r = x^\delta$, the following incremental hash function is derived from equation 6.5:

$$H(k') = H(k) + x^{\delta i}T(k_i) + x^{\delta i}T(k'_i) \tag{6.6}$$

The minus operation from equation 6.5 is replaced by an +, because addition and subtraction are the same in ring R. Now, consider an arbitrary member $q \in R$ with $q(x) = q_{w-1}x^{w-1} + q_{w-2}x^{w-2} + \ldots + q_0$. The multiplication of the x and $q(x)$ is the following:

$$xq(x) = q_{w-1}x^w + q_{w-2}x^{w-1} + \ldots + q_0x \tag{6.7}$$

$$= q_{w-2}x^{w-1} + q_{w-3}x^{w-2} + \ldots + q_0x + q_{w-1} \tag{6.8}$$

Equation 6.8 is equation 6.7 reduced to modulo x^w+1. The multiplication by polynomial x results to a left rotate of the coefficients in $q(x)$, hence the name cyclic polynomials. These multiplications by the radix are therefore easily

implemented by bitshifts on the bitmask. The addition can be implemented using an exclusive-or.

There is only one variable left unmentioned, namely δ. The choice of a δ for the radix x^δ was experimentally evaluated by [13]. No δ clearly stood out. For $\delta = 1$, the incremental hashing function worked well and they used is subsequently for their experiments. For this reason, this thesis shall now also take 1 for δ.

As described in [13], cyclic polynomials have one weakness. They have a cycle length of size w for which it computes the hashcode of zero. For example, if a key of size $2w$ starts with w elements followed by another identical sequence of w elements, then the hashcode for that key is zero. In practise, such keys are extremely rare in model checking, as their size must be exactly nw-sized, where $n \in \mathbb{N}$, and that its contents should be also w-cyclic as well.

The time-complexity of the incremental hash function is differently defined compared to traditional hash functions. A fast traditional hash function has a time-complexity in $O(N)$, where N is the array length. The incremental hash function hash function has a time-complexity of $O(1)$ for one change to the array. Theoretically, the incremental hash function is faster if the amount of changes between successive states is smaller than N. This is usually the case in model checking, where the amount of changes is usually 1 or 2 and almost never near N.

6.4 Implementation

A straightforward implementation of equation 6.6 would do, though there is an opportunity to save a few computing instructions by rewriting the formula using the laws of distribution and association. Considering $\delta = 1$, the rewrite is as follows:

$$H(k') = H(k) + x^i T(k_i) + x^i T(k'_i) \tag{6.9}$$

$$= H(k) + x^i (T(k_i) + T(k'_i)) \tag{6.10}$$

As described in the previous section, the + operator is implemented as a exclusive-or and the multiplication with the radix is implemented as a left rotate. Although nearly every programming language has an exclusive-or operator, this does not holds for the left rotate. In C# for example, a left rotate is simulated by using left and right bitshifting, like as follows:

```
public static int H(int k, int i, int ki, int ki_prime) {
    int diff = ki ^ ki_prime;
    return k ^ ((diff << i) | (diff >> (32 - i)));
}
```

Implementations for other programming languages are easily derivable from the above example.

6.5 Experimental Method

The incremental hashing function was originally developed for MMC. It was planned to conduct experiments with MMC and prove empirically that the incremental hashing function is better than non-incremental hash functions. However, initial tests with MMC showed that there was no measurable performance gain. We studied the results of these initial tests and the conclusions followed from it made us to decide to conduct experiments with Spin. Therefore, the experimental evaluation of it split into two parts, one part that describes the initial tests with MMC and one part that describes elaborate experiments with Spin.

6.5.1 Mono Model Checker

MMC was extended with incremental hashing by hashing the state vectors and collapsed components incrementally. Several models were then used for initial tests. The results were quite unexpected. Although they could be elaborated in much detail, it is only meaningful to mention that no performance gain was measured with any of the models.

The big question was: why? By running MMC through the ANTS profiler[1], the answer became evident. By comparing at the absolute time spent in the traditional hash function and the incremental hash function, it was clear that the latter was superior. The reason that this was not measurable without a profiler, is that hashing in MMC constitutes only a small fraction of the total running time. Any performance gain in hashing would not be visible in the total running time.

This experience forced us to learn an important lesson in optimisation. Profile first and optimise the bottlenecks. Hashing is not a bottleneck in MMC, so improving it serves no purpose. The ANTS profiler did however show that the theoretical improved time-complexity pays off. This gave us a glimpse that if the incremental hashing function would be implemented into an application in which traditional hashing consumes a relative large part of the total running time, a speedup might be gained.

6.5.2 Spin

Spin uses Jenkins's hash function by default, which is known for its speed and good uniformity [38]. It is a traditional hash function in the sense that calculates the hashcode by using the whole array as input. To measure

[1] A .NET code and memory profiler from Red Gate Software Ltd.

whether this hash function is a bottleneck in Spin, we ran a profile run on the petersonN model (distributed along with Spin) using gprof, the GNU profiler. The results were promising. The GNU profiler showed that Jenkins's hash constitutes 20% percent of the total running time. This indication was convincing enough for us to implement the incremental hash function in Spin.

Our implementation strategy was by adding a call to the incremental hash function before every change made to the state vector. The address of the changed element in the state vector is used as the index. This works for almost all Promela constructs except for unsigned integers and bits. Unsigned integers and bits are transformed by Spin into bitfields, which, according to any C standard, do not have addresses.

Initial testruns showed that the incremental hash function performed poorly compared to Jenkins's hash function. The total running time even increased with the incremental hash function. Collisions were to blame for this, as the incremental hash function distributed the keys very poorly. The source of the collisions lied in the entropy of changes between state vectors of successive states. Transitions are often of low entropy, like changing changing a variable from 0 to 1 or add 1 upon variable `i`. The incremental hash function recalculates the hash function upon such changes, but since the entropy is low, the resulting hash would not differ much as one desires for a good hash function. To improve entropy, every value to be hashed is first multiplied by Knuth's golden ratio of 2^{32}, which is 2654435761 [63]. This ensures that bits of the hashed values are better spread among the word (which is 32 bits) and therefore increasing entropy. Elaborate details on the resulting collision-rate is described later in this section. The incremental hash function used in Spin therefore looks as follows:

```
uint c_hash(uint k, uint i, uint ki, uint ki_prime) {
    uint diff = (2654435761*ki) ^ (2654435761*ki_prime);
    return k ^ ((diff << i) | (diff >> (32 - i)));
}
```

Instead of devising our own models for the experiments, it is more convincing to use a large existing benchmark set. The research group in Brno created the BEEM suite, consisting of 57 models that range from communication protocols, mutual exclusion algorithms, election algorithms, planning and scheduling solvers and puzzles [51]. The model are parameterised to yield different problem instances. The total amount of models is 298. 231 of them are in Promela. We used all the Promela models for our experiments. Each model was verified with four configurations, namely by every combination of the -O3 and -O0 compiler optimisation flags, Jenkins's hash function and the incremental hash function. The rationale behind considering compiler optimisation flags as factors shall become evident upon the

discussion of the results. All verifications were done on a Linux machine, equipped with 4 GB memory and a 1.86 GHz processor. The used compiler options are -DMEMLIM=3600 and -DSAFETY. The hashtable size was set to 2^{26} and the maximal DFS stack size was set to 10.000.000.

6.6 Results and Discussion

Table 6.1 shows the results of models that have a state space of one million and larger. Besides the number of states, it shows the size of the state vector of each model, along with the collision rates (in percentages of the number of states) of both Jenkins's and the incremental hash function. The verification time with Jenkins's hash function is taken as reference (100%). The verification times of the incremental hashing function are in percentages of the verification times of Jenkins's hash function. Note that the verification times were measured for both models compiled with -O0 and -O3, as we found out that this changes the performance gain of incremental hashing function. This difference shall be explained later on.

model	**sv (bytes)**	**states ($\cdot 10^6$)**	**Jen. (%)**	**inc. (%)**	**Jen. $\hat{=}$ 100% (sec)**	**inc. (%)**	**Jen. $\hat{=}$ 100% (sec)**	**inc. (%)**
			collrate		**time -O0**		**time -O3**	
telephony.6	36	239	16	17	390	87	197	88
peg_solitaire.6	52	234	12	19	598	90	283	94
telephony.5	44	214	13	14	353	89	174	92
telephony.8	44	208	14	14	337	90	168	92
phils.7	45	206	3	3	188	83	82	93
peg_solitaire.3	60	200	11	33	855	94	408	104
anderson.3	20	197	20	20	294	89	155	88
fischer.7	36	195	15	16	373	83	189	84
fischer.5	36	194	14	30	329	100	185	89
peg_solitaire.2	60	192	11	15	608	92	265	101
lann.7	52	186	11	14	529	91	264	90
bakery.8	36	182	14	34	269	85	127	91
anderson.5	28	170	11	22	247	89	123	93
lann.6	44	162	12	12	415	94	212	86
peterson.7	36	149	14	30	222	86	114	93
anderson.7	36	147	8	19	211	90	102	94
lann.8	52	144	10	14	413	92	206	92
anderson.8	36	141	13	55	242	93	118	100
phils.6	54	139	8	8	283	100	170	80
lamport_nonatomic.5	60	132	5	8	304	88	143	101
...								

(continues on next page)

model	sv (bytes)	states ($\cdot 10^6$)	Jen. (%)	inc. (%)	Jen. $\leqq_{100\%}$ (sec)	inc. (%)	Jen. $\leqq_{100\%}$ (sec)	inc. (%)
			collrate		time -O0		time -O3	
lann.5	68	132	8	17	425	93	214	84
at.5	28	125	10	14	205	89	101	96
peterson.5	28	124	7	14	172	87	89	87
phils.8	53	121	4	4	158	102	88	87
mcs.5	36	116	10	31	182	93	100	95
telephony.7	36	114	7	8	184	89	80	101
at.6	29	107	11	12	181	86	89	94
bakery.7	28	95	7	11	130	86	59	90
elevator_planning.2	46	93	6	55	127	83	60	95
blocks.4	37	93	3	3	108	91	48	97
loyd.3	29	89	4	4	108	94	49	90
production_cell.5	85	88	5	5	593	95	261	95
anderson.6	37	87	7	45	164	94	87	101
extinction.3	76	83	7	7	235	88	110	91
frogs.5	44	82	10	11	183	88	87	88
extinction.4	76	80	8	8	227	89	110	92
train-gate.5	60	76	7	21	236	89	101	99
train-gate.6	60	73	8	20	233	96	105	94
train-gate.4	60	73	7	20	228	89	98	99
firewire_link.3	76	72	7	13	294	94	140	91
production_cell.6	93	71	4	4	573	94	256	96
elevator.3	61	70	6	8	216	91	103	92
iprotocol.6	60	69	5	5	196	91	88	96
iprotocol.5	60	69	5	6	194	94	89	96
iprotocol.7	60	69	5	5	201	90	89	92
cambridge.5	77	68	6	7	220	90	97	93
krebs.4	36	67	4	4	123	86	54	101
at.7	38	64	7	8	116	88	59	88
telephony.4	36	64	4	7	100	88	50	92
driving_phils.3	84	62	7	12	131	83	53	87
train-gate.7	68	61	7	12	203	97	96	90
lamport_nonatomic.4	52	60	5	22	166	91	84	97
elevator.4	62	58	6	9	183	90	86	94
public_subscribe.5	68	58	7	9	194	95	95	90
train-gate.3	44	57	5	15	147	94	66	93
driving_phils.5	92	56	6	6	122	87	50	89
elevator2.3	36	55	3	3	75	89	34	97
elevator2.3_prop4	36	55	3	3	75	92	34	97
cambridge.7	92	55	4	4	192	86	84	91

...

(continues on next page)

model	sv (bytes)	states ($\cdot 10^6$)	Jen. (%) collrate	inc. (%) collrate	Jen. $\leqq_{100\%}$ (sec) time -O0	inc. (%) time -O0	Jen. $\leqq_{100\%}$ (sec) time -O3	inc. (%) time -O3
cambridge.6	84	53	5	5	173	87	75	91
firewire_link.6	116	51	5	10	266	90	112	100
lann.4	61	51	4	5	180	91	81	93
train-gate.2	44	50	5	15	128	94	58	92
brp.6	44	48	5	8	126	93	59	94
schedule_world.3	44	44	2	3	65	87	31	90
production_cell.4	60	42	5	5	165	91	70	97
elevator.5	88	39	3	4	132	92	66	88
bakery.6	28	38	3	11	52	87	24	92
frogs.4	44	36	4	3	76	89	35	95
fischer.6	36	33	3	3	64	83	32	86
peterson.6	28	33	3	3	49	86	26	91
lamport.8	28	31	3	11	40	93	21	99
driving_phils.4	84	30	2	46	64	85	27	93
bridge.2	28	27	4	4	66	96	32	93
sorter.4	44	27	4	5	67	89	34	95
at.4	29	25	2	2	41	88	20	95
bakery.5	28	25	2	8	34	87	16	92
lann.3	44	24	2	2	63	95	33	88
production_cell.3	69	24	2	2	128	94	57	92
sokoban.3	220	23	1	284	93	100	38	135
brp.5	44	20	2	3	54	90	24	97
lamport.7	28	19	2	12	26	93	14	99
krebs.3	36	16	1	1	26	96	13	92
hanoi.3	80	15	2	34	41	85	17	93
hanoi.4	80	15	2	8	39	91	18	91
brp.4	44	13	1	3	36	93	17	99
firewire_link.5	116	12	1	6	67	94	31	96
fischer.3	28	12	1	2	22	84	11	99
adding.6	20	12	1	1	15	85	8	92
msmie.4	60	11	1	1	36	103	19	104
iprotocol.4	52	8	0	1	24	93	12	94
adding.5	20	8	1	1	11	87	6	90
protocols.5	36	8	1	12	19	93	10	97
at.3	29	6	0	2	10	94	6	92
cambridge.4	61	6	1	1	19	90	9	96
peg_solitaire.4	36	5	0	0	12	92	6	101
adding.4	20	5	0	1	7	86	4	94
fischer.4	36	5	0	0	9	99	5	88

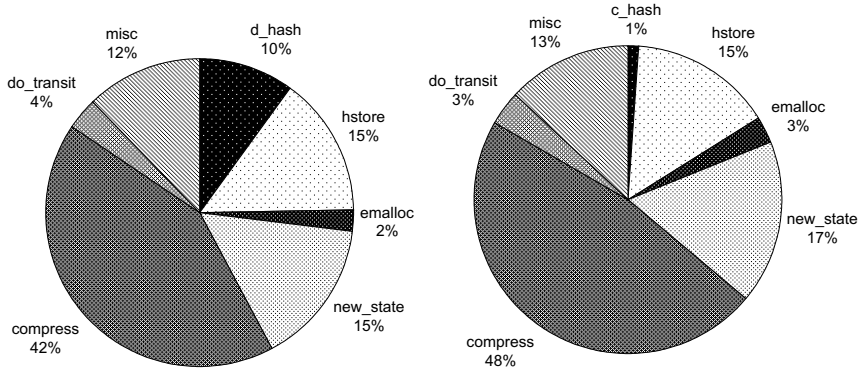
...

(continues on next page)

model	sv (bytes)	states ($\cdot 10^6$)	Jen. (%) collrate	inc. (%) collrate	Jen. $\leqq_{100\%}$ (sec) time -O0	inc. (%) time -O0	Jen. $\leqq_{100\%}$ (sec) time -O3	inc. (%) time -O3
reader_writer.3	68	4	0	0	47	102	20	99
leader_filters.5	36	4	0	2	7	94	4	98
phils.5	46	4	0	0	9	86	6	86
lamport.6	28	3	0	2	5	91	3	96
rushhour.4	125	3	0	0	11	84	6	92
telephony.3	36	3	0	2	6	85	3	91
phils.4	38	3	0	0	6	87	4	90
szymanski.3	28	3	0	0	5	89	3	90
protocols.4	36	3	0	10	8	94	4	98
adding.3	20	3	0	0	5	88	3	95
iprotocol.3	52	3	0	0	9	90	5	94
krebs.2	36	3	0	0	5	96	3	95
peterson.4	28	2	0	9	5	90	3	91
blocks.3	31	2	0	0	5	101	3	92
bopdp.3	44	2	0	0	6	97	3	96
sokoban.2	76	2	0	15	5	88	3	93
sorter.3	36	2	0	5	5	96	3	95
hanoi.2	65	2	0	4	5	90	3	95
rushhour.3	126	2	0	0	6	86	3	91
cambridge.2	61	1	0	2	6	94	3	97
cambridge.3	62	1	0	0	6	96	3	98
extinction.2	85	1	0	0	5	90	3	95
adding.2	20	1	0	0	3	91	2	96
pouring.2	90	1	0	0	15	97	7	98
brp.3	44	1	0	1	4	96	3	98
extinction.1	61	1	0	0	4	92	2	97
elevator2.2	38	1	0	0	3	93	2	96
Average	**52**	**62**	**5**	**12**	**144**	**91**	**69**	**94**

Table 6.1: BEEM benchmark results of Jenkins versus incremental hash function. The gain is the performance improvement of the incremental hash function over Jenkins's.

The key statistics of this table are as follows. With -O0, incremental hashing improves the exploration time up to 17%. The average improvement is 9%. With -O3, incremental hashing improves the exploration time up to 20%, with an average of 6% improvement. Thus, in general, incremental hashing is a substantial improvement upon Jenkins's.



(a) Jenkins. (b) Incremental hashing.

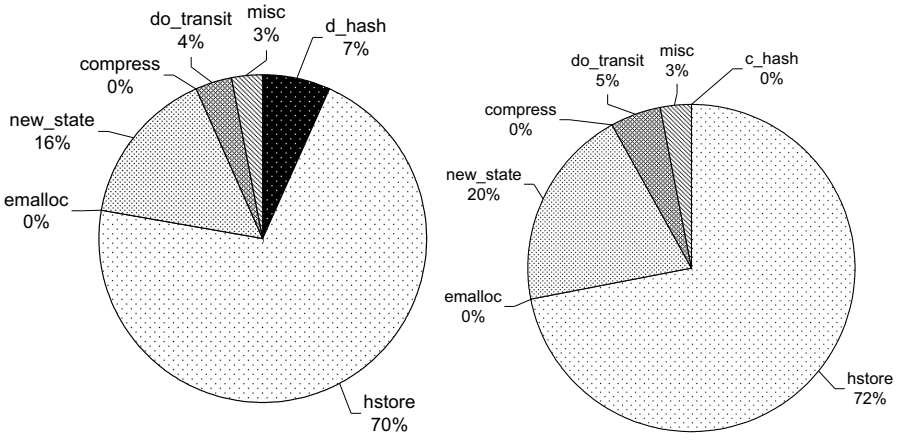
Figure 6.1: Percentual stakes of six most time-consuming functions for BEEM benchmarks compiled with -O0.

The outlier in the table is the sokoban.3 model. For this model, the performance decreases with incremental hashing by 35%. This extreme value is explained by looking at its collision rate. Its collision rate with incremental hashing is 284%. This percentage might look strange. However, Spin does not count how often a key collides with an index, but how much it collides with other keys in the chain[2]. So, it appears for sokoban.3, the incremental hash function hashes lots of different states to the same index. Besides this spike, the collision rate of Jenkins's is on average 5% whereas that of incremental hashing is on average 12%, thus the incremental hasher has a more worse uniformity, but not that much more worse.

To explain the difference in performance improvement between the models with -O0 and -O3, all configurations were ran again with GNU profiler enabled. For the explanation, we summarised the profiling data of the models with a state space of million states or more into pie-charts, see figure 6.1 and 6.2. The slices represent the relative stakes of the total running time of the six most time-consuming functions, namely the used hash function (c_hash is the incremental hashing function, d_hash is Jenkins's hash function), the hstore[3], compress[4], new_state[5] and do_transit[6]. Stakes of remaining functions are summed under misc.

A close look at the stakes of the hash function for models compiled with -O0 reveals that Jenkins's hash function has a considerable stake in the total running time. When the model is compiled with -O0, the average stake of Jenkins's hash is 11%. This drops to 1% when incremental hashing is

[2]Spin uses chaining for collision management
[3]hstore stores the current state in the hashtable
[4]compress removes bytes from the state vector which are known to never change
[5]new_state is the DFS routine
[6]do_transit performs one transition from the current state



(a) Jenkins. (b) Incremental hashing.

Figure 6.2: Percentual stakes of six most time-consuming functions for BEEM benchmarks compiled with -O3.

used. This shows that incremental hashing therefore eliminates hashing as a bottleneck. And more generalised, this proves that the incremental hash function is a substantial improvement upon traditional hashing.

Let us have a close look at the stakes of the hash function for models compiled with -O3 in figure 6.2. It also shows that here incremental hashing also improves upon traditional hashing by completely eliminating hashing as a bottleneck. If we compare those stakes of hashing with -O0, we see that Jenkins's hash is quite optimisible, as its average stake drops to 7%. This also reduces the maximal gain, and therefore also explaining the difference in the performance gain between -O0 and -O3 shown in table 6.1.

We hypothesised a correlation between the performance gain and the state vector size. We mapped these two factors out using the results from the BEEM benchmarks, but we did not found a correlation. There is a simple explanation for this: the BEEM suite does not cover models that have both large state spaces and large state vectors. The amount of invocations of Jenkins's hash is therefore small, causing to leave the hypothesised correlation disguised. This explanation is backed up by a small experiment we performed with a modified six-processes Szymanski model that was artificially extended by adding a global array which we varied in size. The results are shown in figure 6.3. The figure shows that for both -O0 and -O3 the performance gain grows with the state vector size. The growth declines after state vectors of 1646 bytes or larger. A stronger claim of the correlation should however be backed up by experiments with a variety of other models besides Szymanski's that have both large state spaces and large state vectors.

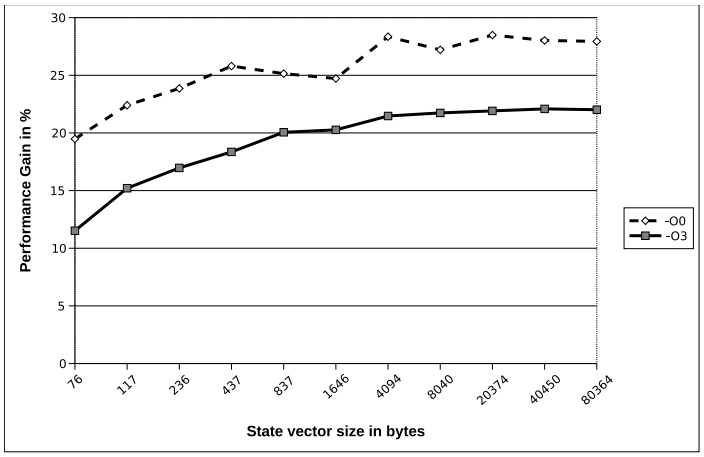


Figure 6.3: Relative performance gain of the incremental hash function with Szymanski' mutual exclusion model plotted against the state vector size.

A final remark on the differences between the results measured by -O0 and -O3. In general, we found the verification times with -O3 are on average 51% faster than those with -O3. This performance gain comes however at a cost, as the -O3 flag takes longer to compile. With -O3, we measured that all models compile within 21 seconds, with an average of 5 seconds. With -O0, all models compile within 2 seconds, with an average of 1 second. This difference in compilation time is in relative terms enormous, but nothing compared to tenths of minutes of verification the average BEEM model might take.

6.7 Future Work

The weaker uniformity of the general incremental hashing scheme leaves room open for future tweaking. For instance, a hashcode from the incremental hash function can be rehashed by an integer hash function, like Wang's integer hash [63], for increasing uniformity. One can also tweak the mapping of integers to ring R. The current mapping works by multiplying by Knuth's golden ratio, but other, less simplistic mappings are devisable. In any case, before one proceeds with such tweakings, we recommend one to analyse the outliers in collision rate in table 6.1 first, and further proceed from that point on.

For our experiments, we used 1 for δ (see equation 6.6), as this was experimentally decided to work out well by [13]. His work was however on hashing n-grams, which is a different application than model checking. It is interesting to see whether other values for δ might improve the uniformity of the incremental hashing function in the context of model checking.

Figures 6.1 and 6.2 show clearly that the hstore function has the biggest stake in the total running time. It is worth to profile this function in detail and optimise the bottlenecks in it. Performance might be also be gained by enhancing the collision management. Each key does not necessarily have to be hashed once, but can also be hashed twice or more by different hash functions. If a collision needs to be resolved, an indication of equality can be faster checked by comparing all hashcodes before a byte-to-byte comparison. This is of course expensive if traditional hashing schemes are used for calculating the hashcodes. Incremental hashing is much cheaper, and applying it for this context might strongly reduce the costs of collision management.

On the engineering front, the current implementation of the incremental hashing scheme in Spin does a call to c_hash at every change of the state vector. This is a quite invasive implementation, making future changes to Spin more error-prone. Also, this approach could not be applied for hashing bitfields, which are used to represent unsigned integers and bits in a Promela model. In a private communication, Holzmann suggested a difficult implementation approach. Instead of the more fine-grained incremental hashing scheme, he suggested to rehash a known and addressable portion of state vector (including the bitfields) upon a change. For example, like rehashing the whole local process structure if one local variable would be changed. This approach however does lead to a smaller performance gain, as the incremental hash function would have to work more.

Another architectural future-proof implementation would be piggybacking on an incremental state collapser. Such an incremental collapser does not exist in Spin, yet. Currently, if the model is compiled with -DCOLLAPSE, Spin recollapses the whole state upon each visit. However, the observation that only a small part of a state changes between successive states is very well applicable to collapsion. An implementation of this already exists in software model checkers like JPF and MMC, where only changed parts of the state are collapsed. For Spin, such an incremental collapser may improve performance by itself. The combination with incremental hashing may improve performance even further.

In this chapter, incremental hashing was purely applied to exhaustive state space search. It is also perfectly adaptable for approximative approaches, like bitstate hashing [30]. A 20-fold scheme is recommended by [65]. To give an indication of the possible gain, we profiled the same Szymanski's model with 20-fold bitstate hashing. The profiler measured a 32% stake of Jenkins's hash function with -O0 and a 15% with -O3. [42] even recommends a 30-fold scheme if hardware resources allows to. Using those recommended settings, we measured a 26% stake of Jenkins's hash function with -O0 and 15% with -O3. Those percentages are also the maximal performance gain with an incremental bitstate hashing scheme.

The general incremental hashing scheme can also be used for hash compaction [65]. With hash compaction, we even measured a higher possible

gain. The same Szymanski's model shows a 51% stake of Jenkins's hash function with -O0 and 40% stake with -O3. As with bitstate hashing, these even larger stakes indicate the possible performance gain with if the general incremental hashing scheme is adapted for hash compaction.

Lastly, the BEEM benchmark suite served their purpose for the greater part of our experiments. It was only lacking on one point, and that is where we wanted to unfold a correlation between the state vector size and the performance gain. The problem lies in the lack of models that have both large state spaces and large state vectors. Adaptations of models in the current suite, or a series of new models that do have those properties would be welcoming for increasing the usefulness of the BEEM benchmark suite even further.

6.8 Conclusions

Benchmarks with the BEEM suite showed that the use of the generalised incremental hashing scheme improves performance up to 20%. It does this by completely annihilating the relative stake of hashing from the total running time. The performance gain is reduced when the models are compiled with -O3, though still the resulting improvement is significant and noticeable. When it comes to uniformity, the generalised incremental hashing scheme is weaker than Jenkins's hash. The loss of performance because of that is however greatly compensated by the speed of the incremental hashing scheme. We hypothesised a correlation between the state vector and the performance increase, but we could not reveal such a correlation from the BEEM benchmark results. The suite lacks suitable models for measuring that. Instead, a small experiment with a six-process Szymanski model showed a strong indication that the incremental hashing scheme does correlate with the state vector size.

Not only did this research improves the performance of Spin, but more importantly, it opens various new possibilities for further research. A most promising direction for future study is the adaptation of incremental hashing to bitstate hashing and hash compaction as our provisional profilings indicate a major possible time reduction of up to 51%.

Chapter 7

Comparative Analysis

This chapter describes the results of benchmarking MMC against its main competitors, JPF and Bandera. These two are briefly described, followed by a rationale on the experimental method. The chapter concludes with an overview of the results, its discussion and suggestions for future work.

7.1 Competition

Java PathFinder was the first Java model checker and also pioneered the interpreted bytecode exceution verification approach. It has been used for verifying several real-world applications, most notable a prototype of the Mars Rover, Deep-Space 1 fault-protection and Shuttle ground control software. We retrieved JPF's version of 10 October 2007 from their SVN repository for our benchmarks. Features in this version not (yet) implemented in MMC are heuristic search and symbolic execution.

Bandera is also a Java model checker and also one of the earliest software model checkers. It distincts itself by its architecture: Bandera is a pipeline of tools combined to provide the functionality of a model checker. The version we used, version 1.04a from May 2006, is based on Bogor, which is a model checking framework. Features present in Bandera, but not in MMC, are a program slicer and a state abstracter.

The reason why we chose to set MMC against JPF and Bandera is that they 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 expresiveness.
- 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 [6], is different in that it interprets C code 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 [9] is similar, but uses a different abstraction approach.

The software model checker Zing [2] is also different in that is has its own object-oriented modelling language, the Zing language. This language is not, as far as we know, used for building software. The main reason is that the language lacks the expresiveness for that. And most importantly, it lacks inheritance.

XRT [26] is, like MMC, 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 is not taken with the benchmarks.

7.2 Benchmark Setup

All benchmarks were run 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. Verifications that exceeded 10 hours were automatically terminated. Also, verifications that exceeded 1.5 Gb use of memory were also automatically terminated. All three 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 verifications with MMC were run with the combined POR approach enabled, collapsion of SII's 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. For Bandera, POR using object escape analysis was also set to enabled.

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. Bandera cannot process bytecode that are targetted at a higher version.

Our initial benchmark runs showed that JPF explored a much smaller state space than MMC. We found that this was caused by a POR bug. JPF 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

config.	model checker	time (sec)	memory (Mb.)	states	revisits	max. depth	states/sec
2-1	MMC	434	1470	1481603	1062794	27885	5863
	JPF	411	1488	1377258	1319071	26686	6560
	Bandera	1772	o.m.	3045	0	99596	2
2-2	MMC	1447	o.m.	1928282	789872	195892	1878
	JPF	28618	o.m.	65681631	65270139	194710	4576
	Bandera	7841	o.m.	3962	0	210646	1
2-3	MMC	78	o.m.	245878	0	245878	3163
	JPF	508	o.m.	567965	0	567964	1118
	Bandera	7793	o.m.	10779	0	267473	1
3-1	MMC	913	o.m.	2725664	3021684	65706	6296
	JPF	31442	o.m.	65681631	128783286	63307	6185
	Bandera	2310	o.m.	11330	0	106769	5
3-2	MMC	91	o.m.	327597	0	327597	3591
	JPF	19610	o.m.	7864076	11395825	565215	982
	Bandera	12770	o.m.	16320	0	264632	1
3-3	MMC	153	o.m.	151782	0	151782	993
	JPF	529	o.m.	457719	0	457718	865
	Bandera	11466	o.m.	12501	0	267314	1

Table 7.1: Results of the MolDyn benchmark comparing MMC, JPF and Bandera.

thread-unshared. We fixed this bug in JPF and reran all experiments again. The results from it are shown in table 7.1 and table 7.2.

7.3 Results

We analysed the results from three points of view: speed, reduction of the state space and memory utilisation.

Speed Let us have a look at the performance in states per second. The results from the MolDyn benchmark are shown in table 7.1. The table shows that Bandera is no match against JPF and MMC. Its performance in states per seconds is between 1 and 5 states per second, which is in sharp constrast to the multiple thousands states per second achieved by JPF and MMC. MMC outperforms JPF in terms of states per second on configurations 2-3, 3-1, 3-2 and 3-3. JPF is faster on configurations 2-1 and 2-2. However note, that in table 3.2, we saw that the combined POR in MMC was significantly slower for these configurations than purely POR using object escape analysis. For configuration 2-1, MMC processed 21246 states per second and for configuration 2-2 it processed 6536 states per second when only POR using

config.	model checker	time (sec)	memory (Mb.)	states	revisits	max. depth	states/sec
	MMC	1	37	844	579	73	1231
2-1	JPF	14	1488	2205	1978	214	299
	Bandera	13051	o.m.	2589	2135	28292	0
	MMC	109	664	65923	53264	3173	1091
2-2	JPF	112	1488	67511	64180	3318	1176
	Bandera	o.t.	-	1472	140	40780	0
	MMC	o.t.	1151	79673	19899	19972	3
2-3	JPF	o.t.	1488	66352	33063	33286	3
	Bandera	o.t.	-	541	0	503544	0
	MMC	38	483	53631	71076	145	3278
3-1	JPF	172	1488	63207	117031	425	1048
	Bandera	o.t.	-	78981	143300	39234	6
	MMC	o.t.	1571	32520383	248967	3245	910
3-2	JPF	9944	1488	3590219	6938176	3528	1059
	Bandera	o.t.	-	60736	57395	51722	3
	MMC	23	o.m.	43330	0	43330	1890
3-3	JPF	379	o.m.	214594	0	214593	566
	Bandera	o.t.	-	197220	0	464478	5

Table 7.2: Results of the RayTracer benchmark comparing MMC, JPF and Bandera.

object escape analysis was enabled. This outperforms JPF by far. The same observation can be seen in the results with the RayTracer benchmark 7.2. MMC 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. Bandera is here the slowest too, as it maximal speed is measured at 6 states per second. We intended the investigation of this slow rate. Due to time-constraints, we did not managed to do that. The foremost reason is Bandera's complexity. It is built as a chain of tools, of which we would have to study all to understand how Bandera exactly works. We prioritised to investigate the results from JPF more thoroughly as there are more competetive with MMC and we also found JPF's source code easier to study.

State space reduction When it comes to state space reduction, MMC and JPF are both able to fully verify MolDyn configuration 2-1 and RayTracer configurations 2-1, 2-3 and 3-1. JPF is able to also fully verify RayTracer configuration 3-2. This observation was at first surprising, as we expected that MMC would be better at reducing the state space than JPF because it employs stateful dynamic POR. This is only true for the fully explorable RayTracer configurations. The increased reduction is however not big. There are three reasons for this. First, we saw in §3.3 that stateful dynamic POR is not much effective on the MolDyn benchmark, which explains the absence of increased reduction in MMC when compared to JPF. Second, MMC only performs object escape analysis via object reachability. It does not consider locking information to detect thread-unshared objects. JPF does, and this helps it to reduce the space space a lot more because both benchmarks makes extensive use of locking for rendez-vous barriers. The third reason is the semantic difference between Java and .NET. Especially the locking semantics of .NET cause an additional state explosion. 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. In general, there are too many factors involved with conducting experiments on two different platforms (.NET and Java) with two different tools (MMC and JPF) and two different representations of the model (the .NET version and Java version) in order to draw hard conclusions. That is why we have not stated concrete numbers on the relative performance difference between JPF and MMC.

Memory usage The last discussion point is about the memory usage. Bandera does not measure the peek memory use, and this is reflected by a minus in the memory column. Bandera does however indicate when it runs out of memory, hence the use of o.m. indicators. 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 likely that this 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 MMC by looking at the pace at which MMC runs out of memory. For all verifications that ran out of memory, MMC 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 §3.3).

7.4 Future Work

Future work, in this context, is mostly about investigating methods to perform comparative benchmarks that gives us better indication of relative differences in speed and memory efficiency.

First, we used barely modified source code of Java Grande Benchmarks benchmarks. While investigating the results, we detected that the current models are unoptimised for model checking. For example, the static field <benchmark class>.nthreads was accessed very often during exploration, but the accesses were only reads. This static field behaves as a final field. If it were marked final, MMC and JPF could make advantage of that in the POR and consider transitions that access that field as independent. We suggest to further investigate these models and optimise them for the purpose of verification.

Second, we observed that, to us known, semantic differences between .NET and Java lead to different state spaces for the same model. It is interesting to understand all differences, and how they affect the behaviour of the benchmarks and their verifiability with MMC and JPF. A start could be made by defining smaller models in both Java and C# and ensure their behavioural equivalence. They can be extended incrementally to eventually become models that reflect real-life situations. It is important to keep focus on the behavioural equivalence during every incremental extension.

Third and last point is about cross-fertilisation. For instance, when we saw that JPF considered accesses to final fields as independent, we considered this applicable to MMC as well, as accesses to readonly fields can also be considered independent. In general, techniques in JPF are usuable in MMC and vice versa. JPF could benefit from stateful dynamic POR, the SII collapser and the memoised garbage collector. MMC could benefit from the better object escape analysis by also considering locking information. For benefiting most from the cross-fertilisation, a study of source code of both

model checkers is recommended. Otherwise, details like the independency of accesses to final fields is likely to be overlooked.

7.5 Conclusions

A comparison between JPF and MMC requires a thorough knowledge of both Java and .NET platforms. Little semantic differences between them affect the size of the state space, and thus makes it is difficult to compare the degree of reduction achieved by JPF and MMC. Therefore, we suggest to investigate the semantics of Java and .NET within the context of software model checking. An approach for this is described in the future work (see §7.4).

Even though there were many factors involved in the benchmarking process, we can draw general conclusions. JPF and MMC are similar in terms of performance. In terms of raw states per seconds, MMC is generally faster. JPF is however better at reducing the state space because it also uses locking information for determining dependencies. JPF also utilised memory more efficiently than MMC. This difference in efficiency is caused by the use of stateful dynamic POR, which causes a significant memory overhead. Lastly, all results clearly show that Bandera is outperformed by JPF and MMC in terms of memory utilisation efficiency and speed. However, contrary to JPF and MMC, development of Bandera has been in hibernation since May 2006. This lack of active development may explain the performance gap between JPF and MMC.

Chapter 8

Conclusions

This chapter summarised this thesis by briefly discussing the conclusions from the previous chapters and the most important directions for future work.

8.1 Summary

Improving the mono model checker The principal author of MMC designed its architecture for extendability. We made good use of it. The general architecture is still the same. The only part of MMC we overhauled was the explorer algorithm. We did this to enable a clean implementation of the error tracer and partial order reduction. The error tracer increased the usability of MMC by generating a trace of CIL instructions towards the deadlock or assertion violation. The latter, partial order reduction, has improved MMC's performance a lot. We implemented two POR techniques, namely POR using object escape analysis and stateful dynamic POR. Experiments with .NET ports of the Java Grande Benchmarks show that these POR techniques enable MMC to verify models with much larger state spaces. This is in line with our research goal as stated in the introduction (see §1).

We also improved MMC's compliance with the CLI. A signification addition is that of exception handling. The CLI specifies Structured Exception Handling and is one of the most fine-grained exception handling mechanism for application platforms to date. Our implementation in MMC can be used as reference for future model checkers that implement a similar exception handling mechanism. We also created a test framework based on Microsoft's virtual machine test suite, called the Base Verification Tests. We used this test framework to detect numerous bugs and regressions that slipped in during development. Initially, MMC passed only 83 out of 328 BVT tests. Now, after fixing the bugs and regressions, MMC passes 286 out of 328 BVT tests. This increase coverage allowed us to verify more sophisticated .NET programs, like the .NET ports of the Java Grande Benchmarks.

Furthermore, this gives us more confidence that .NET programs compiled from other languages can also be verified by MMC, like VisualBasic.NET and Haskell.NET.

Some smaller, though significant changes are the ex post facto statement merger and the optimised ChangingIntVector. The first detects states that are on an atomic transition sequence. Such states can be safely purged from the hashtable. We added a runtime parameter to trigger the purging process at a given memory threshold, as it is beneficial to keep as many states of a state space in the hashtable. Furthermore, we optimised the ChangingIntVector, which is the core datastructure of a collapsed state. It contained a bottleneck, namely that it was optimised for lots write of accesses. In practise, it was read far more often. We then simply optimised it for this usage scenario.

Collapsing interleaving information The idea of collapsing interleaving information came from [66] and [53]. 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 a collapsion. 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. This too is in line with our research goal as stated in the introduction (see §1).

Memoised garbage collection The Memoised Garbage Collector uses information retrieved from changes between successive states to determine which objects can be garbage collected. When the changes are small, only a small part of the heap needs to be traversed. This technique therefore has a better time-complexity than Mark&Sweep, which is the dominant garbage collection in use by software model checkers. We ran experiments with the Java Grande Benchmarks to evaluate the effectiveness of Memoised garbage collection. The results show that the Memoised Garbage Collector is on average 9% faster on the MolDyn benchmark and 26% faster on the RayTracer benchmark. This enables the verification of models of larger state space in less time.

Incremental hashing Incremental hashing was first applied by [47]. Their incremental hashing approach was only pratical for hashing stack and queues incrementally. We improved this hashing scheme by generalising it for hashing vector-based datastructures incrementally. It was first implemented in MMC. Its positive effect did however not influence the total running time, as the profiler revealed that hashing constitutes an insignificant part of the total running time in MMC. To prove the effectiveness of our incremental

hashing function, we implemented it Spin. Hashing takes relatively more time in Spin than in MMC, and thus, the positive effect of it would become visible in the total running time. We evaluated it using the BEEM benchmarks, and measured a time reduction of up to 20%. The average time reduction is 9%.

Comparative analysis We evaluated MMC against JPF and Bandera using the Java Grande Benchmarks. Even though there are many factors involved in order to draw concrete conclusions, results indicated that MMC and JPF are on par in terms of performance. MMC is faster in terms of states per second, but JPF is better at reducing the state space because its object escape analysis algorithm also uses locking information. The results also indicate that MMC utilises memory relatively less efficiently than JPF. This causes by the memory overhead incurred by stateful dynamic POR. Bandera was outperformed by both MMC and JPF in all areas.

8.2 Future Work

All future work is described in sections §3.9, §4.4, §5.6, §6.7 and §7.4. We shall highlight the ones that we consider most promising to investigate.

In order to cope with bigger state spaces, we suggest to further investigate the use of POR. MMC currently employs POR using object escape analysis and stateful dynamic POR. The first can be improved by also considering locking information, as described in [18]. Stateful dynamic POR can be further improved by distinguishing more fine-grained independencies, like distinguishing read-write dependencies and read-read independencies. MMC can be further improved by using static POR pioneered by [53]. They use the Indus analyser to extract independencies. The same can also be applied for MMC if a similar analysis tool is developed. A fourth POR technique, sleep sets [23], 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.

In order to reduce the memory use, we suggest to improve state compression. We believe that a big memory reduction can be achieved by storing states using delta's. For example, the collapsed representation of states at an odd depth are stored normally, while states at an even state are stored as a delta of their parent state. A similar scheme is also applicable to further reduce the memory use of SII's.

During profiling, we detected that garbage collection has the biggest stake of the total running time in a software model checker. We made a start to improve this by developing the Memoised Garbage Collector. While

it has a significant time-reduction, we believe there are ways to reduce it even more. A promising direction is by investigating incremental cycle detection, and then combine that with a reference counting garbage collector.

While our incremental hashing proved not to be effective in a software model checker, we proved it to be effective in traditional model checking. This is based on experiments that used the incremental hashing scheme in an exhaustive verification. We believe that applying our incremental hashing scheme to approximative verification methods, like bitstate hashing and hash compaction, will significantly improve its performance. Initial profiling data show a possible time-reduction of up to a half.

8.3 Development Process

We would like to conclude this thesis with a note on developing model checkers. For this Master's project, we developed many improvements for MMC, added incremental hashing to Spin and fixed a POR bug in JPF's source code. During the course, we encountered many bugs, regression and anomalous results. We gradually developed tactics for hunting them more effectively, and therefore becoming more productive on the whole. To finalise this thesis, we described these tactics in appendix A. We hope that colleague model checker developers will benefit from it.

Bibliography

[1] Aan de Brugh, N. Software Model Checking for Mono. Master's thesis, University of Twente, 2006.

[2] Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., and Xie, Y. Zing: A Model Checker for Concurrent Software. In *CAV* (2004), R. Alur and D. Peled, Eds., vol. 3114 of *Lecture Notes in Computer Science*, Springer, pp. 484–487.

[3] Baase, S., and van Gelder, A. *Computer Algorithms*, Third ed. Addison-Wesley, 2000.

[4] Bacon, D. F., Cheng, P., and Rajan, V. T. A Unified Theory of Garbage Collection. In *OOPSLA* (2004), J. M. Vlissides and D. C. Schmidt, Eds., ACM, pp. 50–68.

[5] Baker, H. C., and Hewitt, C. The Incremental Garbage Collection of Processes. In *AIPL* (1977), ACM, pp. 55–59.

[6] Ball, T., and Rajamani, S. K. The SLAM project: Debugging System Software via Static Analysis. In *POPL* (2002), pp. 1–3.

[7] Bellare, M., and Micciancio, D. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost. In *EUROCRYPT* (1997), pp. 163–192.

[8] Ben-Ari, M. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, 1990.

[9] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. The Software Model Checker Blast: Applications to Software Engineering. *STTT 9*, 5-6 (2007), 505–525.

[10] Cherkassky, B. V., Goldberg, A. V., and Silverstein, C. Buckets, Heaps, Lists, and Monotone Priority Queues. In *SODA* (1997), pp. 83–92.

[11] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. NUSMV: A New Symbolic Model Verifier. In *CAV* (1999), N. Halbwachs and D. Peled, Eds., vol. 1633 of *Lecture Notes in Computer Science*, Springer, pp. 495–499.

[12] Clarke, E. M., Grumberg, O., Minea, M., and Peled, D. State Space Reduction Using Partial Order Techniques. *STTT 2*, 3 (1999), 279–287.

[13] Cohen, J. D. Recursive Hashing Functions for N-grams. *TOIS 15*, 3 (1997), 291–320.

[14] Corbett, J. C., Dwyer, M. B., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H. Bandera: Extracting Finite-State models from Java Source Code. In *ICSE* (2000), pp. 439–448.

[15] de Icaza, M. Mono: .NET framework. *Dr. Dobb's Journal of Software Tools 27*, 1 (2002), 21–24.

[16] Demetrescu, C., Emiliozzi, S., and Italiano, G. F. Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms. In *SODA* (2004), J. I. Munro, Ed., SIAM, pp. 369–378.

[17] Dwyer, M. B., Ed. *Model Checking Software, 8th International SPIN Workshop, Toronto, Canada, May 19-20, 2001, Proceedings* (2001), vol. 2057 of *Lecture Notes in Computer Science*, Springer.

[18] Dwyer, M. B., Hatcliff, J., Robby, and Ranganath, V. P. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. *Formal Methods in System Design 25*, 2-3 (2004), 199–240.

[19] Edmund M. Clarke, J., Grumberg, O., and Peled, D. A. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[20] Elmas, T., Qadeer, S., and Tasiran, S. Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets. In *FATES/RV* (2006), K. Havelund, M. Núñez, G. Rosu, and B. Wolff, Eds., vol. 4262 of *Lecture Notes in Computer Science*, Springer, pp. 193–208.

[21] Flanagan, C., and Godefroid, P. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL* (2005), J. Palsberg and M. Abadi, Eds., ACM, pp. 110–121.

[22] Garavel, H., Mateescu, R., and Smarandache, I. M. Parallel State Space Construction for Model-Checking. In Dwyer [17], pp. 217–234.

[23] Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[24] Godefroid, P., Ed. *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings* (2005), vol. 3639 of *Lecture Notes in Computer Science*, Springer.

[25] Graf, S., and Mounier, L., Eds. *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings* (2004), vol. 2989 of *Lecture Notes in Computer Science*, Springer.

[26] Grieskamp, W., Tillmann, N., and Schulte, W. XRT- Exploring Runtime for .NET Architecture and Applications. In *SoftMC* (2005), vol. 144, pp. 3–26.

[27] Groce, A., and Visser, W. Heuristic Model Checking for Java Programs. In *SPIN* (2002), D. Bosnacki and S. Leue, Eds., vol. 2318 of *Lecture Notes in Computer Science*, Springer, pp. 242–245.

[28] Havelund, K., Penix, J., and Visser, W., Eds. *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings* (2000), vol. 1885 of *Lecture Notes in Computer Science*, Springer.

[29] Holzmann, G. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings of Third Spin Workshop* (1997).

[30] Holzmann, G. An Analysis of Bitstate Hashing. *Formal Methods in System Design 13*, 3 (1998), 289–307.

[31] Holzmann, G., and Bosnacki, D. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering 33*, 10 (2007), 659–674.

[32] Holzmann, G., and Puri, A. A Minimized Automaton Representation of Reachable States. *STTT 2*, 3 (1999), 270–278.

[33] Holzmann, G. J. The Model Checker SPIN. *IEEE Transactions Software Engineering 23*, 5 (1997), 279–295.

[34] Holzmann, G. J. Logic Verification of ANSI-C Code with SPIN. In Havelund et al. [28], pp. 131–147.

[35] Huth, M. R. A., and Ryan, M. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, USA, 2000.

[36] Iosif, R. Exploiting Heap Symmetries in Explicit-State Model Checking of Software. In *ASE* (2001), IEEE Computer Society, pp. 254–261.

[37] Iosif, R., and Sisto, R. Using Garbage Collection in Model Checking. In Havelund et al. [28], pp. 20–33.

[38] Jenkins, R. J. Hash Functions for Hash Table Lookup. *Dr. Bobb's* (September 1997).

[39] Jones, R., and Lins, R. *Garbage collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc. New York, USA, 1996.

[40] Karp, R., and Rabin, M. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development 31*, 2 (1987), 249–260.

[41] Kastenberg, H., and Rensink, A. Model Checking Dynamic States in GROOVE. In *Model Checking Software (SPIN), Vienna, Austria* (Berlin, 2006), A. Valmari, Ed., vol. 3925 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 299–305.

[42] Kuntz, M., and Lampka, K. Probabilistic Methods in State Space Analysis. In *Validation of Stochastic Systems* (2004), C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, Eds., vol. 2925 of *Lecture Notes in Computer Science*, Springer, pp. 339–383.

[43] Lerda, F., and Visser, W. Addressing Dynamic Issues of Program Model Checking. In Dwyer [17], pp. 80–102.

[44] Leven, P., Mehler, T., and Edelkamp, S. Directed Error Detection in C++ with the Assembly-Level Model Checker StEAM. In Graf and Mounier [25], pp. 39–56.

[45] Lidin, S. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[46] McMillan, K. The SMV System. *Cadence Berkeley Labs* (1999).

[47] Mehler, T., and Edelkamp, S. Dynamic Incremental Hashing in Program Model Checking. *Electronic Notes in Theoretical Computer Science 149*, 2 (February 2006), 51–69.

[48] Musuvathi, M., and Dill, D. L. An Incremental Heap Canonicalization Algorithm. In Godefroid [24], pp. 28–42.

[49] Parreaux, B. Difference Compression in Spin. In *Proceedings from the 4th Spin Workshop* (1998).

[50] Pasareanu, C. S., and Visser, W. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In Graf and Mounier [25], pp. 164–181.

[51] Peláne k, R. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN* (2007), D. Bosnacki and S. Edelkamp, Eds., vol. 4595 of *Lecture Notes in Computer Science*, Springer, pp. 263–267.

[52] Ramalingam, G., and Reps, T. W. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal Algorithms 21*, 2 (1996), 267–305.

[53] Ranganath, V. P., Hatcliff, J., and Robby. Enabling Efficient Partial Order Reductions for Model Checking Object-Oriented Programs Using Static Calculation of Program Dependencies. Tech. rep., Department of Computing and Information Sciences, Kansas State University, 2007.

[54] Robby, Dwyer, M., Hatcliff, J., and Iosif, R. Space-Reduction Strategies for Model Checking Dynamic Software. *Electronic Notes in Theoretical Computer Science 89*, 3 (2003), 499–517.

[55] Robby, Dwyer, M. B., and Hatcliff, J. Bogor: A Flexible Framework for Creating Software Model Checkers. In *TAIC PART* (2006), P. McMinn, Ed., IEEE Computer Society, pp. 3–22.

[56] Ruys, T. C., and Aan de Brugh, N. H. M. MMC: the Mono Model Checker. *Proceedings of the Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation 190*, 1 (2007), 149–160.

[57] Smith, L. A., Bull, J. M., and Obdrzálek, J. A Parallel Java Grande Benchmark Suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing* (2001), p. 8.

[58] Stutz, D., Neward, T., and Shilling, G. *Shared Source CLI Essentials*. O'Reilly, 2003.

[59] Tretmans, J., Wijbrans, K., and Chaudron, M. R. V. Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System Revisiting Seven Myths of Formal Methods. *Formal Methods in System Design 19*, 2 (2001), 195–215.

[60] van Leeuwen, J., and Wood, D. Interval Heaps. *The Computer Journal 36*, 3 (1993), 209.

[61] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model Checking Programs. In *ASE* (2003), vol. 10, Kluwer Academic Publishers, pp. 203–232.

[62] Visser, W., and Mehlitz, P. C. Model Checking Programs with Java PathFinder. In Godefroid [24], p. 27.

[63] Wang, T. Integer Hash Function. `http://www.cris.com/~Ttwang/tech/inthash.htm`, 2007.

[64] Wilson, P. R. Uniprocessor Garbage Collection Techniques. In *IWMM* (1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*, Springer, pp. 1–42.

[65] Wolper, P., and Leroy, D. Reliable Hashing without Collision Detection. In *CAV* (1993), pp. 59–70.

[66] Yi, X., Wang, J., and Yang, X. Stateful Dynamic Partial-Order Reduction. In *ICFEM* (2006), Z. Liu and J. He, Eds., vol. 4260 of *Lecture Notes in Computer Science*, Springer, pp. 149–167.

Appendix A

Tactics for Debugging Model Checkers

A model checker is a very complex tool and building it can be a hardship. Bugs and regressions slip in easily and unexplained performance degradations are often observed. These issues can be hard to debug. A lot of time can be spent investigating the issue. This appendix describes tactics we applied for investigating such problems as efficient as possible.

A.1 Slice the Model

An issue is usually observed upon verification of a model. This model becomes important, because it usable as an aid to help understanding the cause of the issue. It is likely to loaded repeatedly into the model checker in order to see whether a fix resolves the issue, or when particular points in a model checker's execution path are interesting as introspection moments. Thus to reduce the time of these debugging round trips, it is crucial to reduce the verification time of that particular model.

With slicing, it is important that the sliced model still reveals the issue. Also, it is important to slice as much as as possible, because for debugging some issues, it might be helpful to generate and analyse its full state space.

A.2 Debugging Facilities

During this Master's project, Microsoft Visual Studio 2005 (VS2005) was used to develop MMC. The main reason for this, is that it provides powerful debugging facilities to the developer, which of course can only be of good use if the developer knows how to use them.

Breakpoints are important to pause the model checker at interesting statements. For example, if the issue is likely to come from the garbage collector, then put a breakpoint before the garbage collector is called. In

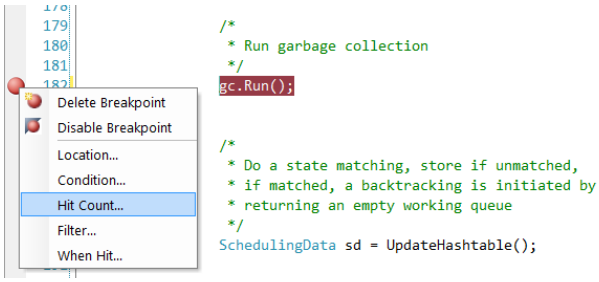


Figure A.1: Setting a hitcounter breakpoint before the garbage collector is called.

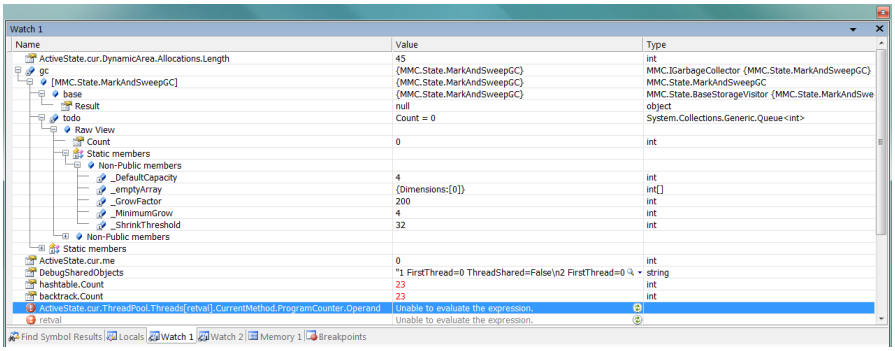


Figure A.2: Watch window upon a breakpoint.

VS2005, breakpoints can also be defined more fine-grained using breaking conditions (like, if it is suspected that the issue only arises by thread 0) or hitcounters (like, if it is suspected that an issue only occurs after n state collapsions). See figure A.1.

When the VS2005 debugging pauses at a breakpoint, introspection becomes possible. The mouse can be hovered over variables to see what values it hold as that moment. Also the variables window can be opened to see the variables and their values of the current method. The callstack window can be opened to introspect a different method state in the callstack. However, the most powerful tool for introspection is the watch window. Expressions can be put here for evaluation, like getting a current state as a string or getting the DFS stack as a string. See figure A.2.

When the debugger pauses at a breakpoint, it is possible to (re)write the code on the fly. The new code is then executed from that point. This is useful when a new feature has to be implemented but only it stub methods have been defined, or when an issue can be resolved on the fly.

Last, a disassembler might also be helpful to see how the compiler compiled the model to bytecode. VS2005 does contain a disassembler, but we have found the tool ildasm.exe more helpful, because it organises the disassembled code to its namespace-class structure.

A.3 Tactics

From our experience, most issues are either a thrown exception from the virtual machine MMC is running in (like a NullPointerException), a regression (like a state collapser bug slipped in during the implementation of the garbage collector) or a performance degradation (like when MMC performs unexpectedly slow on a particular feature). In all cases, to understand the cause of an issue, we need to answer *when* the issue occurs, *why* it occurs and *how*. The latter usually explains the why.

When For exceptions, understanding when they occur is easy. Just run a debugging instance in VS2005 with the sliced model and wait until the exception is thrown. VS2005 will automatically open up the source file and highlight the statement from which the exception is thrown.

For regressions, understanding when an issue happens is usually the most time-consuming part. We created hunches to pinpoint the when. Then this hunch was converted to a breakpoint. For example, if we believed that the cause was from the garbage collector, we put a breakpoint at the statement from which it is called. The breakpoint might be refined with conditions or hitcounts. We then gathered information using introspection (see the paragraph on Why) and used that to refine the conditions and/or hitcounts.

A more advanced tactic to understand the when is using purely hitcount-based tactics. For example, when something was broken in the state decollapser, and this was caused by the garbage collector, we used two breakpoints. The first breakpoint was then set at the state collapser. Additional break conditions were added to the breakpoint if necessary. The second breakpoint was set at the garbage collector. Its hitcounter was set to a big amount (one that is so big that it will not be hit before the first breakpoint is reached). Then we ran a debugging instance and waited until the first breakpoint was reached. When it was reached, we looked at the hitcounter of the second breakpoint and refined it to break at the current hitcount value. The debugger was then rerun until that hitcounter was reached. Both breakpoints were refined and the debuggers rerun until the when was understood. The general idea behind this is to use multiple breakpoints based on conditions and the amount of hits to pinpoint when an issue occurs.

Another tactic we used is an approach that we call dual-step debugging. This approach is particularly effective when a particular feature (like stateful dynamic POR) reveals an issue while another feature (like no POR) does not. We opened up two instances of VS2005 for this, loaded up the same sliced model, but in one instance, we ran the debugger with the feature enabled, and in the other debugging instance, we ran it with the feature disabled. Breakpoints were set at the same places. Then, the situation was assessed using introspection. If a breakpoint was refined, or a debugging step was made, it was done in both. At some point, the other instance

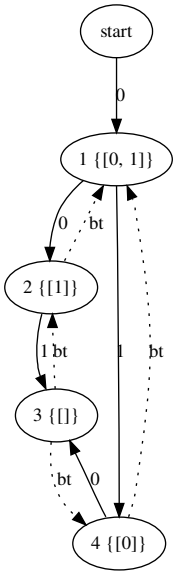


Figure A.3: An example state space visualised by GraphViz's dot.

would do something different from the other, and it is that moment that designates the when.

Another tactic is by generating the state space and inspect it manually. This is especially useful to investigate POR bugs. If MMC is run with the option -d <file> it will generate a GraphViz-parsable graph to <file>. Nodes in the graph represent states and each node is labelled by an integer, its identifier. The undashed edges represent transitions and their labels represents the thread doing the transition. An example is shown in figure A.3. For example, if an issue occurs at the creation of a state with, for example identifier 3, then we set a breakpoint in the explorer and have it break when a state with id 3 is created. The same approach can be used when the issue is suspected to occur from a revisited state.

Why When a breakpoint is hit, the model checker is paused and this moment can be used for introspection, the act of getting information of the model checker's current state. The most powerful means for this is the watch window. Using the watch window, we can read variables, look at an object's structure of inner objects and their values and call methods. The latter is extremely powerful. MMC contains many methods that return a string, which can provide useful information about the object on which it was called. Example methods that are almost always useful to call in a watch window:

- ActiveState.cur.ToString, shows a text-formatted version of the current state.
- ActiveState.cur.me, the identifier of the currently active thread.
- Explorer.DebugLastAccessed, shows a trace of fields accessed of the current path.
- Explorer.DebugWorkingSets, shows the working sets on the DFS stacks.
- FastHashtable.CalculateDistribution, shows the distribution of elements in the hashtable.

It is also possible to create other methods yourself, showing the information you need, and have them called in the watch window.

How Sometimes knowing when and why is not enough. An issue can have occurred due to cause and effects over time, and that needs to be understood before the whole cause of an issue is understood. The main approach for how is using dual-step debugging and/or repeated refinement of breakpoints. For example, if the issue is caused by the POR (because for example, it calculates the wrong persistent set), set a breakpoint at the method that determines the persistent set, and use debugging steps (like step over, step in) to determine what happens step by step.

Also, in case of understanding POR bugs, we created a -A option to MMC, which will cause it to explore only one trace to an end state, and print the intermediate thread-unsafe instructions. This is helpful to understand the points at which POR reduces the enabled set to a persistent set.

A.4 Profiling

We extensively used the ANTS profiler to profile performance bottlenecks in MMC. ANTS has two modes, the fast mode and the detailed mode. We used the fast mode to quickly pinpoint the methods that bottlenecked. In this mode, ANTS runs the profiled application (in this case the model checker) with JIT optimisations enabled. The information it provides is not precise, because the JIT compiler inlines lots of methods as an optimisation. If we needed to understand why a particular method bottlenecked, we used detailed mode with a filter on that method. ANTS then reports how much time each statement consumes in that method. Note that in detailed mode, the JIT compiler is disabled, and therefore it takes more time to profile.

In cases we ran the profiler on MMC with a big model, we took regular intermediate snapshots that provided us the current profiling statistics so far. Usually, these snapshots were sufficient to understand the bottleneck.

A.5 Conclusions

In general, understanding when an issue happens, why it happens and how it happens, involves tools and their proper use. In our case, we extensively used Microsoft Visual Studio 2005 and the ANTS profiler. These tactics are based on their features. Yet, we were also limited by them. Dual-step debugging for instance is great, but we had to do the steps manually. A tool that would perform dual-steps automatically until a discrepancy occurs would be enormously helpful. Also, the means for understanding the how of an issue are also limited. Tools that help us to capture a program's behaviour over time would be helpful here. Therefore, the tactics described in this chapter should be seen as a starting point for searching (or developing) new tools that help model checker developers to be more productive.