Software Model Checking for Mono

Niels Aan de Brugh

August 28, 2006

Abstract

Software has become larger and more complex than the human mind can grasp. This results in software that can contain errors, even when the software is thoroughly tested. Automated validation techniques can be a helping hand in the process of eliminating these errors. Specifically, model checking techniques can be used to systematically "test" all possible executions of the software, finding possible errors automatically. The development of tools to facilitate this can be considered a new trend in research on validation techniques. Parallel to this trend, a software development platform called .NET was released by Microsoft. This platform runs applications written in many different programming languages. Mono is a free and open implementation of .NET. This project is an attempt to combine both the error-hunting capabilities of software model checking and the versatility of the Mono platform. In the course of more than a year we have written our own model checking virtual machine, capable of checking applications in one of the many languages that can be compiled to run on .NET and Mono. In many respects the approach is similar to the Java PathFinder, a tool to model check Java applications, developed at Nasa. We introduce the reader to the background of Mono and software model checking. Then we discuss our implementation in full detail, explaining design choices and algorithms used. Finally, we present some experimental data, suggest several improvements that may be useful to investigate in the future.

Preface

In many publcations about formal verification, such as model checking, the author advotes techniques in this field as an excellent remedy to computer errors. While this is at least partially true from a formal standpoint, the tools to make the theaory applicable in practice are usually quite difficult to use.

A trend in the recent years has been to apply model checking to software, so the developer can use his actual product itself as a base for checking. Ultimately, model checking should be a real push-button thing that is as natural to software developers as using a debugger. In my opinion, providing tool that do this is both necessary and interesting.

This ultimate goal is one of the motivations for me to work on this particular subject. It was suggested to me by Theo Ruys as I was still working on an interesting but very formal decision algorithm at IRST. Interested in the practical side and use of the subject I decided to accept the challenge.

Speaking in retrospect, I cannot say I missed much of the practical side. In fact, the larger part of the assignment involved writing code, redesigning some parts, writing more code, redesigning once more, and so on. The project is at least three to four times larger than anything I have written before. I am sure to say that writing a manageable large application is an art, and I think by working on this assignment I have at least gained some valueable experience in it.

The report you are reading is the result of my master's thesis project which took more than than a year to complete. The reason for the delay is both the lack of momentum in my working routine in the beginning of the project, and my constant urge to adjust existing code to coincide with a new design idiom I adapted while solving problems in another part of the code.

The fact that this report is lying in front of you is to a large extend thanks to my supervisor Theo Ruys. He helped me out of a serious low in (study) activity and inspiration in a very friendly and patient way. Also, I very much enjoyed the frequent and often informal chats we had about various technical subjects. It was in fact Theo who taught me how to program eight years ago, and although I have certainly gained experience during those years, he was still able to give new insights in the course of this project. I would hereby sincerely like to thank him for all his help.

Niels Aan de Brugh
Enschede, August 2006

Contents

1 Introduction **5**

2 Mono and the CLI **7**
- 2.1 The Mono Project . . . 7
- 2.2 Mono Overview . . . 8
- 2.3 Machine Model . . . 10
  - 2.3.1 Computing Environment . . . 10
  - 2.3.2 Tasks of the Virtual Machine . . . 11
- 2.4 Languages . . . 12

3 Software Model Checking **18**
- 3.1 Introduction . . . 18
- 3.2 Traditional Model Checking . . . 18
- 3.3 Existing Tools . . . 20
  - 3.3.1 Full Code Coverage . . . 20
  - 3.3.2 Translation Based Tools . . . 21
  - 3.3.3 Abstraction Based Tools . . . 22

4 Implementation of the Model Checker **24**
- 4.1 What is it? . . . 24
- 4.2 Architectural Overview . . . 27
- 4.3 Explorer . . . 29
  - 4.3.1 Exploration Algorithm . . . 30
  - 4.3.2 Rescheduler . . . 33
  - 4.3.3 Deadlock Detection and Assertion Violations . . . 36
- 4.4 Instruction Executors . . . 37
  - 4.4.1 Safe and Unsafe Instructions . . . 38
- 4.5 State Storage . . . 40
  - 4.5.1 Recursive Indexing . . . 40
  - 4.5.2 Doing it Better . . . 44
- 4.6 Backtracking . . . 45
  - 4.6.1 Building the Delta . . . 46

5 Implementation of the Active State **49**
- 5.1 The Aggregation of Parts . . . 50
- 5.2 Data Elements . . . 50
- 5.3 Dynamic Allocations: the Heap . . . 52
  - 5.3.1 Design of the Heap . . . 52
  - 5.3.2 Garbage Collection . . . 55
  - 5.3.3 Allocation Placement . . . 61
- 5.4 Static Data: Classes . . . 66
  - 5.4.1 Class Loading and Initialization . . . 67
  - 5.4.2 Class Storage . . . 68

5.5 Locking 70
5.6 Thread Pool 74

6 Testing and Benchmarking 76
6.1 Test Setting 76
6.2 Test Results 77
6.2.1 Execution Performance 77
6.2.2 Locking: Dining Philosophers 78

7 Future Work 82
7.1 Partial Order Reduction 82
7.2 Detailed Back Traces 83
7.3 Exception handling 83
7.4 Data and I/O Abstraction 84

8 Conclusion 86

1 Introduction

Software has become a normal part of our daily lives. Nowadays people are working with software more than ever before, most of the times without even realizing it. Televisions and mobile telephones are more complex and provide more functionality than a high-end workstation did less than a generation ago. Even a machine that look almost entirely mechanical, such as a car, may run complex software to make sure the machine continues to function as intended.

Unfortunately, with the impressive advancement of technology and functionality comes unforeseen failure. The complexity of the design and implementation of system has become too much for a single human being to grasp. The list of famous computer errors (bugs) is long, and several had extreme consequences. Such as the loss of NASA's Mars Polar Lander and ESA's Ariane 5, a bug in one of the first Pentium series processors Intel released and had to recall later, to even the loss of human life in the Therac-25 debacle.

Of course, not all software bugs need to have such dramatic consequences to be harmful. The loss of an hour's work due to a crashed word processor is both costly and can be downright frustrating to the user.

One of the causes of the increased complexity of software is its increase in size and functionality. A lot of errors introduced in this way can probably be caught by traditional testing, given enough time. Unfortunately, that time usually not available.

However, many systems no longer perform just one sequential task that can be monitored from the start to the end. With the increase in computing power came the possibility to do multiple tasks at the same time. Unfortunately, it is very hard to catch all errors that are caused by concurrent execution of software by means of traditional testing, and a bug that slips through may cause harm a long time after the software has been released.

Although providing fixes and patches long after the software has been released is a common practice nowadays, it is desirable for both the provider and consumer of the software to get most things right the first time.

A mechanical and mathematically inspired technique that is specifically successful in finding errors that are often missed by human testers is called model checking [8]. Using this method, we can test if a formally stated propositions holds on a formalized model of the system. A nice feature of model checking is that it is an automated process, requiring little to no human interaction.

Much research has gone into the field of model checking, resulting in a long list of tools, each with its own strengths and weaknesses. However, all of these tools check properties of a formalized model of a system, not the system itself. With the increase of computational power and development in the theoretical field, a new trend in the model checking world has arisen, that is to apply the method to real software rather than a model. This eliminates the requirement to create a formal model of the system, and thereby the inconsistencies that may be introduced in the modelling process.

The abundance of errors is not the only problem that plagues the software world. A large and still rapidly expanding web of programming languages, operating system and architectures has formed itself over the last two decades. Applications written for one architecture cannot be easily deployed on another, and (legacy) code that has been developed in one programming language cannot be easily used from code written in another. To keep the overview in such a situation requires even more expertice of the developer. Still, a lot of essentially redundant work is being done.

Sun presented Java as a solution to this problem. Java promises to run the same code on a wide variety of operating systems and architectures. By now it has matured in a usable development platform, but it does not solve the incompatibility problem of programming languages. Microsoft's .NET initiative promises to tackle this last problem by providing a system similar to Java that can run code written in many different programming languages.

Inspired by the .NET technology, a group of developers started working on a free, open source version, which they named Mono.

In this project we try to combine the virtues of both software model checking and Mono. That is, we aim to build a tool that is capable of finding software errors in applications written for Mono. By the nature of Mono and .NET this tool would allow developers to check code written in many different languages. We call this tool MMC, short for Mono Model Checker.

The rest of this document describes how we developed MMC. We will start by introducing the reader to Mono (chapter 2), and what is currently happening in the field of software model checking 3. After that, we will discuss the design and implementation of MMC in chapters 4 and 5. In chapter 6 we present experimental results, comparing MMC to a similar tool. We will finish our discussion of MMC with an elaboration on what future work MMC could benefit from (chapter 7), and finally we conclude in chapter 8.

2 Mono and the CLI

In this introductory chapter, we shall describe the Mono platform in enough detail so the reader should be able to understand the rest of this report. This includes a short overview (section 2.2) and a description of the languages and machine model (section 2.4 and 2.3 respectively). We will start with an informative introduction to the Mono project, shortly describing its history and goals.

2.1 The Mono Project

In this section we give a succinct summary of the history and background of the Mono project. It contains no technical information needed to understand the rest of the document, but is provided for the interested reader. For this section only, some background knowledge about the current computer industry is required.

The Mono Project is an open development initiative sponsored by Novell [33]. Its aim is to develop a free implementation of Microsoft's .NET cross-platform development platform.

.NET [34] is marketed by Microsoft as a complete solution for (web) applications and includes many company-wide initiatives. Mono restricts itself to the development platform only. Novell sponsors the development of Mono so developers will be able to deploy their software on Linux as well as Windows, taking a competitive disadvantage away from the former platform.

Although technically not strictly related, .NET and Mono both promote the use of the new C# language. This high-level object-oriented programming language resembles the Java language, but contains more features and resembles C++ more closely. The language was developed by Microsoft, and standardized in ECMA-334 [13].

In February 2001, Miguel de Icaza started working on a part (mcs, the C# compiler) of what is now the Mono project. At time De Icaza was working for Ximian, a company he co-founded in 1999, which was later bought by Novell. In July 2001, Ximian announced the Mono open source project as an effort to increase developer productivity for the Gnome[1] community. By then, De Icaza's compiler was able to parse itself.

In June 2004 Mono announced their 1.0 release. At the time of writing version 1.1.16.1 is the newest version. The product is currently being used for the development of several commercial and open source (desktop) applications (a list can be found on [33]). At the moment it is not being used as much as main-stream programming languages like C, C++ or Java.

In many aspects Java is Mono's obvious competitor. Both serve the same purpose of providing a cross-platform application platform, targeted by a powerful yet easy to learn programming language. There are at least three identifiable reasons why Mono is (at the time of writing) not quite as popular as Java.

First of all, a reasonably fast version of Mono is only recently available, whereas Sun's Java has been available for over ten years. Mono does not have the advantage of being the only product of its kind on the market like Java had ten years ago. Secondly, support for the language features of the C# 2.0 standard (such as generic types) is still experimental. Finally, the performance of Mono is not quite as good as that of the newest version of Java, although Mono outperforms many popular languages such Python and Ruby.

Regardless, we feel the choice for Mono is a valid one, mainly based on the technical advantages of the platform. Some of these technical virtues will be discussed in the technical overview given in the rest of this chapter, specifically in section 2.2.

Also, we think Mono will be more easily adopted by developers of desktop applications. Mono already has binding for several powerful GUI frameworks, a field where Java has failed to take advantage of its head start. Additionally, a likely yet speculative argument is that Mono applications will be easily deployable on the .NET platform integrated in future versions of Microsoft's operating system.

Aside from technical considerations, we consider the free and open nature of Mono's development to be harmonious with the goals of this project.

With this last remark we conclude this informative section, and continue the rest of the chapter looking at the technical side of things, starting with a technical overview in the next section.

2.2 Mono Overview

virtual machine

Mono [33] is a platform to develop and run applications. It is based on the concept of a *virtual machine*, i.e. an extra layer between the application and the actual operation system, thereby allowing for development of easily

[1]Gnome is a free and open-source desktop environment for Linux and other unices. Ximian has written several applications for the Gnome desktop.

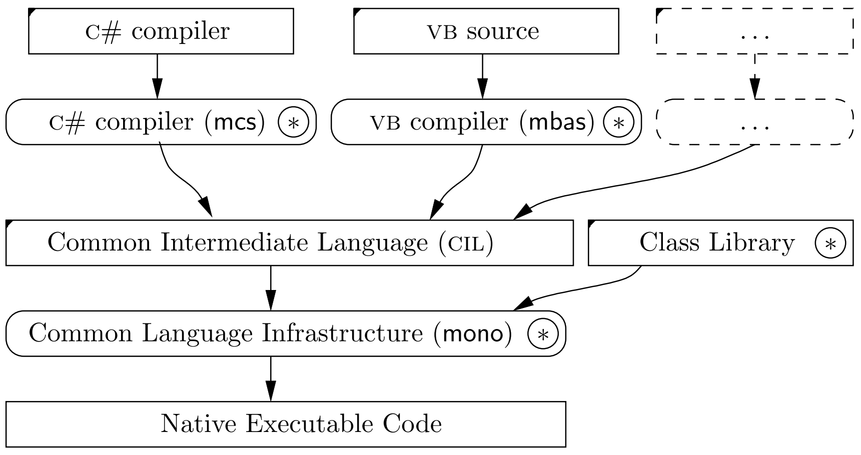


Figure 2.1: Architectural overview of Mono.

portable applications. The reader might be familiar with Sun's Java Virtual Machine, which is very similar.

Mono is a free and open implementation of the Common Language Infrastructure (CLI), a standard developed by Microsoft which was published by ECMA as standard 335 [28, 14]. Additionally, Mono is bundled with a C# [13, 37] compiler, mcs.

The CIL is not tied to a specific programming language. Quite to the contrary, the CLI describes all language independent aspects that are needed for components written in different languages to work together. This includes a common virtual execution system, type system and executable file format.

The situation is sketched in figure 2.1. Parts bundled with Mono are marked with an asterisk. Additionally to mcs and mbas, compilers for many languages can be downloaded from the internet.

Additionally to the compilers and a run-time environment, Mono comes with a class library containing common functionality such as threading, networking, data containers, an introspection framework, and much more. For a full overview we refer the reader to the Mono website [33].

A library that has proven to be extremely useful to us is the Cecil library. It is still not an official part of Mono, but can be downloaded separately [6]. We used Cecil to inspect assemblies, compiled pieces of software. Unlike the reflection framework in the class library Cecil is capable of inspecting those files at instruction level.

Completing the high-level overview of Mono, we will now continue our introduction providing a little more detail, starting with the machine model as provided by Mono (and MMC) in the next section.

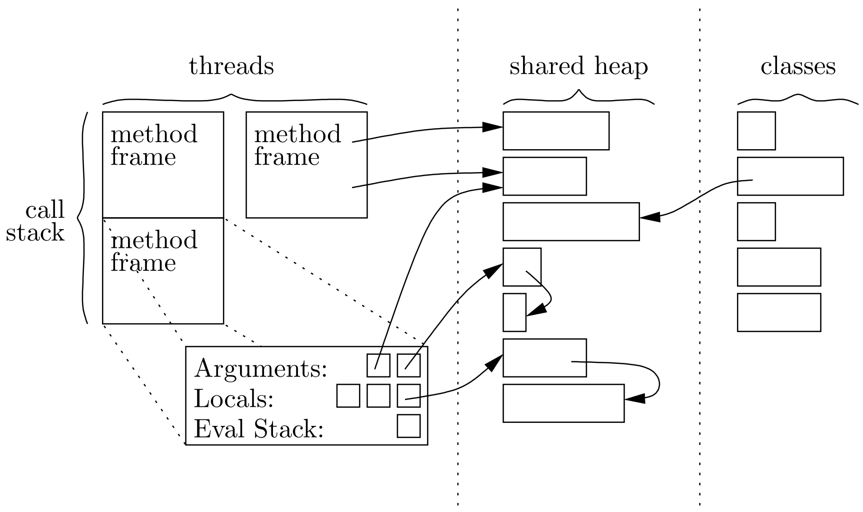


Figure 2.2: Computing environment of Mono.

2.3 Machine Model

In this section we will describe what the virtual machine (VM) model looks like. In a few aspects the virtual machine is similar to a typical computer, but in most it is different. Where applicable, we shall use the readers prior knowledge about computers in general. We do not assume any prior knowledge about virtual machines.

First, we will describe the Mono world as seen by the code, i.e. the computing environment (section 2.3.1), and next discuss what tasks the VM performs (section 2.3.2), that is what we can expect from the run-time environment.

2.3.1 Computing Environment

The computing environment is a stack-based machine with access to a heap for dynamic allocations, and a static data part. It is similar to the Java VM, but unlike a normal computer. Figure 2.2 depicts a sketch of the environment.

Similar to a normal computer, data blocks (frames) for the called methods is organized in a logical stack: calling a method pushes a frame on top of the call stack, and returning from a method pops one off the stack.

Computations are done by pushing values on and popping values off an *evaluation stack*. For example, to calculate 2+3, we first push 2 and 3 on the stack, and than call the `add` instruction, which will pop 2 and 3, and push back 5. Registers (found in all but a few hardware computers) do not exist. This approach is common for virtual machines, since the number of registers at our disposal is only known at run-time, so the register-allocation can only happen then. Also, since the architecture of the CLI is based on multiple compilers and just one run-time, it makes sense to keep the compilers as simple as possible.

The memory model of Mono is divided in a *stack*, a *heap* and a *static* part. The stack is a local block of memory that is created when a method is called, and that is automatically removed when returning from that method. Both local variables and arguments are stored on the stack. The heap is a pool containing dynamically sized blocks of memory that are allocated at run-time. Multiple processes may access the heap simultaneously. The static part contains space to store loaded classes, and is in many respects comparable to a store of global variables. At run-time at most one instance of a class is loaded, and the classes can be accessed by any of the running threads.

reference and value types

Two kinds of values exist in the CLI: reference and value types. The first kind includes arrays, objects and delegates, which are all allocated at the shared heap. Multiple threads may access this data. It is up to the programmer or compiler to provide concurrent access control. To access an allocation on the heap, a reference to it is needed.

Value types include primitive types (such as integer and floating point numbers, boolean values, and so on), and references to allocations on the heap. Additionally, depending on the high-level language used, a special kind of object is available, called a struct in C#. This type of object is allocated on the stack rather than the heap, and is typically used for relatively small aggregations of data.

2.3.2 Tasks of the Virtual Machine

Like a normal computer, the VM's task is to run client applications by processing instructions (what these instructions look like will be discussed in section 2.4). But very much unlike a normal computer, the *virtual* machine is implemented in software instead of hardware. The VM itself needs some sort of computing machinery to run on, but which kind of hardware is mostly invisible (and irrelevant) to the client application. It can therefore regard the VM as a uniform computing environment regardless of underlying hardware. *Hardware abstraction* is an important task of the VM.

As a machine that abstracts away underlying hardware running object oriented (OO) software, the VM needs to provide high-level OO functionality. This functionality is traditionally provided by the compiler or libraries.

A compiler for a typical (more traditional) object-oriented programming language (e.g. C++) deals with all details involving object-orientation. Details such as handling polymorphism and virtual calls, alignment of fields and the stack, proper handling of exceptions, just to name a few. In case of Mono, the compiler can remain agnostic about all these aspects, as they are handled by simple instructions to the VM.

The VM also burdens itself with additional tasks typically done by the programmer. Examples include locking, disposing of no longer used allocations, wrapping primitive types in objects and process (and thread) management. The former and latter aspect are handled via calls to declarations in the class library, the other two again via instructions.

The VM is instructed by the client application in two possible ways. First of all, the client application contains CIL instructions, which are discussed in more detail in the next section. Second, it can call methods in the class library. However, no code for these methods is provided inside the class library. Instead the VM is expected to handle the call internally. Therefore, these calls are called internal calls. In many respects they resemble the software interrupts (or traps) used in traditional programming.

In the next section we will discuss two languages relevant in the context of Mono, the new language C# and the format of the compiled code, CIL.

2.4 Languages

As described above, many high-level programming languages may target the CLI. The MMC itself is written in C#. An introduction to this language can be found in [11], and a language reference in [37]. C#'s syntax and language features resemble Java and C++. There are some subtle differences, which are not important for this project, though.

Mono is capable of running both *safe* and *unsafe* code. The first is code that is executed by the virtual machine. The latter is compiled directly into machine code. It may contain low-level constructs such as pointers, and allocations of arbitrary and untyped blocks of memory. MMC is not capable of checking unsafe code, so we do not discuss it here.

The C# source code is translated into a common (intermediate) language (CIL), which can be regarded a very high-level form of assembler code. We shall (partially) introduce this language by example. For a full description we again refer to [28].

The CIL code example were extracted from the binary files using a small tool written by us, CILDUMP. Like MMC, it uses the Cecil library to inspect assemblies and extract CIL bytecode.

Example 1 (Counting characters)**.** As a first example, we look at the CIL code for a simple routine that counts the number of occurrences of a character in a string, and returns it. The C# code for this method is listed in listing 2.1.

The following code results from compiling[2] and disassembling the resulting PE file.

[2]We used MCS version 1.1.13 when writing this document.

Instruction	Description
`ldc.`$t.c$	Push constant c of the type t.
`ldarg.`x	Push argument x.
`ldloc.`x	Push local x.
`ldelem.`t	Pop i and A. Push element i of array A as type t.
`ldlen`	Pop A. Push the length of array A.
`ldfld F`	Pop O. Push the value of field F of O.
`conv.t`	Pop a. Push a, converted to type t.
`add`	Pop element x and y, and push $x + y$.
`sub`	Pop element x and y, and push $x - y$.
`mul`	Pop element x and y, and push $x \times y$.
`div`	Pop element x and y, and push x/y.
`stloc.`x	Pop element, and store it in local x.
`stfld` F	Pop v and O. Store v in field F of O.
`br` B	(Unconditionally) jump to address B.
`bne` B	Pop a and b. Jump to address B if $a \neq b$.
`blt` B	Pop a and b. Jump to address B if $a < b$.
`callvirt` M	Perform a virtual call to M.
`call` M	Perform an instance call to M.
`newobj` M	Allocate a block of memory on the heap to hold the declaring class of M. Then call M, passing a reference to the new allocation as its first argument.
`ret`	Return from a called method. If the stack of the callee contains a value, push this value on the stack of the caller.

Table 2.1: Some common CIL instructions and their description.

```
public static int CountChar(char c, string str) {

    int count = 0;
    char[] chars = str.ToCharArray();

    for (int i=0; i < chars.Length; ++i)
        if (chars[i] == c)
            ++count;

    return count;
}
```
Listing 2.1: Counting characters in a string (C# code).

For a description of the individual CIL instructions, we refer to table 2.1. The push and pop actions described in the description are performed on the evaluation stack of the method that executes the instruction.

```
0000: ldc.i4.0      0011: br 0033       0026: ldc.i4.1      0033: ldloc.2
0001: stloc.0       0016: ldloc.1       0027: add           0034: ldloc.1
0002: ldarg.1       0017: ldloc.2       0028: stloc.0       0035: ldlen
0003: callvirt ...  0018: ldelem.u2     0029: ldloc.2       0036: conv.i4
0008: stloc.1       0019: ldarg.0       0030: ldc.i4.1      0037: blt 0016
0009: ldc.i4.0      0020: bne.un 0029   0031: add           0042: ldloc.0
0010: stloc.2       0025: ldloc.0       0032: stloc.2       0043: ret
```

The numbers before the colon on each line give the instruction offset in bytes, it is not a line number. Not all instructions are equally big. For example, the `ldc` instruction at offset 0 is only one byte, whereas the `callvirt` instruction takes 5 bytes. This is because it includes an operand, in this case a reference to a method, that takes up space.

We will shortly describe the code by block. In instructions 0 to 8, the two local variables count and chars are initialized. The method name that was omitted in the code is of course ToCharArray. Instruction 8 and 11 set i to zero and branch to the block where the loop condition is checked (instruction 33 through 37). Notice that although in C# the value of i is only accessable in the for-construct, it is treated like any other local variable in CIL. Scoping is purely a compiler thing.

If the branch condition holds, the the ith element of chars is read onto the stack, and compared to c, the first argument. If these are equal, i is incremented in block 25–28. Notice that this simple operation takes four CIL instructions, and thus is not atomic. Finally, the step statement of the for loop is executed in instructions 29–32, after which we arrive again at the loop condition check.

Notice that the CIL code does not allocate memory for its local variables and evaluation stack. This is a task left to the VM. The reader might be interested to know the code is not linked to any libraries. In fact, Mono does not have a separate linker tool. So in this case, the code of ToCharArray is not included in our assembly, and is to be loaded dynamically from mscorlib.dll by the VM. □

We will a more advanced example, introducting several C# language features. The CIL code will show some of them are mere syntactic sugar.

Example 2 (A simple singleton linked list)**.** Please take a look at the C# code in listing 2.2. It defines a class called ProcessList which is meant to be used as a *singleton*. That is, it is meant to have just one instance, accessable via a static member (line 5). To ensure no other instances are created, the constructor is made private (line 34).

```
namespace ProcessControl {

  sealed class ProcessList {

    public static readonly ProcessList pl =
        new ProcessList();

    class Process {

      Process m_next = null;
      public readonly int PID;

      public Process Next {
        get { return m_next; }
        set { m_next = value; }
      }

      public Process(int pid) {
        this.PID = pid;
      }
    }

    Process m_head = null;

    public void AddProcess(int pid) {

      Process new_proc = new Process(pid);
      lock (this) {
        new_proc.Next = m_head;
        m_head = new_proc;
      }
    }

    private ProcessList() {}
  }
}
```

Listing 2.2: Process ID List (C# code).

The process list holds a linked list of Process classes (lines 8–21). These classes hold a process ID and a reference to the next process in the list. The PID is a readonly field, which means it can only be assigned in the instance or type constructor. The constructor is translated into the following CIL code.

```
0000: ldarg.0
0001: call System.Object::.ctor()
0006: ldarg.0
0007: ldarg.1
0008: stfld ProcessList/Process::PID
0013: ret
```

First of all, it is important to know the this pointer (i.e. a pointer to the object the method is invoked upon) is passed to each member function as the first argument. The caller of the constructor is responsible for allocating memory for the object to be instantiated, and passing it as the this pointer to the constructor.

The process constructor implicitly calls the constructor of its base class, which in this case is System.Object, the method of all classes in C#. It then assigns its second argument, which is its first explicit argument pid, to its field PID, and returns.

Method AddProcess constructs a new Process class, and prepends it to the list. However, since multiple processes may be executing this code, it applies locking to prevent two processes update the m_head field at the same time, corrupting the list. Locking is explained in detail in section 5.5. For now, it is sufficient to know this solves the problem. Let us see how this is translated into CIL code (names of field and method have been shortened).

```
0000: ldarg.1
0001: newobj Process::.ctor
0006: stloc.0
0007: ldarg.0
0008: stloc.1
0009: ldloc.1
0010: call Monitor::Enter
0015: ldloc.0
0016: ldarg.0
0017: ldfld ProcessList::m_head
0022: callvirt Process::set_Next
0027: ldarg.0
0028: ldloc.0
0029: stfld ProcessList::m_head
0034: leave 0046
0039: ldloc.1
0040: call Monitor::Exit
0045: endfinally
0046: ret
```

Instructions at offsets 0–6 create a new Process, and stores it. New objects are created by the `newobj` instruction. It creates a new allocation on the heap, and passes it as the this pointer to the constructor, which is called just like a normal method.

It then stores a local copy of the this pointer (instructions 7 and 8). This may seem rather silly, but it does this because we will be locking on it. The compiler always stores code to store a copy of whatever it is locking on locally. This is quite sensible, since we need the same value again at the end of the lock block. The expression between parenthesis on line 28 may have changed by then. Obviously not in this case, but there is no easy way for the compiler to know this.

Next, it enters the protected block of code by calling the (static) Enter method. It then loads and the m_head field, and assigns it to the Next property of the Process it created earlier. The property mechanism is an example of syntactic sugar in C#, and no such thing exists in CIL. The get and set code blocks are translated into an explicit getter and setter method by the compiler, in this case get_Next and set_Next.

As a short example, we give the CIL code of set_Next. We will not elaborate on this code.

```
0000: ldarg.0             0002: stfld ProcessList/Process::m_next
0001: ldarg.1             0007: ret
```

After updating the m_head field in instructions 27–29, we leave the lock block. This is done by giving the offset of where to continue, but first we have to execute the implicit *finally* clause the compiler has inserted for us. This clause spans until the `endfinally` instruction, so instructions 39–45. It unlocks this by calling the static Exit method in the Monitor class.

The reason the compiler put the unlocking code in a finally clause is because that clause will always be executed, even when an exception was raised in the protected block. We shall not discuss exceptions in this document since the theory would span multiple pages, and is not essential to understand this example. People familiar with the concepts of exceptions will probably agree the compiler put the unlock code where it belongs.

This concludes our discussion of this example. We have not shown all CIL code, but the rest will most likely not give additional insights. □

With these two example we hope to have given some insight in what CIL code looks like. Of course, lots of details are omitted. For the interested reader, we refer to the CLI standard [28] for an complete and precise overview of CIL and its file format.

This concludes our discussion of Mono. We have introduced the reader to some of its history and relation to .NET, the architecture of the run-time and the components that are bundled with Mono. In the last section we gave two examples of C# code translated into CIL to give the reader an impression of what this code looks like.

3 Software Model Checking

The Mono Model Checker is a software model checker for Mono applications. In this chapter, we will zoom in on the field of software model checkers, see some of the underlying theory, and take a look at what other tools are currently available, most notably the Java Pathfinder.

3.1 Introduction

A model checker is a software application that verifies formal systems. By *verify* we mean checking that a number of desirable properties hold for the given system. An important feature of model checkers is that this verification process can run without any interaction from the user.

The *formal system* is usually a labelled transition system, i.e. an automaton that is an abstract mathematical model of a real system. This automaton has a graph-like structure, where nodes coincide with the states of a system, and directed edges coincide with transactions of one state to another.

The goal of *software* model checkers is to bring the model checking goodness to the field of software development. That is, to provide tools to verify software in an automated manner. This means the clean and mathematical transition system is replaced by a complex and dynamic real product.

After shortly introducting traditional model checking in section 3.2, we will give an overview of what the field of software model checking currently looks like (section 3.3), and where MMC fits in.

3.2 Traditional Model Checking

First of all, let us stress this section only touches the surface of model checking theory. Most notably, we do not discuss the most important algorithm used in any model checker: the decision algorithm. For a fairly gentle introduction into the theory we refer to [15]. Although this section deals with classic model checking, we keep in mind that we ultimately want to develop a software model checker, and not all theory is applicable to both fields.

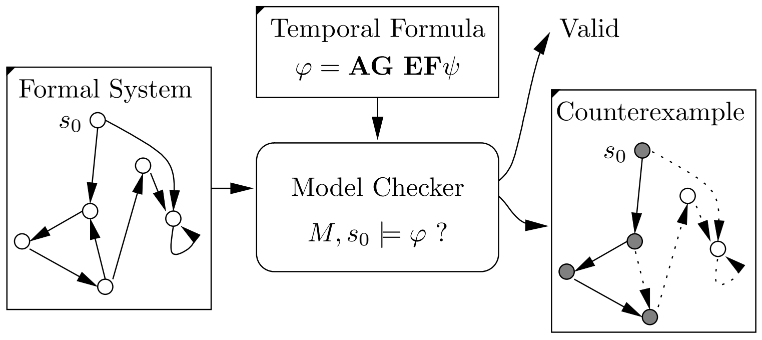


Figure 3.1: High-level overview of the model checking process.

As mentioned in the introductory section, traditional model checking is the process of verifying formal model (of a system) in an automated, algorithmically way. Formally, let M be the model with initial state s_0, φ a property that needs to hold on M, then model checking boils down to checking whether $M, s_0 \models \varphi$. The relation between the different players in the model checking process is depicted in figure 3.2.

The *model* (or formal system) is usually a Kripke structure, a tuple $(\mathcal{S}, \mathcal{I}, \mathcal{R}, \lambda)$ containing a countable set of states $\mathcal{S}$, a subset $\mathcal{I} \subseteq \mathcal{S}$ of initial states, a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, and a labeling mapping $\lambda : \mathcal{S} \rightarrow 2^{\mathrm{AP}}$.

Intuitively, $\mathcal{S}$ models all states a system can have (e.g. on and off for a simple light switch), $\mathcal{I}$ defines the state the system starts in (the light switch is initially off) and $\mathcal{R}$ defines how the states relate, i.e. which states follow from what other states (e.g. state on can follow from state off, and vice versa). Mapping λ defines what propositions, or boolean facts, hold for a given state. Continuing the light switch example, suppose we have propositions s (the girl in the room can sleep) and e (electricity is being used) mapped onto states as follows: $\lambda = \{\mathsf{on} \rightarrow \{e\},\ \mathsf{off} \rightarrow \{s\}\}$.

The properties that need to hold in the model are typically given as temporal formulae. Several dialects exist, varying greatly in expressive power, succinctness, and complexity of validity checking. The formula used as an example in figure 3.2 is CTL [8] and specifies that in each and every state ($\mathbf{AG}$), there should be at least one path to a state ($\mathbf{EF}$) that satisfies some formula ψ.

Completing our light-switch example, we might demand that "in any given state, there is always a future state in which no electricity is being used". In CTL this is expressable as $\mathbf{AG}\ \mathbf{EF}e$.

We will not go into the details of temporal logic since we will not be using them in the rest of the document. For a rather extensive coverage of the theory of temporal logic we refer the reader to [3, 15].

The decision procedure, i.e. the algorithm that ultimately decides wether a given formula φ holds on a given model M, forms the very hearth of any model checker. The internals vary greatly from one algorithm to the other, depending on the logical dialect of φ.

At this time we only check implicit properties of the system. That is, MMC does not accept arbitrary formulae, but checks for certain properties hard-coded in the implementation. This is why we have chosen not to discuss the decision procedure of e.g. CTL and LTL [32], as these algorithm are non-trivial and do not necessarily give greater insight in how MMC works. We refer instead to [15].

Many tools have been developed that implement the referred theories. The interested reader might want to look at Spin [24] or NuSMV [7]. In the next section we will introduce several software model checking tools.

3.3 Existing Tools

In this section several existing tools [38] in the field of software model checking are introduced.

The tools differ greatly in the way to achieve their common goal, i.e. to help the developer in improving the quality of software, especially in the amount of detail that is included in the analysis. Some tools consider one aspect of the software, e.g. the control flow, whereas others try to consider as much detail as technically possible.

We will begin our discussion of related work with the tools that try to check the entire code of an application, or at least most of it (section 3.3.1). This is where MMC is to be categorized. After that, we will discuss several other projects that take another aim at trying to find software errors, i.e. tools based on translation (section 3.3.2) and on abstraction (section 3.3.3).

3.3.1 Full Code Coverage

Tools in this category aim preserve as much detail as possible of the software to be checked, rather than concentrate on just one aspect. This makes full code coverage tools very powerful and versatile since they can be universally applied to software. However, as a jack of all trades, the analysis of one particular aspect may fall short compared to the that of a tool that was specifically designed for that aspect. We will discuss two software model checkers here: JPF and XRT.

The Java PathFinder [39, 40], or JPF (currently at version 4) is a prominent project in this category. It pioneered the concept of implementing a software model checker as a virtual machine that simulates the (binary) code of the application to be checked. In case of the JPF, this is Java bytecode. The (Java) source code is not required.

The JPF is an explicit-state model checker that systematically explores the state-space of a Java program, thereby generating it on-the-fly. It reduces the size of the state-space by applying partial order reduction (POR) techniques [17, 25, 36], as well as symmetry reduction [9, 18]. The size of each individual state is reduced using the recursive indexing method, developed by Holzmann et al. [26]. By systematically exploring the state-space the JPF aims to find deadlocks and uncaught exceptions.

The list of similarities between MMC and JPF is long, which is not surprisingly, as JPF has served as both an example and an inspiration. When discussing the implementation of MMC in chapters 4 and 5 we will point out these similarities.

At Microsoft Research, XRT [19], an exploration framework for .NET is being developed. Like MMC and JPF it is suitable for transactional exploration, although the XRT allows arbitrary exploration strategies.

A notable difference between XRT and JPF and MMC is the first is based on rewriting of the code. To achieve this, the framework uses instruction rewriters which alter the code of a method, the basic entity of code in XRT. An instruction rewriter can substitute instructions, or even rewrite the whole control flow of the method. Another important difference is XRT allows state extensions, most notably an extension that allows a state to be stored symbolically.

Like MMC and JPF, XRT uses compression for stored states, although the technique used (hash all components into a vector, [19]) is less sophisticated than the recursive indexing used by JPF and MMC. And unlike JPF the source code, and indeed a working implementation, of XRT is currently not available.

Both XRT and JPF work on the bytecode generated by a compiler. In the next section, we will see two tools that work with the uncompiled source code.

3.3.2 Translation Based Tools

Tools that fall into this category re-use existing (traditional) model checkers. The software to check is translated into a model in the input language of another tool. This tool is then used to do the actual checking. Both Bandera [20, 10] and the first edition of the Java PathFinder [21] take this approach.

JPF1 and Bandera both accept Java source code as input. The first outputs a model described in PROMELA, the input language for the SPIN [24] model checker, the latter can output in several other languages as well.

However, as explained in [40, 22], there were two important disadvantages to the approach of translating source code to (say) PROMELA. The first of which is the source code actually needs to be available. This may not be the case when the application to check uses third-party libraries that are provided without source code, or if code is dynamically loaded over the network.

Another perhaps more immediate problem that was encountered was there is no complete mapping of the input language (Java) to the output language (PROMELA). Although the latter is a very expressive specification language, it is not as rich as a real programming language such as Java. For example, PROMELA lacks floating point numbers. This is why JPF1 was abandoned, and superseded by the JPF discussed in section 3.3.1.

Bandera [20] is also based on the concept of source code to model language translation. It focuses on an open modular design and the reuse of existing technology [10]. Bandera's approach resembles that of an optimizing compiler: the Java source code is parsed and translated into an intermediate language (Jimple). Given the property being checked, two optimizations are performed on the Jimple code.

First of all, lines that are not relevant to the property being checked are sliced out. Second, an abstraction engine specialized the Jimple code. For example, for the specific property many possible values of a variable may be equivalent. After these two analysis phases, the Jimple code is translated into another intermediate language. This result is finally translated to the input language on an existing model checker.

The techniques used in Bandera are advanced, but its approach differs greatly from MMC. And as a translation based tool, Bandera suffers from the same drawbacks as the first Java PathFinder.

3.3.3 Abstraction Based Tools

Tools in this category handle the complexity of software by (at least initially) cutting away code (and thus behavior).

Tools based on abstraction construct an abstract over-approximated model of the program to check, and analyze this model rather than the original program. If an error is found, the tool has to check whether this error is also present in the original program, or merely the result of over-approximating the programs behavior. In the former case, the tool has successfully identified an error in the program. In the latter case it needs to adjust the abstract model to rule out the errornous behavior. This process is called refinement.

Slam [2], developed at Microsoft Research, is a tool to do reachability analysis for sequential C programs, specially device drivers. It consists of three programs that together form the Slam toolkit.

The first program, C2BP, translates a C program into a boolean program, which consists of the control graph of the original program, but with only boolean variables and values. Next, another tool (Bebop) is used to check for the reachability of a specific program statement (e.g. an error). If the statement is not reachable, it is also not reachable in the C program, so the work is done.

If the statement is reachable, Slam check if this path is indeed feasable in the orginal C program using a third tool, Newton. Newton employs a technique known as symbolic evaulation. If the path is feasable, Slam reports the path to the user, and is done. If it is not, passes the infeasable trace to C2BP, which refines its boolean program, and the checking cycle can start again.

A big difference of Slam's approach to that of MMC and JPF is that it only works on sequential programs. It does not handle multi-threaded applications.

Blast [23] takes an approach similar to Slam. That is, it uses a check-and-refine loop on an abstract model, making it more concrete on every iteration. However, Slam is freely available on the internet [4], including source code.

This concludes our introduction in the field of software model checking. We have given an impression of what traditional model checking looks like, although we omitted relevant theory that cannot be applied to the field of software model checking. After that we introduced and shortly described several related tools. For a comprehensive overview of many tools currently available, we refer the reader to [38].

In the next chapter we will discuss the design and implementation of our own tool, MMC, in full detail.

4 Implementation of the Model Checker

To demonstrate the formal checking techniques described in chapter 3 can be successfully applied to something as complex as software, we wrote a first implementation of a software model checker. The design and implementation were heavily inspired and influenced by the Java Path Finder 2 [40], a similar project for the Java platform.

This chapter describes this implementation in detail, i.e. the architecture, design, and its non-trivial data structures and algorithms. For each of these topics, we will motivate the choices made, and discuss the strengths and weaknesses of the chosen approach.

Note that at the moment the intended audience for this tool is other software developers who want to contribute to and experiment with the field of software model checking. The current implementation should be regarded as a scaffolding to "plug" new ideas and methods into. As such, extensibility and simplicity of design is a prime concern.

In the text, we shall often refer to graphical view of some part of the design. The notation used is not standard, but should be easy to understand to those who are used to reading class diagrams. For a description, refer to figure 4.1.

In the following text, we first explain what the program actually does (section 4.1), then describe the architectural overview (section 4.2), and continue to describe the non-trivial parts of the implementation one by one.

4.1 What is it?

The Mono Model Checker (MMC), is a tool to aid a programmer in finding bugs in his or her program. The name contains two important aspects: it is a tool aimed at aiding *Mono* developers and is developed in Mono, and it does this by applying *model checking* techniques.

As described in chapter 2, Mono applications are run on a virtual execution system (VES). The programmer writes the program in some language, and a compiler then translates it to a common intermediate language (CIL), which the VES then executes.

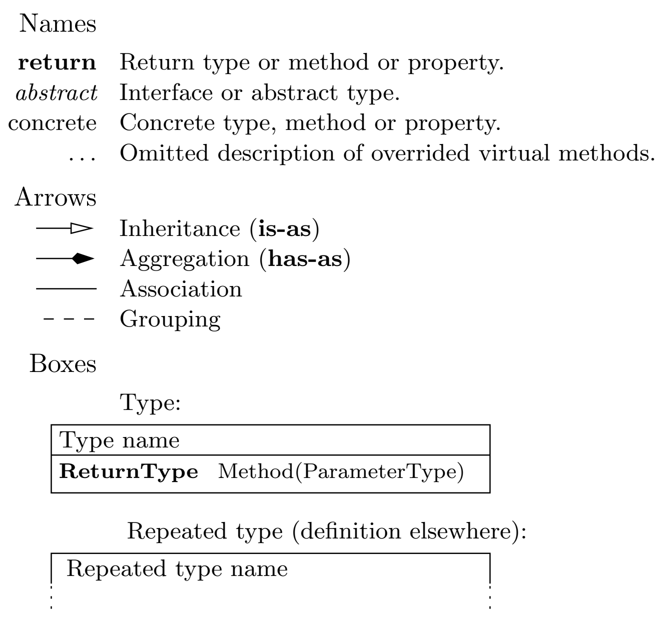
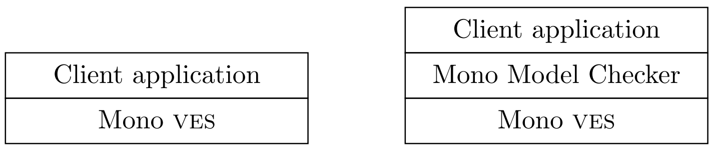

Names
return Return type or method or property.
abstract Interface or abstract type.
concrete Concrete type, method or property.
... Omitted description of overrided virtual methods.
Arrows
Inheritance (**is-as**)
Aggregation (**has-as**)
Association
Grouping
Boxes
Type:
Type name
ReturnType Method(ParameterType)
Repeated type (definition elsewhere):
Repeated type name

Figure 4.1: Legend for design diagrams.

The MMC simulates the execution of CIL code by implementing its own VES. This VES is not as fast as Mono, but it does give us full control over everything a simulated application does. See figure 4.2 for a graphical view where the MMC fits in.

The client application should see no difference between running on Mono or the MMC, i.e. the MMC should behave exactly the same as the Mono VES would.

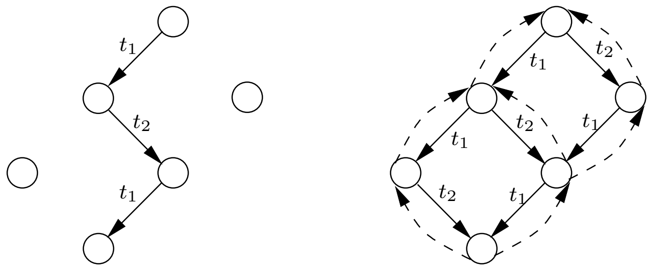


(a) Normal execution (Mono) (b) Full exploration (MMC)

Figure 4.2: Placement of MMC in the computation model.

This is made easier by a large extend by the fact that the MMC itself runs on the Mono VES. Calls and operations can be "passed down" by the MMC by performing exactly the same action as the client application. The result it gets from Mono can then be passed back to the client. Some examples of such calls are arithmetic and system queries.

Mono supports multi-threaded applications, i.e. applications that have



(a) Execution (Mono) (b) Full exploration (MMC)

Figure 4.3: Execution vs. Exploration

multiple concurrent processes that have access to the same data, but run in parallel. Since the number of threads may exceed the number of physical processing units (CPUs) in a system, the virtual machine should implement some scheduling mechanism to divide blocks of execution time among the threads. An ordered sequence of threads that are run in sequence is called an *interleaving*.

interleaving

An important difference between the MMC and Mono virtual machines is that the first is capable of systematically and completely executing all possible interleavings, whereas the first only executes one interleaving. Therefore, the MMC is capable of finding errors that may stay invisible when running the application on Mono.

state space exploration
state
transaction

The act of systematically running and checking interleavings is called *state space exploration*. The model of a program's execution is made up of *states* (snapshots of the virtual machine data), and *transactions* between those states modelling the (partial) execution of a thread.

To facilitate the state space exploration, the MMC is capable of storing its state, and return there later to choose a different thread to run, exploring all possible interleavings in the process. We shall illustrate this using an example.

Example 3 (Exploration vs. Execution)**.** Consider figure 4.3. In this example the program to check consists of two threads t_1 and t_2. At this time, we abstract from what is behind the states (the circles) or what is actually being done in a transaction (the arrows), but we note that t_1 has work for two "steps", and t_2 for one. The transactions (or steps) are numbered to clarify the order of execution.

The left model gives a graphical impression of what is being done in a regular virtual machine, such as Mono. First, thread t_1 does one step, then t_2 (thereby completing its work), and then t_1 finishes. We write this interleaving as (t_1, t_2, t_1).

The right model is how MMC explores the same system. The system starts by exploring interleaving (t_1, t_1, t_2), but instead of stopping when all work is done, it *backtracks* to the situation where another thread could be run instead of the one chosen before. In this case, this is the situation where only the first step of t_1 has been executed, and continues to explore the interleaving (t_1, t_2, t_1). After that, the system backtracks again, this time to the first state where it originally began, and explores last possible interleaving (t_2, t_1, t_1), thereby completing executing all interleavings. □

By systematically exploring all interleavings, MMC is capable of finding all (causes of) *deadlocks* and *assertion violations*. A deadlock is a situation where the system has no runnable processes, but it is not terminated in the intended way. A common cause of deadlocks is inproper use of locking or synchronization. *Assertions* are checks the developer puts in the source code to check if a specified condition holds. Exploiting the fact we explore all possible behaviors of the system, this condition is checked for every situation. Both checks are discussed further in section 4.3.3.

Now that a shallow overview has been given of the system, let us see how this reflects in the architecture of the system.

4.2 Architectural Overview

In this section we will describe the architecture of the MMC, i.e. its high-level structure, thereby abstracting away from design and implementation. As described in section 4.1 we described that the tool runs applications like a virtual machine (or VES), and is capable of restoring a previous state. This reflects directly in the architecture of the system.

Consider figure 4.4, a schematic overview of the architecture, its most important building blocks and interaction between those blocks. Central to the architecture is the *explorer*. This component drives the state space exploration.

explorer

On the left-hand side in the figure, the part of the MMC that provides the virtual execution environment is depicted. It is based on two components: an active state and instruction executors. The first holds the state of the virtual machine, i.e. the running threads, allocated objects, and so on. This is rather a big component, and will be discussed in full in chapter 5.

active state

The latter part of the VES, the instruction executors, are responsible for executing a single CIL instruction. A typical instruction executor queries and update the active state. The construction and starting of the executors is done by the explorer.

instruction executor

To the right we find the parts of the MMC that are used for storing and restoring states: the state storage and a backtrack stack. Note that the two have only been put together in the figure since they serve the same high-level goal. There is no other immediate relation between the two.

At certain points in the execution path, the explorer has to store the

state store

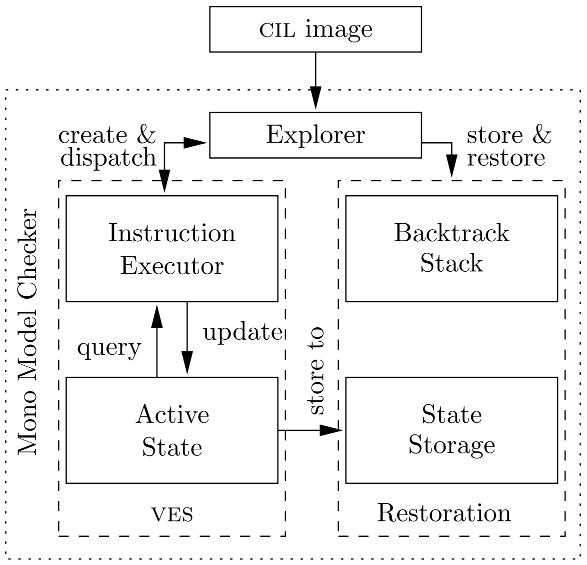


Figure 4.4: MMC architecture

active state in the *state storage*. This storage is used to check if a certain state has already been explored in the past, so we do not need to explore it again.

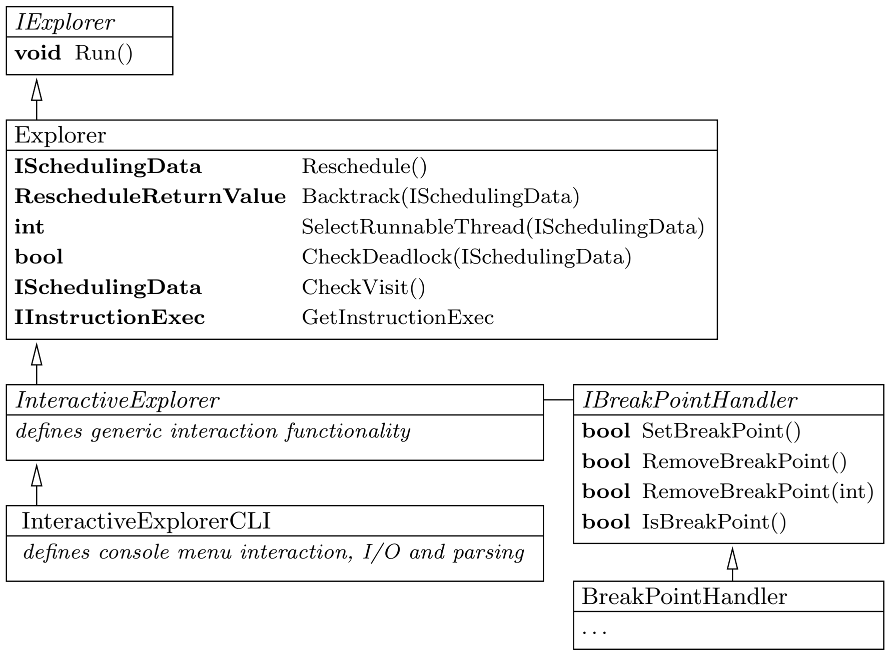
backtrack stack

The *backtrack stack* contains the sequence of states (i.e. the *path*) that has been explored so far. It allows the explorer to restore a previously visited state. It contains scheduling related information, most importantly the threads that have not yet been run from that state, and all data needed to restore the previous state. The way steps are taken forward in the exploration, and then reverted, is reflected in the last-in-first-out nature of the stack.

Since most of the implemented functionality serves the explorer (section 4.3), and more specifically the exploration algorithm (section 4.3.1), this topic will be covered first. Next, the instruction executors will be discussed (section 4.4).

We will continue the chapter with a description of the state store (section 4.5) and the backtracking mechanism (section 4.6).

As noted before, the active (or VM) state is discussed in chapter 5. Most, if not all, of the theory in this chapter will be understandable with just basic knowledge of a typical object-oriented computing environment. Otherwise, the reader might want to read section 5.1 for a quick introduction what structures are to be expected in the VM state.



IExplorer
void Run()
Explorer
ISchedulingData Reschedule()
RescheduleReturnValue Backtrack(ISchedulingData)
int SelectRunnableThread(ISchedulingData)
bool CheckDeadlock(ISchedulingData)
ISchedulingData CheckVisit()
IInstructionExec GetInstructionExec
InteractiveExplorer
defines generic interaction functionality
InteractiveExplorerCLI
defines console menu interaction, I/O and parsing
IBreakPointHandler
bool SetBreakPoint()
bool RemoveBreakPoint()
bool RemoveBreakPoint(int)
bool IsBreakPoint()
BreakPointHandler
...

Figure 4.5: Design of the explorers.

4.3 Explorer

As described previously, the explorer's task is to drive the exploration process, i.e. it runs the exploration algorithm, which involves executing instructions, storing and restoring of the state, and checking for deadlocks.

In a few moments we will be stepping through the description of the exploration algorithm (section 4.3.1) and its rescheduling algorithm (section 4.3.2), which together are the engine of the explorer. But first, we will shortly discuss the class structure of the explorer. The structure was introduced to facilitate easy implementation of multiple exploration "modes".

Please refer to figure 4.5. At the top of the figure, we see the interface IExplorer containing just a Run member function. As the name suggests a call to this function will start the exploration process.

Just below that very simple type is a class named Explorer. Please ignore the member functions for now, they will become clear while advancing through this section. This class implements an automated exploration algorithm, i.e. one that systematically explores all interleavings without any heuristic guidance or user interaction.

Just below that we find InteractiveExplorer, an abstract class that facilitates user guided exploration. That is, at points where the automated explorer would make its own decisions, this class' implementation will ask the user how to continue. To make life easier for the user, a break point handler was added so the user can set certain points where the application asks for user input, while the rest of the process remains automated.

A concrete implementation of this class, with the CLI suffix, is a command-line interface implementation of the interactive explorer. Communication with the user is done via a terminal application. More concrete sub-classes are not implemented, but one could think about implementing a GUI-based implementation.

This completes our introduction to the different types of explorers. For the rest of this section, assume we are talking about the Explorer class, i.e. the automated unguided explorer. Up next is a description of how it explores the state space.

4.3.1 Exploration Algorithm

We will get to the point immediately. Consider the exploration algorithm pseudo code in listing 4.1.

This is the main exploration algorithm without many details that we will fill in later in this document. Let us now take a closer look.

The exploration algorithm is a *do-while* loop that runs until the state space has been fully explored, or a deadlock was encountered. Each iteration in the loop corresponds to either a step *forward*, or a sequence of steps *backward*. Taking a step forward may involve a *rescheduling* and an *execution* phase. Stepping back only involves rescheduling.

Although the rescheduler will be discussed fully in section 4.3.2, it is important to know a bit about its use already, since it heavily influences the exploration algorihm heavily. Indeed most of the code in the algorithm is concerned with when to call the rescheduler, and dealing with the consequences.

The rescheduler will do the following. First, it will *store* the current state of the VM somewhere, and it checks if it has seen it before. If this is the case, it will guide the search backward by restoring a previous state. Finally, the rescheduler may select a different thread to run in the next iteration. The rescheduler can thus seriously mess with the line of execution, especially replace the current VM state with another one, and we need to take this into account.

Back to the pseudo code. Lines 11–16 declare and initialize several (local) variables. The vm variable is a *singleton* value[1], which in our pseudo code means exactly one instance exists which is always accessable. The vm variable holds the current state of the virtual machine (cf. chapter 5).

The only two fields in vm we use in this procedure is current_thread (of class thread), which is a reference to the thread we currently selected to run,

[1]The singleton keyword was chosen because it is the name of the design pattern that is usually applied to implement such a construct.

```
structure reschedule_return_value

    var next_thread      : thread
    var backtrack_count  : integer
    var continue_expl    : boolean

end reschedule_return_value

procedure explore()

  singleton vm      : state

  var do_reschedule : boolean
  var do_execute    : boolean
  var skip_next_res : boolean
  var res_retval    : reschedule_return_value

  do
    if vm.current_thread.is_runnable() then
      do_execute      ← true
      do_reschedule   ← ¬skip_next_res ∧
          ¬vm.current_thread.current_instruction.is_safe()
    else
      do_execute      ← false
      do_reschedule   ← true
      skip_next_res   ← true
      if vm.current_thread.trd_state = running then
        vm.thread_pool.terminate(vm.current_thread)
      fi
    fi

    if do_reschedule then
      res_retval      ← reschedule()
      skip_next_res   ← skip_next_res ∨
          res_retval.backtrack_count > 0
      do_execute      ← do_execute ∧
          res_retval.backtrack_count = 0 ∧
          res_retval.continue_expl
    fi

    if do_execute then
      skip_next_res   ← false
      execute()
    fi

    if do_reschedule then
      vm.current_thread ← res_retval.next_thread
    fi
  od
  while res_retval.continue_expl

end explore
```

Listing 4.1: Main Exploration Loop

and thread_pool which manages all threads. The thread class contains the member function is_runnable, and two fields. Function is_runnable returns true if and only if the thread has an instruction to execute (referenced by its field current_instruction) and its state (field trd_state) is set to *running*. The thread_pool contains a member function terminate, which properly terminates a thread. Its work includes setting its state to *stopped*, the rest is irrelevant here.

The variable res_retval is a tuple (or structure) contains three values that will be returned from a call to reschedule. The definition of this type is given in lines 1–7. The first field contains the thread to run in the next iteration. The next contains the number of backward steps the rescheduler performed. The third field is a flag indicating if we should continue exploration.

The other three variables, i.e. do_reschedule, do_execute and skip_next_res are boolean flags that control the execution of the two phases. The former two are self-explanatory, the latter is a one-shot flag that prevents the next rescheduling phase from taking place. For the moment, please ignore all occurrences of this variable in the pseudo code. We will discuss its use at the end of this section.

The rest of the procedure (lines 18–48) is executed until the condition at line 50 is violated, which is the case if we encountered a deadlock, or we completed exploring the state space.

We start the iteration by checking if the currently selected thread is still runnable. This is for example not the case if the thread has just completed executing all its code (current_instruction points nowhere), or if it is now waiting to enter some critical section (its state is not *running*). If it is, we give a green light for execution (do_execute is set), and we check if we should call the reschedule routine. We call this routine if the current instruction (i.e. the instruction that will be executed this iteration) is *unsafe*. What this means exactly is discussed later, for now please assume some instructions are safe and others are not.

If the currently selected thread is not runnable, reset do_execute so no execution will take place. Also, we want to reschedule this iteration, so we set the do_reschedule flag. If the thread is not properly terminated yet (i.e. its state was not set to *stopped*, do this by calling terminate.

Now consider lines 32–39. If the do_reschedule flag is set, the function reschedule is called. As discusses before, a call to this function can mess up our execution order if it decides to restore an old state, or if it encounters a deadlock. We reset do_execute if any of those conditions are true, since the currently selected thread is not valid any more.

Then there is still the matter of actually executing the instruction (lines 41–44), if the do_execute flag is set. We abstract the execution away for the moment. The bottom line is it updates the VM state.

Finally, we set the current selected thread to the one the rescheduler suggested (if we rescheduled). This will be picked up in the next iteration.

We perform this next iteration only if the rescheduler told us to continue exploration.

This concludes the basic coverage of this part of the exploration algorithm, but we need to address one more thing, and this thing involves the skip_next_res variable we ignore until now.

Skipping the Rescheduler

The problem with the algorithm as described above is the following. If we (intentionally) do not change the VM state between two consecutive calls to reschedule, it concludes it has seen the state before, and will guide the search backward. There are two cases where this is unwanted behavior, and in those cases, we set the skip_next_res flag so the second reschedule call is not executed.

The first case is when a state is restored by the rescheduler. It will have chosen a runnable thread to resume our exploration. Suppose that thread is currently pointing to an unsafe instruction. Because of this, reschedule will be called before a part of the state is changed. Clearly, the state is already in the state store, since it was just restored by the rescheduler. The rescheduler will see this, and direct the search backward.

The other case is somewhat more tricky, and happens when a thread has become no longer executable. It can be the case that a thread has executed all its code (but is still marked as runnable), or it is now waiting for another thread.

Suppose a thread is no longer runnable. We then set do_reschedule so another thread to run will be chosen by the rescheduler. We select another thread, and return to the beginning of the iteration. Suppose this thread starts with an unsafe instruction. We call reschedule again, before execute, and the rescheduler will see the same state it saw one iteration ago, directing the search back.

Setting the skip_next_res flag prevents both these cases, as do_reschedule will be false. Note that the thread selected after one of the above cases will always be runnable, so including the flag in the condition on line 21 is sufficient.

We hereby conclude our discussion of this part of the exploration algorithm, and will describe the rescheduler in full in the next section.

4.3.2 Rescheduler

Let us now discuss the rescheduler we have already heard so much of in section 4.3.1. Consider the pseudo code in listing 4.2.

The function definition on line 53 states the rescheduler will be returning a reschedule_return_value structure, defined in listing 4.1 on page 31 in lines 1–7. The semantics of the fields have already been discussed in section 4.3.1.

```
function reschedule() : reschedule_return_value

  singleton vm  : state
  singleton bt  : stack of (list of thread) × rdata

  var runnable  : list of thread
  var restore   : rdata
  var retval    : reschedule_return_value

  runnable                 ← ∅
  restore                  ← none
  retval.next_thread      ← none
  retval.backtrack_count ← 0
  retval.continue_expl    ← true

  if ¬seen(vm) then
    runnable ← vm.thread_pool.runnable_threads
    restore  ← vm.get_restore_data
    if check_deadlock() then
      return (none, 0, false)
    fi
  fi

  while run.is_empty ∧ ¬bt.is_empty do
    vm.restore_state(restore)
    (runnable, restore) ← bt.pop()
    retval.backtrack_count ← retval.backtrack_count + 1
  od

  retval.continue ← retval.continue ∧ ¬runnable.is_empty

  if ¬runnable.is_empty then
    retval.next_thread ← runnale.dequeue()
    bt.push((runnable, restore))
  fi

  return retval

end reschedule
```

Listing 4.2: Rescheduler

As shortly discussed before, the rescheduler directs the search, which begins with analysis of the current VM state. If the current VM state has already been seen, we will guide the search back to avoid duplicate work, and to short-citcuit cycles in the state space. If the state is new, we do let the exploration continue forward. In either case we select a thread to run next, and pass it back to the exploration algorithm.

The definition of the VM state needs some extension for this algortihm. First of all the tandem get_restore_data and restore_state. The first gets a data structure to restore the current VM to the state it was in when the previous call to get_restore_data was made, i.e. the data to undo what has been done between two consecutive calls. The function restore_state restores the VM to a previous state, using that data.

Additionally, we extend the definition of thread_pool, adding function definitions all_threads and runnable_threads which return a list of all threads and just the runnable threads respectively. Finally, we introduce the singleton bt, which is our backtrack stack. It contains tuples of runnable threads and restore data.

In lines 58–66 three local variables are declared and initialized: runnable will contain a list of (runnable) threads, restore will hold a piece of restore data, and the latter is our return value.

The first thing to do is check if we have seen this state before (lines 68–74). How exactly this is done will remain abstract for now. If the state is found to be new (i.e. not seen before), we get all runnable threads in that state and the data needed to restore the previous state. Then, we check for deadlocks (line 71). If we find one, we will not continue the exploration. The details of deadlock detection will be discussed in section 4.3.3.

The next step is *backtracking*, i.e. restoring a previous state if that is necessary. The system will do this if there are no runnable threads, i.e. if runnable is empty. To revert to a previous state, the algorithm calls restore_state (line 77) with the restore data it has gathered before. It then pops a list of threads and restore data from the stack. This data was put there before, as we shall see shortly. Of course, if the bt stack is empty, we cannot continue doing this.

After the backtracking loop we hopefully have a list of runnable threads in runnable, and some piece of restoration data in res. If runnable contains at least one runnable thread we can continue the exploration process (line 82), and we pick one thread from the list and store it in the return value structure. Line 85 uses dequeue, which takes the first element, but we could also have used another strategy. The resulting list (which may now be empty) is then pushed on bt together with res for a future call to the rescheduler to find (cf. line 78).

All work is done, we are ready to return three values to our caller. This concludes our discussion of the rescheduler.

```
 92 function check_deadlock() : boolean
 93
 94   singleton vm         : state
 95   var       possible : boolean
 96
 97   possible ← false
 98   foreach trd in vm.thread_pool.all_threads do
 99     if trd.is_runnable then
100       return false
101     else
102       possible ← possible ∨ ¬trd.is_terminated
103     fi
104   od
105
106   return possible
107
108 end check_deadlock
```

Listing 4.3: Deadlock Detection

4.3.3 Deadlock Detection and Assertion Violations

At the moment MMC checks for deadlocks and assertion violations while exploring. In this section we will describe how this is implemented.

deadlock

As briefly noted before, a *deadlock* is a situation where there are no runnable processes in a system, but not all processes are terminated, i.e. at least one process is waiting for an event that will never happen. This is an undesirable situation, usually the cause of incorrect locking or synchronization.

A process that is waiting for an event is in a state that in Mono is called WaitSleepJoin. The name is a aggregation of a state in which the process is waiting to acquire a lock, sleeping for a certain amount of time, or joining with another thread. We cover the former and latter cases in section 5.5 and 5.6 respectively. Sleeping is currently not supported by MMC.

Consider the pseudo code found in listing 4.3. It defines a function check_deadlock that is called in the rescheduler at line 71. This is a very basic check, that detects a deadlock only when the system is already deadlocked. It iterates over all threads: if there is one runnable thread, the system is not deadlocked (line 100), since that thread can be run. Else, it is only deadlocked if not all threads have been properly terminated (line 102), which means there is at least one waiting thread.

MMC is usually able to detects "deadlocks" earier, i.e. when the real cause of the eventual deadlock happens. In this situation the system is not yet deadlocked, but is already in a state where at least one process will be in state WaitSleepJoin forever. This check is done when a thread is attempting to acquire a *lock*, and when *synchronizing* threads. As noted above, this will be covered in sections 5.5 and 5.6.

Next is the check for *assertion* violations. An assertion is a statement that is entered explicitly inside the code by the developer to check a certain property holds at that time. MMC checks these conditions like a normal runtime would, but by the nature of MMC the condition is checked for every possible situation, not just one.

assertion

One could consider assertions as a form of code annotation, i.e. the adaption of the source code especially for the purpose of using MMC. However, assertions are a useful tool by themselves, so developers often use them even if the code was never intended to be formally checked. The fact that MMC can use them as well only makes them even more useful.

Assertions are checked in Mono by calling one of the static Assert methods defined in the Debug class, which in turn is defined in the System.Diagnostics namespace. Calls to these methods are filtered out by name. The first argument will contain the (evaluted) condition to check, so we only see if this is a true value. Otherwise, we report a violation.

4.4 Instruction Executors

The instruction executors (IEs) are objects responsible for executing the CIL instruction. In figure 4.4 on page 28 we see the IEs are created and dispatched (started) by the explorer, and they query and update the VM (active) state. In listing 4.1, the IEs play a role in the is_safe and execute functions.

There are many IEs: one for each type of CIL instruction. Each IE is in turn implemented in its own class. This approach closely resembles the *command* design pattern discussed in e.g. [16].

Since there are many different instructions, some hierarchical ordering is applied. Let us look at figure 4.6 for an (partial) overview of what this looks like.

The lowest level of depicted classes (i.e. Call, CallVirt, LdC, and so on) implement code to execute the instruction on an active state. One level above that we find classes defining the functional groups of instruction executors, for example call instructions, load instructions, or (not depicted) store, arithmetic and object-model instruction groups. A group class defines common functionality used by its children, i.e. the executors. Finally, at the top we see the InstructionExec class. This class defines functionality common to all instruction groups.

InstructionExec

There are several advantages as well as disadvantages to the applied pattern, most of which are described in the literature. We will motivate the application of the pattern for this project.

The merit of the command pattern in the MMC is that code to execute CIL-instructions can be seen as first-class citizens.

First of all, this allows us to add more CIL-instructions to the program

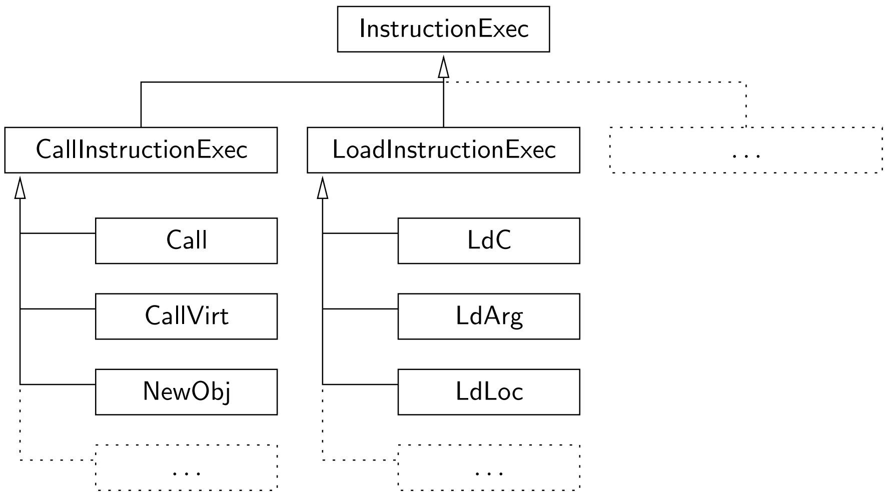


Figure 4.6: Instruction executor hierarchy.

Execute

without modifying existing code. All that needs to be done is define a class with the same name as the instruction (in the correct name-space), and overload the Execute method in that class. Reflection will automatically find the class and use its code. Additionally, caching of executors is easily done.

Finally, meta-data that is associated with the instructions can be added to the executor classes, thereby eliminating the need for big look-up structures. An example of such meta-data is the list of exceptions an instruction can throw, or instruction safety, as will be discussed in section 4.4.1.

The disadvantages of the chosen approach is somewhat larger code (e.g. an operation as simple as a multiplication still needs a full class definition), as well as some run-time overhead. Also, there is less room for “smart” optimizations, e.g. skipping parts of the instruction interpretation that are known to be unnecessary.

4.4.1 Safe and Unsafe Instructions

As described in the previous section, meta-data can be associated with each instruction (executor). An important part of this meta-data is instruction safeness. The concept of safe and unsafe instructions is used in the exploration loop in listing 4.1, in the is_safe function.

The rescheduler is only called if an instruction to be executed is *unsafe*. Unsafe instructions are scheduling dependant, whereas safe instructions are not. Scheduling dependant can in turn be interpreted as: the order in which the threads were (and are going to be) executed is important for this instruction.

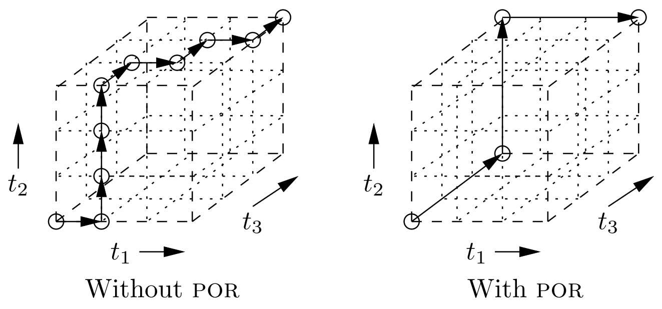


Figure 4.7: Example State-Space with and without Partial Order Reduction.

Take for example a `ldc` instruction. This instruction is safe. It loads some constant on the local evaluation stack. For other threads, it is not visible if this instruction is executed or not. And in this case invisibility means it is not relevant. So we do not need to carefully check all possible orders of execution around this instruction.

Now, take another example, the `ldsfld`, which loads the value of some static field on the local evaluation stack. This is an unsafe instruction. Static fields are shared among all threads, so some other thread can write a different value to the static field. In this case, it is important to know what the exact order of execution is, so we run the scheduler, and systematically check all possible behaviors.

The *merging* of safe instructions into one transaction can be considered a mild form of partial order reduction (POR) [17]. Spin employs a similar technique [25], called statement merging. JPF also uses a form of partial order reduction [40].

In MMC statement merging is not feature, but rather a necessity. In PROMELA (Spin's input language) one statement can do relatively much work, whereas one CIL instruction always does very little. As a result, even a simple program contains many instructions (cf. section 2.4). Without instruction merging the number of states would soon become unmanageable. To illustrate this, consider figure 4.7.

An example state space is shown for a system consisting of three processes t_1, t_2 and t_3, each performing two safe instructions and one unsafe, in that order. One can visualize this state space as a cube, where each of the dimentions represent a process. In this example, an instruction executed by t_1 can be illustrated as taking a step to the right, t_2 by taking a step upward, and t_3 by stepping into the depth. Note that we have not drawn all lines of the $4 \times 4 \times 4$ grid.

On the left-hand side a possible exploration path is shown if no partial order reduction is used, i.e. if each of the thee instructions each thread must execute calls the rescheduler. Adding one begin state, the number of state in a path equals $3 \times 3 + 1 = 10$, and the total number of state in the state-space is $4^3 = 64$ (cf. the *volume* of the cube).

On the right-hand side, the three instructions are merged into one transaction. Each exploration path visits exactly 4 states; one for the full execution of a thread plus one common begin state. The total number of states equals 8 (cf. the number of *corners* of the cube).

Suppose the general case where t threads each execute $n-1$ safe instructions and one unsafe. Without POR this state-space will contain $(n+1)^t$ states, whereas the same system with POR contains 2^t $(n = 1)$. Even for a relatively small n this is a very significant reduction in the size of the state-space.

4.5 State Storage

Remember the pseudo code of the rescheduler (listing 4.2 on page 34). On line 68 it calls the `seen` function. This function is used to check if we have seen a VM state some time in the past. Behind the scenes this involves rather a lot of work.

Essentially, `seen` simply stores all states it ever needed to check. It has not seen a state before if it is not stored in its memory. However, a big problem is: the states can get really big, and typically, there are many states. Comparing the VM states in their original form is not efficient.

We will combat this problem by employing some compression technique, called recursive indexing. This technique is discussed in section 4.5.1, but first we will shortly illustrate the bigger picture.

Consider the left-hand side of figure 4.8, i.e. the "base" scenario. The right-hand side shows an extension we will discuss later in this document.

The data flow of state storage is linear: the VM state is transformed into a collapsed state by a module called the state collapser. This state collapser employs the algorithm we shall describe shortly. The collapsed state is a compressed version of the original state, suitable for storage. Specifically, it is easier and less expensive to compute the hash value of a collapsed state and to compare it to others. As a last component, we have the state storage. This can be any structure capable of efficiently storing data. In our case, it is an encapsulated hash table.

4.5.1 Recursive Indexing

As noted above, we aim to reduce the size and complexity of states, in order to make them more efficiently storable and comparable. We will do this by applying a compression technique. There are several to choose from. We chose to use a technique called *recursive indexing* or *collapsing*, which is also used by the JPF [30] and Spin [26].

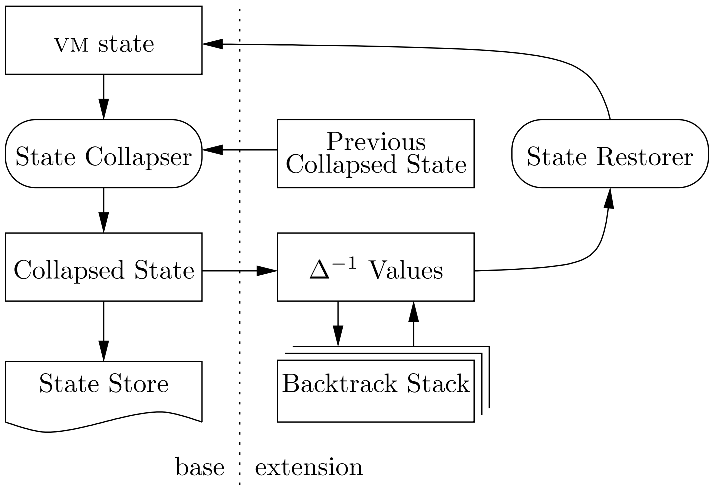


Figure 4.8: State Storage.

Holzmann et al compared several techniques, including several static compression algorithms (run-length encoding, byte masking, Huffman compression), a technique called recursive indexing and two-phase compression [26]. Of the investigated alternatives, recursive indexing was said to give good reductions in state size at relatively modest run-time costs. Visser et al decided to adapt the recursive indexing technique as well [40].

The principle of state collapsing is as follows. There is a data structure called a *pool* that stores objects. Each stored object (O) is assigned a *unique* indexing number by the pool ($\mathcal{P}$). That is, the indexing number of two object is the same if and only if the two objects are the same. Once an object is stored, it is never removed, and once assigned, the index number remains the same.

Assume we have a part of the state that consists of a number of objects in a fixed order: $L_1 = [O_1, O_2, \ldots O_n]$. We store these n object in pool $\mathcal{P}$, and replace all object for their respective index numbers in $\mathcal{P}$, giving us the list of numbers $C_1 = [\mathcal{P}(O_1), \mathcal{P}(O_2), \ldots, \mathcal{P}(O_n)]$. We call this translation *collapsing*. There is no loss of identity, since every object has a unique number.

Now assume we want to compare a second list L_2, to our previous list L_1. We could compare L_1 to L_2 by checking the objects at every index for equality. An alternative approach is to collapse L_2 to C_2 in the same way as we did for L_1, using the *same* pool $\mathcal{P}$, and check if C_1 and C_2 are equal. This will be the case if and only if L_1 and L_2 are equal. Since both collapsed lists consist solely of numbers, comparison is very straightforward and efficient.

An additional and important advantage of collapsing shows when we want to store lists L_1 and L_2. The pool stores each object only once, so if

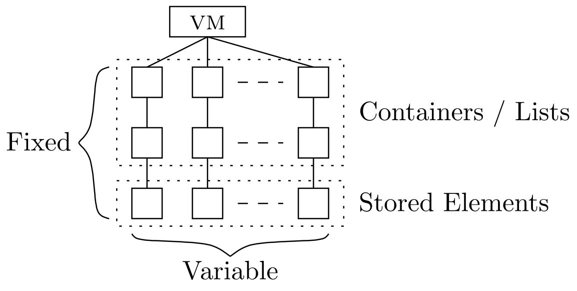


Figure 4.9: Worst case when collapsing the state.

both lists share objects, only one copy is actually stored. Both lists refer to the same object in the pool. This means there is less memory required to remember both lists than when keeping the two original copies.

Of course, we only safe memory when two lists share the same objects. But luckily, this is very common when looking at VM states. Two consecutive states in an exploration path usually differ in only a few parts, so most data is shared.

So far, collapsing looks a lot like traditional hashing, and indeed the hashing is probably being used inside the pool data structure. But, we will take the collapse approach only level further. One that will account for the name *recursive* indexing.

The pool stores objects, so why not store lists of numbers in it as well. This will make the result of a collapse usable in a next collapse. By applying this principle multiple times, we can eventually compress an entire tree into one number. We do not actually do this in MMC, as explained in 4.5.2.

Of course, this all comes at a (computational) price. It is hard to give exact numbers, since we have not yet discussed the precise structure of the VM state. However, the VM state is structured hierarchically. That is, it is a tree-like structure. This tree has a finite and fixed depth. We will call the elements at the bottom *leafs*, and the others *nodes*. We have chosen our collapse algorithm in such a way that each leaf can be collapsed in a constant time.

Suppose the worst case where one leaf is put in one node. This node is again put in another node, and so on. This structure will yield the highest number of nodes for a given amount of leafs. The situation is depicted in figure 4.9.

After collapsing the bottom row, we have paid a run-time cost proportional to the number of leafs. Now, we collapse all the nodes, which will become lists of numbers. Clearly, each list can be collapsed in a constant amount of time, as it contains a fixed number of elements, i.e. exactly one. Since the depth of the tree is fixed, the number of nodes is proportional to the number of leafs.

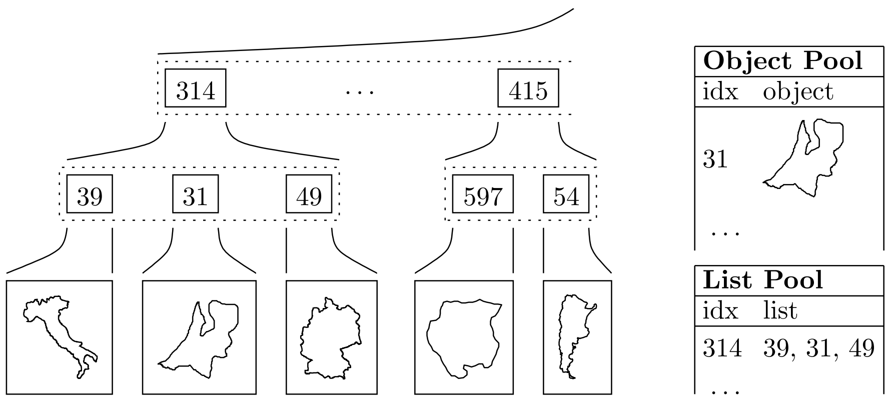


Figure 4.10: The principle of collapsing

Therefore, introducing a level of indirection by storing lists will not make change the time complexity of the collapse algorithm, but it will probably multiply its real run-time by a contant factor.

We will conclude our explanation of collapsing with a small example. Since we have not yet revealed all the details of the VM state, we will use countries and continents as data.

Example 4 (Countries and Continents)**.** Someone is keeping a database containing information about countries. Since there are quite a few countries, they are organized by continent. Consider the figure 4.10. Suppose we want to check if two of those databases are identical. So we apply recursive indexing.

To our left, we see a part of the tree structure organizing the data. The first three objects belong to the same parent container, named "Europe". The last two are members of the group "South America". On the right-hand side we see a small part of two pools: one containing the countries, and one containing lists.

First, we add all countries to the pool. We get a unique index for each of the countries. For continent Europe, this will result in a list $[39, 31, 49]$. This list is then added to the list pool, and gets a unique number 314. The group South America is collapsed in a similar way.

Once all continents have been collapsed, the list can be stored in the list pool, which will give us a number that represents the entire database. Comparing two databases can now be done by simply checking the equality of two numbers. □

One important aspect is still undiscussed. Who decides when an object is an element to be stored in the pool, or when to disassemble it further and store its parts instead? In the previous example,we put the countries in the pool as a whole, but assembly the continents out of their sub-parts (i.e. the countries). Why not simply put the continents in the pool and not bother with the individual countries? Indeed, why not put the entire database in the pool and be done with it?

Or, in a different direction, why not disassemble the data of the countries further, store the sub-parts, and assemble the countries again as a list of collapsed sub-parts, which in turn can be collapsed into a single number?

The answer to these questions is hard to give. Storing too small parts in the pool will greatly increase the percentage of overhead. As an extreme case, storing individual numbers in the pool will gain us nothing. We just substitute one number for the other. On the other side, storing too big parts will kill the advantage we can take of common data. Again, to take an extreme case, suppose we store the entire state (or database) in the pool. If we change just a little part, and then collapse it, we need to store the entire thing again. In this case our compression ratio drops to zero.

Where to draw the line and consider something an elemental object to be put in the pool as a whole is heavily dependant on the data we are working on. In the MMC we chose to use a rather fine-grained collapsing, i.e. we keep the elemental objects relatively small.

4.5.2 Doing it Better

Although the collapse method is quite good at compressing the state there is still an aspect that is not optimal. Every time we call the seen function, we need to run the collapser on the entire current active state. That is, all objects in the state need to be looked up in the pool, lists are constructed, and probably looked up in some pool as well.

This is mostly redundant work. The seen function is typically called quite often, and the as a result, two consecutive state S_1 and S_2 do not differ that much.

A rather straightforward solution presented itself: keep a copy of the previous collapsed state S_1 (a vector containing numbers), and only collapse the parts of S_2 that differ from S_1, i.e. those that have actually changed. The rest of the data can simply be copied from the old vector. The right-hand side of figure 4.8 shows this extension to the data flow, along with several other optimizations we will discuss shortly.

Although conceptually simple, this involves keeping track of all changes in the active state as code is being executed. However, most of the code is concentrated in the container classes, and we can use this feature for another nice optimization which we present in section 4.6. So we added code to keep track of changed values, using a conveniently encapsulated bitmask.

After collapsing a state, we reset the VM state to be clean, preparing it for the next transaction.

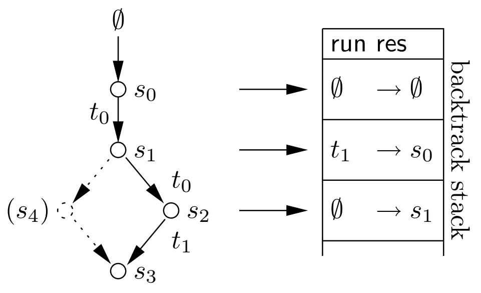


Figure 4.11: Example state space and backtrack data.

4.6 Backtracking

Again, take another look at the rescheduler in listing 4.2, especially lines 76–80. This is the part of the rescheduling algorithm that restores a previous state in order to continue exploration there. The relevant pseudo code elements here are restore_state on line 77 and getting the restore_data in the first place on line 70.

We will give an example of a backtrack stack to give an intuitive idea of how the backtrack stack is constructed. Next, we will discuss what exactly this restore data is.

Consider figure 4.11 for an example state space and its corresponding backtrack stack. Note that the stack grows down here, so the newest element is at the bottom. The restore data that is pushed on the stack is used to restore the previous state, i.e. the state at one level of depth less. So, if we reach state s_2, we store the data needed to return to state s_1.

So what exactly is this data to restore the previous state (e.g. s_1)? We cannot really do the same as Spin, that is, put (the reverse of) the executed operations on the stack. CIL instructions are too fine-grained, and a transaction usually consists of many of these instructions.

An approach we have chosen not to implement is to keep a stack of all instruction executors that we execute in one transaction. An executor is pushed on the stack upon execution, and popped from it when unwinding the transaction. This requires that we implement the reverse of the Execute for each instruction executor.

For several instructions the reverse cannot be executed without storing additional data. For example, we cannot re-create the original two operands of a `mul`tiply instruction (unless those operands are both prime numbers). Storing of this additional data is possible due to our system of instruction executors, although it requires one to be careful when combining this with the caching of instruction executors.

Instead of storing the code that is executed, we have chosen to store snapshots of the state on the stack. As we shall shorly see this allows us to re-use functionality that we already implemented to facilitate state compression.

A naive alternative approach would be to store a copy of s_1 on the stack. This would require making a full copy of the state. When calling restore_state, we can simply assign res (which comes fresh of the stack) to the state variable. This approach requires us to make a full copy of the state. Also, we have to keep the entire state in memory. This is bad as it seems, since this is only the backtrack stack which is expected to be much smaller than the state storage, however it is far from optimal.

A better approach was suggested by Willem Visser et al, which is to store the collapsed state on the backtrack stack rather than the full state, and provide a way to *uncollapse* the state [40]. If we reach a state we need to collapse the state anyway, this would involve no extra work. Additionally, the collapsed state is already stored in the state store, and never modified, so we do not even have to make a copy, we can just keep a reference.

However, one problem remains. One that shows when we are actually using the data, i.e. restoring a collapsed state. If we uncollapse, we have to copy (or clone) the values out of the pool into the active state. So, we end up doing the same amount of work as we did when storing a full copy, we have just postponed the work. The only thing we have gained is that the backtrack stack requires less memory.

We use a technique that exploits work we have done before to optimize the collapse method. It is introduced in the following section.

4.6.1 Building the Delta

In section 4.5.2, we introduced a way to keep track of changes in the active state, to optimize the collapsing of states. We can exploit this also for the purpose of backtracking. Consider figure 4.8 on page 41 again, this time paying attention to the right-hand side especially.

As seen, we extract an object called Δ^{-1} *values* from the collapsed state, and store this on the backtrack stack. This "reverse delta" is a partially collapsed state. It contains the old values of the fields that have been overwritten. From now on, we will simply call this object the *delta*, but be aware that it contains the *old* values.

delta

When backtracking to rebuild the full state, these parts still need to be copied from the pool, but at least we do not need to copy all of the state. The problem remains how to construct the delta in an efficient way, but this problem will prove not to be very hard.

We have implemented a utility number vector (an array that can grow) that remembers its old values, by wrapping two lists of numbers (new and old) into one. This way, we can relatively easy get the delta when we construct the new collapsed state.

For the sake of completeness, (partial) pseudo code for the vector is given

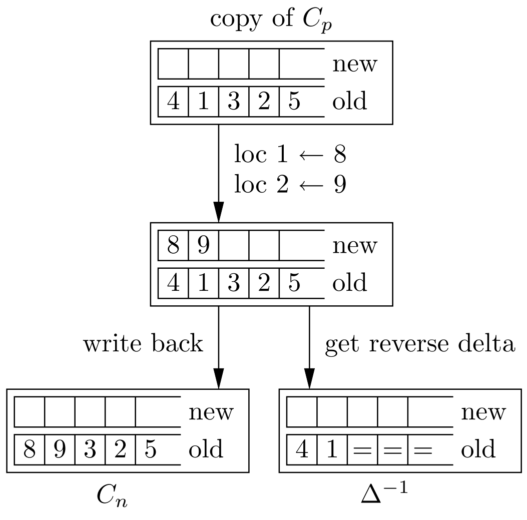


Figure 4.12: Creation of the new collapsed state and delta.

in listing 4.4, although this data structure will probably not be too hard to grasp with the aid of the example given in figure 4.12. We have omitted the (trivial) definitions of get_value, set_value and get_length. The pseudo code should be self-explanatory.

The following is happening in this figure. Function seen has just been called so we need to collapse the current state. To speed this up, we use a copy of the previously collapsed state C_p (cf. section 4.5.2). Analysis of the active state shows two altered parts, which map on locations 1 and 2 of the vector. These are written in the copy of C_p, but instead of overwriting the old values (in old), we keep them in a separate list (new). Once we are done updating, we can create two vectors: the new collapsed state C_n and the delta Δ^{-1}. The former is constructed by overwriting the old values with the new ones. The latter is created by assembling all old values at locations that have a new value assigned.

dirty

The altered parts of the VM state are found by keeping track of changes as we execute code. All values that are written are compared to their old value, and if these two differ the part is marked as *dirty*. This technique is used throughout the state.

For this book-keeping process there is a trade-off between overhead and precision. For example, for a list we could keep a record of all elements that are altered, which requires us to allocate a set of dirty elements, and keep it up to date. Or, as a cheap but less precise alternative, we could just remember whether any element has been altered or not. This requires just one boolean flag, and once one alteration has been done, we no longer need to keep track of additional changes.

```
structure intvector
  var new_values : list of integer
  var old_values : list of integer
end intvector

function get_write_back(vect : intvector) : intvector
  var retval : intvector
  foreach i in 0...get_length(vect) do
    retval.old_values = get_value(vect, i)
  od
  return retval
end write_back

function get_reverse_delta(vect : intvector) : intvector
  var retval : intvector
  foreach i in 0...get_length(vect) do
    if vext.new_value = none then
      retval.old_values[i] = unchanged_value
    else
      retval.old_values[i] = vect.old_values[i]
  od
  return retval
end write_back
```

Listing 4.4: Memento Integer Vector

We have chosen to be precise up to the point where elements are stored as a whole when collapsing the state. That is, for those elements we only know if they are dirty or not. For all parts, we know specifically which elements or sub-part is dirty.

There is one little problem that until now remains unsolved: combining the optimized collapsing technique and backtracking. In the case of a straight sequence of explored states, the C_p is trivially available in the collapser: we just keep a copy of a newly collapsed state before returning it to the explorer. The next time the collapser is called, we use this copy. But, how do we get the C_p if we backtrack to a state?

This problem is easily solved by pushing the collapsed state C_n on the stack in line 86, together with run and res. This requires almost no extra memory, since the state is already stored in the state store. When we backtrack, we overwrite C_p in the state collapser using a copy of the collapsed state we just popped off the stack. This should happen right after backtracking, i.e. just after line 80. Note that we only have to copy the vector, no elements from the pool. Also, we do not use the collapsed state for anything other than resetting the state collapser.

This concludes our discussion and explanation of four parts of the MMC. In the next chapter, we will talk about the last one: the active state.

5 Implementation of the Active State

Looking back at figure 4.4 on page 28, we see the active state in the left box, i.e. it plays a role in the virtual machine function of the MMC. Indeed, the active state holds the current state of that virtual machine.

In this document we sometimes refer to the active state as the VM state or, if no confusion with collapsed states can arise, simply "state". The active state is rather big, and consists of several subcomponents, which will all be discussed in depth in this chapter.

The state is mostly a big collection of data. We will introduce the reader to several algorithms later, but its raison d'être is to *hold the state* of the VM. It is being queried and updated by the instruction executors, and it can be stored in a state storage.

Although at a design level the active state is not too hard to grasp, it is its size and complexity that gives us a hard time managing the structure at run-time. This is where the mentioned algorithms come into play.

The active state can functionally be divided into several parts, which we shall describe in the next section. After that, we will discuss the three most important parts of the active state one by one.

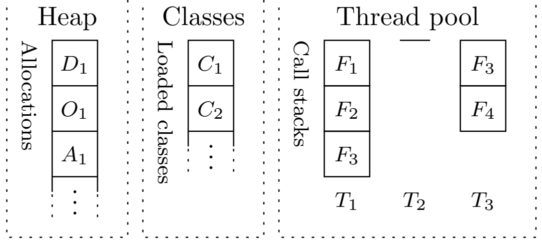


Figure 5.1: Active state of the virtual machine.

5.1 The Aggregation of Parts

Let us consider figure 5.1 for an example of a somewhat randomly assembled active state. Three component (separated by dotted lines) can be identified, i.e. the heap, classes and thread pool. We will describe each of these components briefly. A little while later we shall go into more detail about the design of each individual component.

heap

The *heap* holds dynamic allocations. There are three kinds of allocations: objects, arrays and delegates. These allocations are created at run-time by means of the `newobj` or `newarr` CIL instructions. The number of allocations in the heap can in theory grow infinitely. We shall go into more detail in section 5.3.

classes

So called static fields are stored in the *classes* component. Static fields are data members of a type that are not associated with an allocated object, but instead with the type itself. As a result exactly one exists in a run-time environment. The class storage, or static area, is described in section 5.4.

thread pool

The *thread pool* contains the concurrent processes that have been spawned, and a stack of methods called by that thread. The thread pool is explained in section 5.6.

Before going into more detail about the above components, let us first introduce the data element, a data structure used throughout the VM.

5.2 Data Elements

The most common data structure used in the VM is the data element. All integer and floating point numbers, references, pointers, and strings are data elements. Take a look at figure 5.2 to see a hierarchical overview of these data elements. Not all types have been depicted, notably the run-time handles (pointers to a method or type).

First we look at the top interface IDataElement. It contains declarations for common comparison and conversion methods, as well as a property called WrapperName. This property specifies which class in the class library to use for boxing (wrapping) the data element. For example, an plain old int value is wrapped as a System.Int32 object.

IDataElement has several specializations: INumericElement, IReferenceType and IManagedPointer. The first interface declares several arithmetic and conversion (e.g. integer to float) methods. The second is a common interface to all references to heap objects, and declares a property Location, which is a location in the heap (cf. section 5.3). The latter interface, IManagedPointer, is the interface to all managed pointers in MMC. These pointers all point to some data element, which is accessable through the interface.

We shall not describe the rest of the figure in more detail, as the names of the classes are self-explanatory. The exact details of what fields and

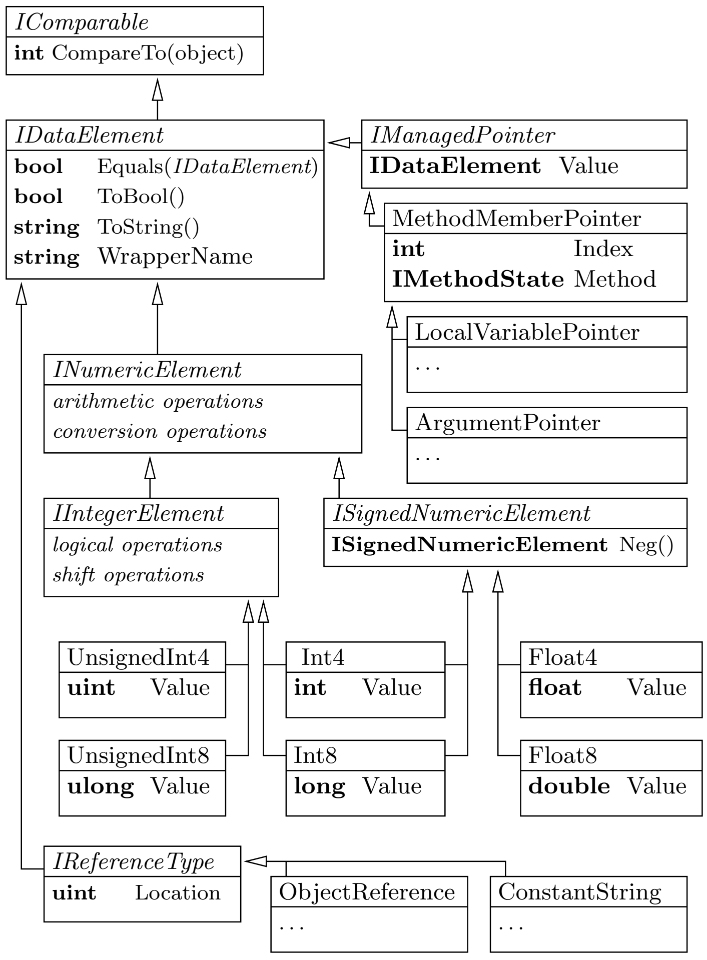


Figure 5.2: Hierarchy of elemental data in the virtual machine.

operation are defined is only of interest to those who plan on working on MMC. For those readers we refer to the code documentation.

The reason for introducing the concept of a data element is mostly a practical one. It wraps value types (cf. section 2.3.1) used by our VM, thereby introducing typing information. Also it allows us to define additional functionality to otherwise primitive types. Data elements are implemented as `struct`s rather than `class`es to reduce overhead.

Without this typing information, we would be forced to deduce the type of element from the context every time that element is used in the execution of an instruction. For example, when executing an `add` instruction, we would need to check if we are dealing with two integers (long or short, signed or unsigned), or floating point numbers (again, long or short). Now, we can just assume both operands are of the `INumericElement` type, and call the addition method defined for those classes.

Additionally, we defined two common containers of data elements: the data element list and stack. Both implementations are straight-forward, but using them in favor of one of Mono collection classes, or even plain arrays, has the advantage that we can easily add functionality to them. Aside from adjusted comparison and hashing methods, we keep track of changes to the contents of the containers, and support reference counting, a technique explained as a part of the following section.

5.3 Dynamic Allocations: the Heap

The heap is the part of the active state where dynamic allocations, i.e. data structures created at run-time, are stored. It is used often, since many modern (object-oriented) programming languages use dynamic allocations (objects) for everything more complex than a simple number, and even those are sometimes modelled as an object.

We shall first describe the design of the heap (section 5.3.1), and then describe two algorithms that play an important role in checking equivalence of two heaps (i.e. garbage collection in section 5.3.2 and allocation placement in section 5.3.3).

5.3.1 Design of the Heap

Consider figure 5.3, it depicts the design of the heap. It may look a bit confusing at first sight, but much of that is caused by abstracting each concrete class using an interface. This interface is commonly named the same as the concrete implementation, with an *I* prefix.

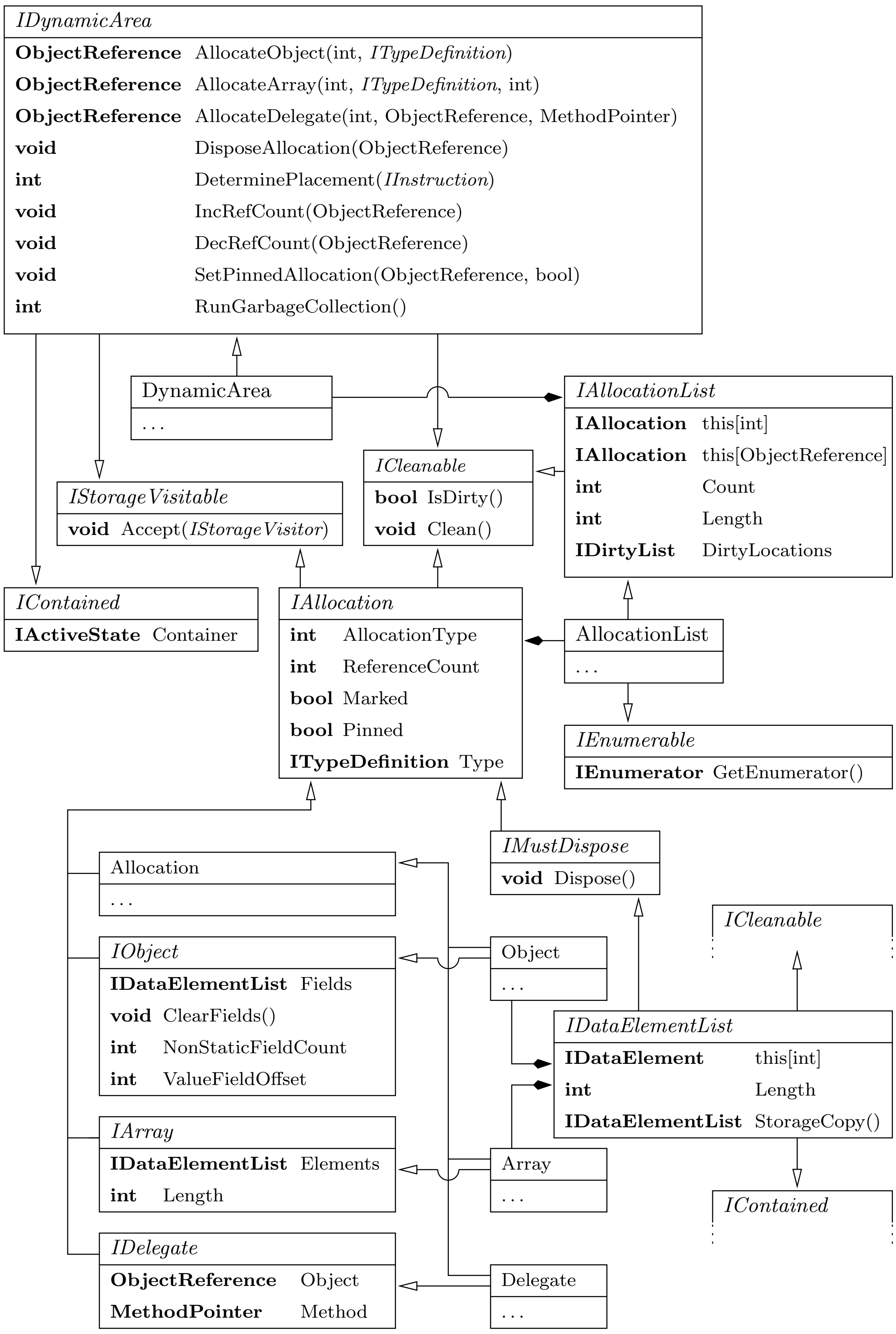
Let us take a very quick tour through the essentially hierarchical structure of the heap in the following paragraph.

The top element is an implementation of the IDynamicArea interface, which owns a list of allocations in a type implementing the IAllocationList. An implementation of this list interface in turn contains zero or more IAllocation objects.

Derived from the IAllocation interface are the IObject, IArray and IDelegate types[1], representing the three types of allocations that live on the heap. Implementations of the IObject and of the IArray types contain a list of data elements in the form of an IDataElementList type.

This summarizes the design of the heap. The last mentioned type, the IDataElementList, may contain an element that references an allocation on the heap. This reference is a normal value type however, and does not make our heap itself a cyclic structure.

[1]The actual class names in the application are AllocatedObject, AllocatedArray and AllocatedDelegate, to prevent mix-up with the same classes in the System namespace.



IDynamicArea
ObjectReference AllocateObject(int, *ITypeDefinition*)
ObjectReference AllocateArray(int, *ITypeDefinition*, int)
ObjectReference AllocateDelegate(int, ObjectReference, MethodPointer)
void DisposeAllocation(ObjectReference)
int DeterminePlacement(*IInstruction*)
void IncRefCount(ObjectReference)
void DecRefCount(ObjectReference)
void SetPinnedAllocation(ObjectReference, bool)
int RunGarbageCollection()
DynamicArea
. . .
IAllocationList
IAllocation this[int]
IAllocation this[ObjectReference]
int Count
int Length
IDirtyList DirtyLocations
ICleanable
bool IsDirty()
void Clean()
IStorageVisitable
void Accept(*IStorageVisitor*)
IContained
IActiveState Container
IAllocation
int AllocationType
int ReferenceCount
bool Marked
bool Pinned
ITypeDefinition Type
AllocationList
. . .
IEnumerable
IEnumerator GetEnumerator()
IMustDispose
void Dispose()
Allocation
. . .
IObject
IDataElementList Fields
void ClearFields()
int NonStaticFieldCount
int ValueFieldOffset
Object
. . .
ICleanable
IDataElementList
IDataElement this[int]
int Length
IDataElementList StorageCopy()
IArray
IDataElementList Elements
int Length
Array
. . .
IContained
IDelegate
ObjectReference Object
MethodPointer Method
Delegate
. . .

Figure 5.3: Design of the heap.

The fact we chose to implement the "heap" as a list may seem confusing because there is also a tree-like data structure called a heap. This has nothing to do with our heap, which emulates a pool of memory used to store allocations. It is sometimes referred to as the free store, mainly by C++ developers.

We will now describe the separate parts of the design one by one in a somewhat slower and more detailed fashion.

The heap type is called IDynamicArea (cf. the JPF), which contains all needed functionality to manage allocations, i.e. determine the place of new allocations, create them, keep track of reference counts, pin and unpin (explained in section 5.3.2) allocations, run garbage collection algorithms and finally, dispose allocations.

The allocations themselves are kept in an IAllocationList type, which is essentially a container class that holds IAllocation types, and nothing more.

The IAllocation type is a top-level type that has four derived types: three to represent the different kinds of allocation as described above, and one abstract type containing common functionality and reasonable defaults.

All allocations contain run-time type information, a constant defining the type of allocation, a reference count, and flags for being marked or pinned down. For explanation of the reference count and flags, see section 5.3.2.

The Object class contains the data of a dynamic allocation created with the `newobj` instruction. In essence, an object is a typed ordered list of data elements (*fields*) stored in a data element list. All meta-data of the fields is stored within the type definition.

Note that possibly not all fields of an object are stored on the heap, only the non-static ones. Static fields are not associated with an object, but rather with a type. We will talk about static data in section 5.4. However, even for the fields that are not stored on the heap, space is reserved. This is done to keep the direct mapping of field number and storage in the data element list.

An Array allocation is a fixed-size list of elements of the same type. Like objects, the elements are stored in a data element list, but unlike objects the length of this list is not known a priori[2] so this data needs to be provided at allocation time. Other than that, arrays behave quite similarly to objects.

The last allocation type, the Delegate supports type-safe function pointers. It can be stored on the heap like an object, and "called" later. The only data contained in a delegate is a reference to the method to call, and reference to the object to call this method on.

[2]Many languages that target the CLI (e.g. C#) support dynamic arrays, so the length is only known at run-time, unlike objects where the number of fields is intrinsic to the type.

5.3.2 Garbage Collection

Looking at the rescheduling algorithm in listing 4.2 on page 34, we see that at line 68 the algorithm needs to check if it has seen some state before. We have already talked quite a lot about this function. Without repeating all that has already been said, seen involves checking if two states and, as a part of those states, two heaps are equal.

Comparing two heaps H_1 and H_2 can be done by checking if a location in both heaps contains the same allocation. Assume the largest occupied location in both heaps is n (if this number is different for the two heaps they are trivially different), $\forall i \in \{0 \ldots n\} H_1(i) = H_2(i)$.

There is a problem here, and that is the rather unforgiving = sign. Suppose at some location i, H_1 is empty, but H_2 contains an unused allocation. For example, an allocation that has been used as a temporary value in a calculation, that is no longer needed or visible anywhere. This allocation is called *garbage.*

Clearly, an empty place and one occupied by garbage are not equal according to the equality sign. For the behavior of the client application however, those places are indeed equal. A way to make the heap comparison better at checking equivalences, we should remove garbage allocations before comparing two heaps. This process is called *garbage collection.*

Note that the concept of collapsing the state, introduced in section 4.5.1 on page 40, is orthogonal to this theory. Storing a state in an efficient way does not solve this problem. Indeed, one of the properties of collapsing is that states do not loose their identity, so if two heaps are considered to be different in their original form, the collapsed version will also be different.

Two separate garbage collection algorithms are implemented in MMC, i.e. reference counting and mark and sweep. We shall explain these two algorithms here. Even before explaining any of them, note that using reference counting is superfluous if mark and sweep is applied, and is only available as a (possibly) faster but less rigorous technique.

Regardless of the garbage collection algorithm, allocations can be *pinned.* Pinned allocations are never deleted from the heap. Pinning is done by simply setting a flag in the allocation. Currently, the only example of objects that are pinned are the Thread objects: the heap object representation of the running processes.

Reference Counting

An allocation is garbage if no reference to it exists anywhere in the active state. The *reference counting* (RC) mechanism is (as the name suggests) based on the idea of keeping track of how many references point to an allocation. If this number reaches zero, the allocation can be removed.

Of course, we do not want to adjust each line of code that touches object references. Instead, we constructed a reference counting variation of the data element container and the call stack classes. These adapted classes behave exactly the same as the ones that do not count the references. A factory class for the containers and call stack was added to assure an instance of the correct type is created.

The RC-enabled data element list keeps track of adding and removal of object references. Upon adding an object reference, the count is incremented, upon deletion it is decremented. To actually update the count, a call is made to the heap. The data element stack works similarly: if an object reference is pushed on the stack, increment the reference count of the object pointed to, and if it is popped, decrement that count.

The pseudo code in listing 5.1 illustrates how a setter in an RC-enabled data element list is implemented. A little note about the code: the type of each of the locations in a data element list is fixed by the compiler of the client application. So if we are assigning an object reference to some location, the old value at that location, if it exists, is also an object reference. Also note that the type of the referenced allocation is irrelevant; we are only interested in the location and its reference count.

```
procedure set_value(lst : list of data_element,
    loc : integer, val : data_element)

  singleton state : state

  if val is object_reference then
    state.heap.inc_ref_count(val)
    if lst[loc] ≠ none then
      state.heap.dec_ref_count(lst[loc])
    fi
  fi
  lst[loc] ← val

end set_value
```

Listing 5.1: Reference counting in a data element list.

The sole purpose of the RC code in the call stack is to make sure a method frame that is popped off the stack is properly disposed. That is, the data element stack (evaluation) and two lists (locals and arguments) of that frame properly decrease the count of all objects pointed to in those containers.

The heap structure is responsible for updating the reference counts. All allocations with reference count zero can be removed. Ideally this should be done immediately when the reference count reaches zero. An alternative is doing this in one big sweep just before the heap needs to be canonical. This is not quite as ideal.

When performing one sweep to delete allocations we increase its time-complexity. Normally, updating the count requires a constant time to complete, i.e. its complexity is in $O(1)$, but doing this sweep to delete unused allocations makes the time complexity proportional to the number of elements in the heap n, i.e. it is now in $O(n)$. What is worse, if we delete an allocation this way, other reference counts may be updated. This requires that we do the sweep again, a worst case time complexity in $O(n^2)$, as illustrated in example 5.

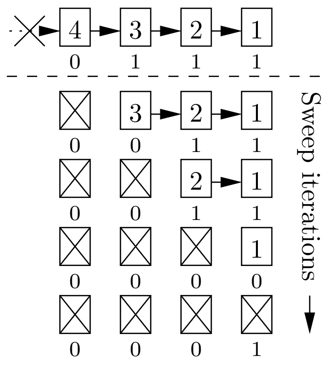


Figure 5.4: RC with a sweep phase is a bad idea.

Example 5 (Removing a linked list using RC)**.** Please consider the situation depicted in the top of figure 5.4. It illustrates a linked list containing four objects. The objects appear in the list in the opposite order in which they appear in the heap, i.e. the object with the highest heap offset is at the head of the list.

We delete the only reference to the head of the list, and are now sweeping the heap using reference counts. The process is shown below the dotted line.

First, we find object 4 with reference count zero after iterating through the entire heap, so we delete it. This updates the reference count of object 3, which becomes zero. We again sweep the entire heap, eventually deleting object 3. This process continues until eventually all four objects are deleted.

So we sweep the heap, which has a time cost proportional to n, and we do this n times, so the time complexity in this worst case scenario is in $O(n^2)$. We would rather not do work in quadric time that can be done in constant time. □

There is one big drawback of reference counting as a way to dispose of unused allocations. That is, it is inapt to delete cyclic structures. Consider the situation illustrated in figure 5.5.

The situation at the left-hand side shows a thread pointing to a linked list of two elements. We assume no other references to any of the objects O_1 and O_2 exist. The number of counted references is displayed below the objects.

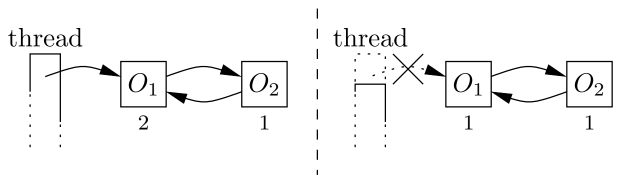


Figure 5.5: RC and cyclic structures do not mix.

Now suppose the thread returns from the method that had the only reference to the head of the linked list. This situation is depicted on the right-hand side. Since we assumed there are no other references, both objects in the list are now inaccessable, and considered garbage. However, they will not be disposed; none of the reference counts equals zero because the objects are still pointing to eachother.

We cannot fix this problem with reference counting, but we can try another way to dispose of our garbage. In the next section we will introduce a technique that is capable of removing such cyclic structures.

Concluding this section, we note that there is one big advantage of reference counting, other than its nice time complexity, which is indeed a strong reason to use it in certain cases. This will become apparent in section 5.3.3.

Mark and Sweep

An alternative garbage collection algorithm that has been implemented in the MMC is *mark and sweep*. This algorithm does not require constant bookkeeping, but instead all the work is done when canonicalising the heap. We shall describe the algorithm using pseudo code in listing 5.2.

In lines 5–11 four local variables are declared and initialized. ref and alloc are just temporary storage. The other two, todo and marked are essential. The former is the all important stack to perform the DFS-like algorithm we will discuss shortly. The latter is a mapping of data elements (more specifically, references to allocations) to a boolean. By default, all references are mapped to false.

The algorithm consists of two phases: the first phase marks all allocations that need to be preserved (lines 13–37), the second deletes (sweeps) all allocations that have not been marked (lines 39–43).

The marking phase can be further divided into two sub-phases: marking the roots (lines 13–24) to bootstrap the marking process, and recursive marking (lines 26–37).

The roots are allocations that are immediately visible to running processes, i.e.: those referenced in the stack frames of running threads (as local variable, arguments or on the evaluation stack), as well as all static fields. Additionally, pinned locations are considered as roots. The roots are not

```
procedure mark_and_sweep()

  singleton state : state

  var todo        : stack of data_element
  var marked      : map data_element to boolean default false
  var ref         : data_element
  var alloc       : allocation

  todo   ← ∅
  marked ← ∅

  foreach trd in state.threads do
    foreach meth in trd.callstack do
      push_refs(meth.locals,      todo)
      push_refs(meth.arguments,   todo)
      push_refs(meth.eval_stack,  todo)
    od
  od
  foreach cls in state.classes do
    push_refs(cls.static_fields, todo)
  od
  push_refs(state.heap.pinned_locations)


  while ¬todo.is_empty do
    ref ← todo.pop()
    if ¬marked[ref] then
      marked[ref] ← true
      alloc ← state.heap.allocations[ref]
      switch alloc.type
        case "object"   : push_refs(alloc.fields,   todo)
        case "array"    : push_refs(alloc.elements, todo)
        case "delegate" : todo.push(alloc.object)
      end switch
    fi
  od

  foreach ref in state.heap.location_references do
    if ¬marked[ref] then
      state.heap.delete(ref)
    fi
  od

end mark_and_sweep

procedure push_refs(lst: list of data_element,
                    todo: stack of data_element)

  foreach elem in lst do
    if elem is object_reference then
      todo.push(elem)
    fi
  od

end mark_refs
```

Listing 5.2: Mark and Sweep

actually marked, but instead pushed on todo, giving the next sub-phase a place to start.

The code in lines 26–37 processes the todo stack, and marks them as reachable. Accessibility is a transitive property, so if an allocated pointed to by a reference on the stack (line 30) has references to other allocations, those are pushed on todo, and will be processed as well in a later iteration.

As a final sweeping step, lines 39–43 delete all unmarked locations.

The design of this algorithm is depicted in figure 5.6. The RecusiveMarker class implements the two phases of the marking procedure. It uses the visitor pattern instead of the switch block found in the pseudo code.

The interface to client classes is just one method which runs both phases consecutively. The recursive marker is a utility class that hides the details of the actual marking process, and is called from the RunGarbageCollection method in the DynamicArea class. This method also initiates the sweep phase, implemented in the heap class, by calling the SweepUnmarked method.

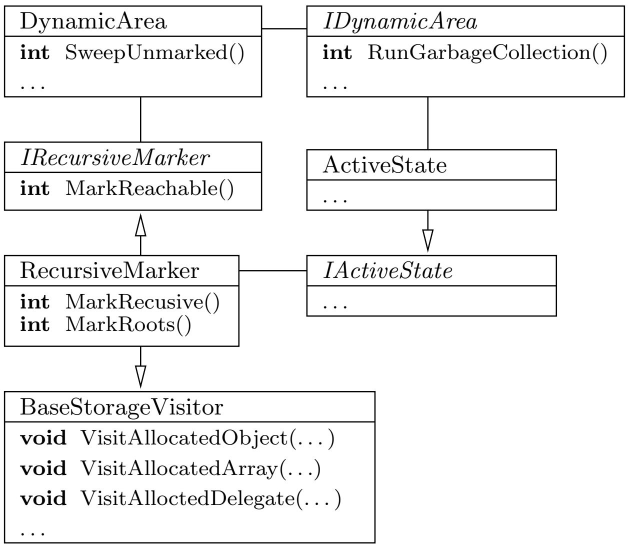


Figure 5.6: Design of the mark and sweep mechanism.

The time complexity of mark and sweep is somewhat higher than that of reference counting. In the first step we mark all roots. The cost of this step is linear in the size of the number of roots r, $O(r)$. Next, we need to iterate through the heap, recursively marking reachable objects. Since we never visit an allocation more than once and each visit takes a constant amount of time, this is linear in the number of allocations n, $O(n)$. In the final sweep, we visit each allocation again, deleting unmarked allocations (in $O(1)$). We also clear the marks on reachable objects, but this is done in the same sweep, and clearly its time-complexity constant and even neglectable.

This gives us total run-time complexity of $O(r + n)$.

In the case of real software, r is likely to be much smaller than n. Typically r is more or less constant, so not even a constant fraction of n. If a heap contains many objects, most of them are likely to be member of a collection or array. It is not very likely that all n objects are referenced by locals, arguments, and so on. We could therefore argue the time-complexity of mark and sweep will be close to linear in n.

5.3.3 Allocation Placement

As mentioned before, the MMC needs to be able to check two heaps for equivalence. Simply comparing the two allocation lists location by location sometimes gives too pessimistic results. In this section we will discuss a technique to make the result more optimistic which is to be used in conjunction garbage collection. At the end of this section we will discuss an alternative approach used by the model checking framework Bogor.

As explained in section 5.3, the heap contains dynamically allocated data. This data is ordered in a fixed way, i.e. allocations have a unique offset in the allocation list. One could see this offset as a pointer where the allocation is put in memory. For a normal execution of the system the offset is irrelevant. References in the CLI differ from pointers in C in that no arithmetic is allowed on the value of the reference (cf. references in Java).

However, in a system like the MMC where we need to compare two states, the offset suddenly becomes relevant, because that is the order in which two heaps are compared. We shall illustrate this using the following example.

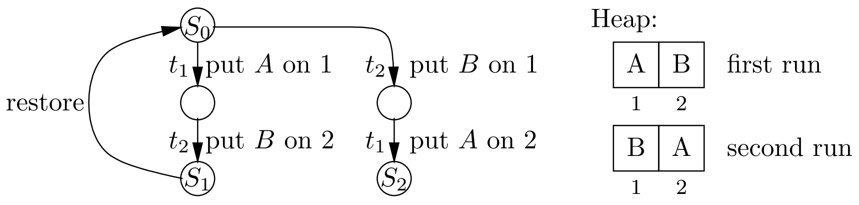


Figure 5.7: Different states after different execution order.

Example 6 (Two threads, two allocations)**.** Consider the state space depicted in figure 5.7. The system starts in state S_0, where the explorer can run two threads t_1 and t_2. Assume t_1 (t_2) wants to allocate an object A (B). First, the explorer selects t_1 to run. It allocates its object A on location 1. Next, t_2 is selected to run, and it allocates its object B on location 2. We are now in state S_1.

After that the execution of the system continues, until at some point all further explorations have been completed, and the system is restored to S_1, and later to S_0. The explorer now chooses a different order of execution, i.e.

thread t_2 is the first to go. It allocates its allocation B on some free place in the heap, which will be location 1! Next, t_1 is allowed to run, and it puts A on location 2, resulting in state S_2.

States S_1 and S_2 are different. But what exactly is different? The internal value of the reference thread t_1 (t_2) has to its allocation A (B). But this value should be irrelevant, so the two states should not be different. □

We can combat this effect by avoiding the creation of symmetric heaps, as explained in [30]. The same technique is applied in the JPF.

The idea is as follows: the first run (or exploration) of the program defines the place an allocation will be stored in the heap. This place is stored in a mapping, and does not change during the rest of the exploration. If, after backtracking and choosing a different path, the same allocations (in order) are put in the previously defined places.

The prime problem here is to specify a minimal but sufficiently discriminating set of parameters to correctly distinguish between allocations. We call these parameters the key. First of all, we include the instruction object, including a reference to the assembly and an offset (or "line-number"[3]) in that assembly. In the pseudo code this information is contained in the instruction type.

This is not sufficient to distinguish allocations that are executed more than once, e.g. it is in the body of a method that is (recursively) called multiple times, or it is part of a loop. So we add an incrementing counter to the key. That leave the problem of multiple threads executing the same instruction.

We have to discriminate between threads in the key. Without it, i.e. when just including the instruction and a counter, the locations would be determined in a first-come first-serve fashion. This will result in different states if a different order of thread execution is chosen, such as illustrated in example 6. So, we need the thread ID as a final parameter.

Consider the pseudo code in listing 5.3. The key discussed above is defined in lines 1–7. We define plmap as a persistent mapping of placement_key to a number (line 12). This value is not part of the state, i.e. it is not stored and restored during the exploration. In line 16–18 we initialize the key tp identify this allocation. The counter field is initially set to zero.

We now try to find a suitable place for this allocation to put (lines 20–30), incrementing the counter as long as no suitable place is found. We first check plmap for an existing mapping of our current key, assigning the result to place (line 21). If such a mapping does not exist (line 22), we will create one. This is done by querying the heap object for a place that is still unused

[3]Note that we are talking about CIL instructions here, not the source code of some high-level programming language, in which it is often possible to write multiple allocations on the same line.

```
structure placement_key

  var thread_id : integer
  var line      : instruction
  var counter   : integer

end placement_key

function determine_place() : integer

  singleton vm    : state
  singleton plmap : placement_key to integer default none
  var key         : placement_key
  var place       : integer

  key.counter   ← 0
  key.thread_id ← vm.current_thread.id
  key.line   r  ← vm.current_thread.current_instruction

  do
    place ← plmap[key]
    if place = none then
      place       ← vm.heap.get_new_slot()
      plmap[key]  ← place
    else if ¬vm.heap.slot_is_free(place) then
      place       ← none
      key.counter ← key.counter + 1
    fi
  od
  while place = none

  return place

end determine_place
```

Listing 5.3: Allocation Placement

until now using get_new_slot. This place has never been used before, it is not a place that was once used and was cleaned by the garbage collector.

If we do find a number in the mapping, we check if that location is free (line 25) by using slot_is_free. If it is, we are done, and return place. Else, we increment counter for the next iteration, and reset place so the loop condition holds and we can try again.

Note that the algorithm requires garbage collection to run efficiently. In MMC, by default, the mark and sweep collector is only run before storing the state because of its non-trivial time complexity. However, reference counting goes hand in hand with this algorithm, as unused allocations are freed as soon as they are no longer needed, allowing us to re-use the slot immediately.

Let us now revisit the previous example, only this time, we are armed with the new placement algorithm. After that, we will discuss some related work in this field.

Example 7 (Two threads, two allocations, revisited)**.** Again, the system starts in state S_0, and t_1 is selected to run. It allocates A on the heap, still on location 1, but by doing so, it creates the following entry in plmap: $(1, 1, I_A) \leftrightarrow 1$. Next, t_2 allocates B on location 2, adding $(2, 1, I_B) \leftrightarrow 2$ to the mapping. The resulting state is S_1, nothing different so far.

The system continues execution, and at some time returns to state S_0. This time, t_2 is to run first. Being thread 2 it executes I_B for the first time, and it finds the allocation key is already mapped to some location, namely location 2. So, B is put on location 2. Next, t_1 is selected, which will find a mapping as well, and put A on location 1. The result is state S_1. □

Related Work

JPF

Although the algorithm devised by the JPF team works nicely and is actually implemented as described above, we would like to make a small remark about its explanation in [30]. In section 2.4 (Exploiting Symmetries) the authors refer to an "occurence number", which is the number of times an instruction is executed before. This number is to be incremented after executing an instruction, and decremented upon backtracking.

This is not entirely correct, since the mapping is not stored or restored together with the state. There are several drawback of the approach of storing the occurence number along with the state. Perhaps most importantly, it increases the size of the state, introducing more states that are found to be different while in fact they are equivalent. This is contra-productive.

Secondly, it does not properly handle any of situations described (in C# code) in listing 5.4. First of all, please look at the Loop, which indefinitely allocates an object, overwriting a previous allocation. Each time line 4 is executed, the occurence count is incremented. This will result in a new key, and a new location to put the object every time. The situation where a method is called mulitiple times (cf. method Caller) fails in a similar way.

```
 1 void Loop() {
 2   object o;
 3   while (true)
 4     o = new object();
 5 }
 6 void Caller() {
 7   while (true)
 8     Callee();
 9 }
10 void Callee() {
11   object o = new object();
12 }
13 void Recursive() {
14   object o = new object();
15   Recusive();
16 }
```

Listing 5.4: Infinite Allocations

Provided the garbage collector is run before an allocation placement is determined, or allocations are deleted once their reference count reaches zero, both situations are correctly handled by our algorithm, which is also implemented in JPF.

Notice that both approaches fail to handle infinite recursion, such as given the Recursive method. This is because the object allocated at line 14 can only be garbage collected after returning from the method, which will not happen. However, the infitely growing heap is not the only problem here. This code will also result in an infinitely growing call stack, so it will crash anyway.

Bogor

A different approach to heap symmetry reduction was explored by the Bogor [5] team. Instead of always keeping the heap in a canonical format, as both the JPF and MMC do, they introduce an algorithm to compare states that does not depend explicitly on the chosen locations in the heap.

Specifically, the Bogor team introduced a technique to compare two threads or objects called the k-bounded thread symmetry (kBOTS) algorithm [36]. The algorithm defines a partial order on threads and objects. By sorting the elements of two states using the same criterion, they can be easily compared on by one. Because we are discussing the heap in this section, we will concentrate on the comparison fo two objects. The algorithm uses this functionality to compare two threads.

The heap can be seen as a *directed graph*, where the nodes coincide with the allocations, and the directed edges are formed by one allocation pointing to another via a reference. This graph defines the *shape* of the heap. kBOTS is a DFS-based algorithm working on this shape graph, in a sense similar to our mark and sweep garbage collector which we explained in section 5.3.2.

Put very succinctly, the kBOTS algorithm works as follows. Suppose we want to compare O_a and O_b, living in heaps H_a and H_b respectively (note that H_a and H_b may be the same heap). We start by comparing the type and primitive values (value types) of O_a and O_b. If this data is equal, recursive calls are made to compare the successors of O_a and O_b, one by one. This results in the shape graphs of H_a and H_b being explored simultaneously in a depth-first fashion. To make sure cyclic structures are handled correctly, we mark allocations we visit at the start of each iteration, and return if we visited any of the two nodes before.

The algorithm can be bounded to not recurse more than k times. If a difference is still not found at a depth k, the two allocations are said to be equal. Note that kBOTS always terminates, even without a bound. If we specify $k = \infty$, kBOTS defines a total order.

We did not implement kBOTS in MMC. We feel the algorithm might be useful to detect thread symmetry, which we do not use at all at the moment. Although thread symmetry reduction is very helpful when checking systems consisting of many identical processes (e.g. the dining philosophers problem used as an example in [36]), it is questionable this is a common scenario in real software.

This concludes our discussion of alternative approaches to the allocation placement problem.

5.4 Static Data: Classes

In this section, we will describe how we implemented static data, especially the loading and initialization of that data, in the MMC. Static data is data that is not associated with a specific object on the heap, but instead with a *type*. In object oriented environments, these types are represented by *classes*. For each instance of a program, there exists exactly *one* class for each type.

From a distance, a class looks a lot like a normal object. That is, it has a type, fields and code. There is one important difference however: since the data is not associated with an object instance, it can be used without a reference. As an immediate consequence, the static data is by definition shared between threads.

This little fact makes class loading and initialization a whole lot more complicated than simple object allocation. In case of a dynamic allocation, it is always clear which thread is responsible for the allocation (making space) and initialization (running the constructor), namely the same thread that executes the `newobj` instruction. With classes this is different. We shall discuss this topic next (section 5.4.1). After that, we will discuss shortly how the classes are stored.

```
 1 class StaticInitTest {
 2
 3   static int fst = snd;
 4   static int snd = 2;
 5
 6   public static void Main(string[] args) {
 7
 8     System.Console.WriteLine("fst={0},␣snd={1}", fst, snd);
 9   }
10 }
```

Listing 5.5: Class Loading example.

5.4.1 Class Loading and Initialization

As multiple threads can read or write the data at the same time, access to the data should be properly guarded. This is the task of the programmer of the tested application, and the VES provides support for it in the form of monitor classes. We do not implicitly provide any locking mechanism whatsoever. To the contrary, it is our task to spot errors and report them, so we surely do not want to prevent them by introducing mechanisms the normal virtual machine does not have.

There is one important exception to this: the initialization of the class, which is implicitly done before the first access to a static variable. The code to do this is contained in a special method called the *class constructor* (cctor). It exists if at least one static member should be initially set to a value other that zero or null.

Typically, a class is loaded upon first access, i.e. a load (ldsfld) or store (stsfld) instruction. After loading, the same thread is responsible for running the class constructor to set the correct initial values. It is not too hard to see a different thread accessing the same static fields while the class constructor is still being executed can easily yield incorrect results[4].

Suppose we implement a naive locking mechanism that states all accesses to the data of a class is blocked while some thread is still executing the cctor. Unfortunately, this introduces a deadlock in certain cases. One of these cases is listed in listing 5.5. Note this is a simplified example that could easily be fixed, but it illustrates the point.

When executing line 8, the data for class StaticInitTest is loaded. This class does have a cctor, and the main thread will be the one responsible for executing it. In the class constructor, the initial value of fst is evaluated, which requires the value of snd. Since the class containing the value of snd is

[4]In fact, this was one of the first bugs found in a small example case: one thread sets a static value to 1 in the class constructor, while the other reads, increments and writes back the same value. There is an interleaving of threads where the resulting static variable will be one instead of the expected 2.

not fully initialized yet, we call its class constructor. This cctor is currently locked, since some thread is already executing it (i.e. the same main thread). By the locking mechanism we introduced, the main thread is now forced to wait for itself, which results in a deadlock.

So this approach will not do. Now what if we relax our lock a little, and do not block a thread if it is that thread that is already executing the cctor. Clearly, this is also incorrect, as it will result in the cctor being recursively called ad infinitum.

A last option is to simply allow the thread that is executing the cctor to access all static data. If this data is already initialized (by itself) the thread is in luch. If it is not, it will have to use the uninitialized data. This is not a perfect situation, but from a stability point of view it is better than a deadlock or a crash (from infinite recursion).

The ECMA standard actually specifies we should use this last option. There is probably a good reason why the folks at Microsoft chose to do it like that. We would have preferred a severe crash, preferably with an error message what went wrong. The code in listing 5.5 thus prints `fst=0, snd=2`.

Pseudo code for the class loading algorithm used in the MMC is given in listing 5.6. The loading procedure starts by checking if a class definition has already been loaded into the class storage (lines 14–20). If not, a new class is constructed, with all its fields set to nil. The class is now loaded, but unless the class does not have a class constructor, it is not yet initialized.

The initialization is done in lines 22–37. We check if there is already a thread initializing the class. If this is not the case, the current thread (me) will do this job, and the class constructor is pushed on the call stack of me.

If there is already a thread initializing the class, the current thread patiently waits (line 33) and is suspended (line 34) for the time being. Unless that would result in a deadlocked situation. Such a situation will arise if we wait for ourselves (checked on line 28) or a thread that is waiting for us (checked in lines 29–31). In this case, the thread is not suspended, and is free to use any field in the class.

Note that after executing the class constructor, the initializing thread will have to awaken all threads that were waiting for it to be initialized.

5.4.2 Class Storage

As said before classes can be referenced without a reference. All they have to identify themselves is their name. The way we store resembles the heap in some sense that we also use a placement mapping, and a linear list to actually store the data. The class store is depicted in figure 5.8.

The first of these structures is a hash table that maps a class name to a unique location. Entries in this mapping are, just like the ones in the mapping used in the heap allocation placement algorithm, never removed.

```
procedure load_class(type: type, thrd: thread)

  singleton vm          : state

  var me                : thread
  var class_store       : map type to class default none
  var cls               : class
  var locker            : thread
  var wait_safe         : boolean

  class_store ← vm.class_store
  me          ← vm.current_thread

  cls ← class_store[type]
  if cls = none then
    cls ← new class
    cls.fields.clear()
    cls.initialized ← ¬type.has_cctor
    class_store[type] = cls
  fi

  if ¬cls.initialized then
    locker ← cls.locking_thread
    if locker = none then
      cls.locking_thread = me
      me.callstack.push(type.cctor)
    else
      wait_safe ← locker ≠ me
      foreach waiting_for_me in me.waiting do
        wait_safe ← wait_safe ∧ locker ¬= waiting_for_me
      od
      if wait_safe then
        cls.waiting.append(me);
        me.suspend();
      fi
    fi
  fi

end load_class
```

Listing 5.6: Class Loading and Initialization

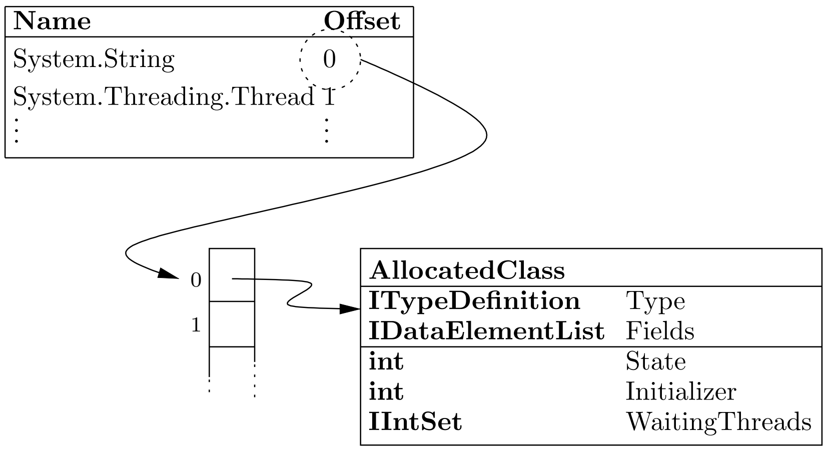


Figure 5.8: Overview of the the class store.

The first run determines the place a class will get in the class store: if no place has already been assigned, a free slot at the end of the list is returned. Else the old assignment is used.

The actual classes are stored in a linear fashion, e.g. in a vector. For each class, we store its fields, both static and non-static ones (again, to keep the one-on-one mapping of the field offset and its place in our data structure), locking information (see section 5.5), and the data needed for class loading (section 5.4.1): whether it is initialized, the thread initializing it, and the threads that are waiting for the class to be fully initialized.

5.5 Locking

In this section we will discuss the locking mechanism the MMC implements. The locking mechanism is implemented inside the Mono run-time environment. It is not CIL code we can simply simulate. Therefore, we implemented this functionality in our virtual machine as well.

The following description deals with *locks*, which are data structures associated with both heap and static elements. Of course, there is only one implementation that is shared by both the dynamic allocations and the classes.

Locking is used in multi-threaded environments to get exclusive access a protected section of code (called the *crtical section*). The process is as follows. First, the thread that wishes to access to the critical section *acquires* a lock *l*. This makes it the *owner* of *l*. While the lock is being held by its owner, no other threads can acquire it. Threads that attempt to do so are suspended and placed in a list called the *wait queue* until *l* becomes available. Suspended threads do not execute code, but instead rest dormant in the background.

When the owner finishes executing the code in the critical section, and no longer needs the lock l, it first send a pulse to one or more suspended threads in the wait queue. This pulse is a heads-up signal that places the thread(s) in a another list called the *ready queue*. Finally, the owner gives up l by *releasing* it. Upon releasing, ownership is immediately transferred to the first thread in the ready queue.

Note that while we talked about one critical section, there can indeed be multiple blocks of code that are protected by the same lock. While one thread is executing code in one of these block, no other thread can enter any of the protected blocks.

The CLI states that the virtual machine should implement a locking mechanism, so programmers are relieved of writing their own locking implementation. This locking mechanism is used via calls to the Monitor class in the System.Threading namespace. Usually, these calls are generated by the compiler. For example, in C# the programmer only has to write a lock statement followed by the block code that is the critical section. In Java a similar keyword exists, called synchronized.

Although the CLI and the compilers make the locking process easier, the concept of locking has proven to be prone to human error. MMC employs checks to help the developer debug these cases.

In the VM state, each object and class has an associated lock. Consider the lock structure (lines 1–8) pseudo code in listing 5.7. This lock is owned by at most one thread, owner. The owner own the lock multiple times, which is stored in count. Furthermore, each allocation has an associated wait and ready queue.

The process of lock acquisition is given in function acquire_lock. It is a quite straith-forward translation of the above description into pseudo code. However, notice line 26. Here, we call a procedure that checks if we have introduced deadlock situation by suspending our currently running thread.

Let the graph $\mathcal{W} = (V, E)$ be the graph where the nodes in N coincide with the threads in the thread pool, and let (t_a, t_b) be an element of E if and only if thread t_a is waiting for t_b. We can now check for deadlocks by checking for cycles in graph $\mathcal{W}$.

The implementation is described in procedure check_deadlock_from. It steps through the graph, starting at the newly waiting thread, which is the only argument to the procedure. It keeps track of the nodes (or threads) it has seen in the seen bit vector (line 42). If it encounters a node twice (checked at line 39), a deadlock is found and a warning message is issued to the user.

Finally, the pseudo code of pulse and release_lock is given in listing 5.8. Accompanied by the informal description both methods in the beginning of this section the code should be self-explanaroty.

In the next section we will discuss the thread pool, the last of the data

```
structure lock

  var owner       : thread
  var count       : integer
  var wait_queue  : list of thread
  var ready_queue : list of thread

end lock

function acquire_lock(o : object) : boolean

  singleton vm : state
  var me       : thread

  me ← vm.current_thread
  if o.lock.owner = me then
    o.lock.count    ← o.lock.count + 1
  else
    if o.lock.owner = none then
      o.lock.owner  ← me
      o.lock.count  ← 1
    else
      o.lock.wait_queue.append(me)
      me.waiting_for ← o.lock.owner
      me.suspend()
      check_deadlock_from(me)
    fi
  fi
  return o.lock.owner = me

end acquire_lock

procedure check_deadlock_from(waiting_thread : thread)

  singleton vm   : state
  var       seen : map thread to boolean default false

  while waiting_thread ≠ none do
      if seen[waiting_thread] then
        error "we introduced a deadlock¬"
      fi
      seen[waiting_thread] ← true
      waiting_thread ← waiting_thread.waiting_for
  od

end chech_deadlock
```

Listing 5.7: Locking 1/2: Acquire and Deadlock Checking

```
procedure pulse(o : object)

  singleton vm : state
  var me       : thread
  me ← vm.current_thread
  if o.lock.owner ≠ me
    error "cannot pulse, not owner"
  else
    if o.lock.wait_queue ≠ ∅ then
      o.lock.ready_queue.enqueue(
          o.lock.wait_queue.dequeue())
    fi
  fi

end pulse

procedure release_lock(o : object)

  singleton vm : state
  var me       : thread

  me ← vm.current_thread
  if o.lock.owner ≠ me
    error "cannot release, not owner"
  else
    o.lock.count ← o.lock.count − 1
    if o.lock.count = 0 then
      if o.lock.ready_queue ≠ ∅ then
        o.lock.count ← 1
        o.lock.owner ← o.lock.ready_queue.dequeue()
        o.lock.owner.waiting_for ← none
        o.lock.owner.awaken()
      else
        o.lock.owner ← none
      fi
    fi
  fi

end release_lock
```

Listing 5.8: Locking 2/2: Releasing and Pulsing

structures used in our virtual machine.

5.6 Thread Pool

The design of the thread pool is given in figure 5.9. The design reflects the hierarchical structure of the thread pool: a higher layer generally contains some meta-data and a container with zero or more lower-level elements.

Specifically, a thread pool contains a list of threads, which have a stack of method states. All the layers have an abstracting interface type.

Aside form being a container for individual threads, the thread pool is responsible for creating and terminating threads, and joining two threads. The latter is directly related to terminating a thread, and is therefore implemented in the same class.

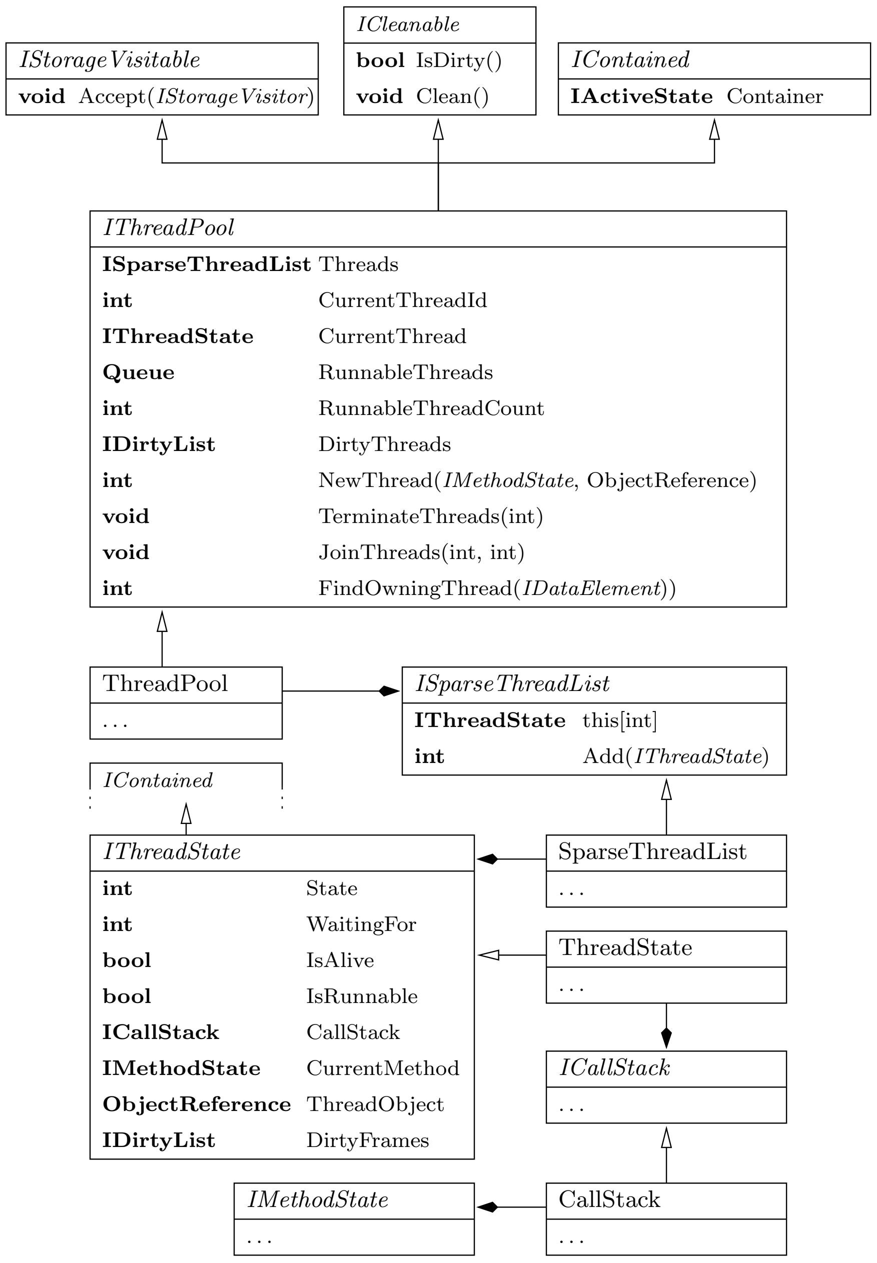
joining

The process of *joining* two threads t_1 and t_2 means the execution of the first thread is blocked until the latter thread is terminated. This may be done if one thread can only continue if the work of another thread has finished.

A nasty situation that may arise when joining threads is that a number of blocking threads is waiting for another blocking thread to terminate. For example, t_1 is waiting for t_2 to finish, which is waiting for t_1 to terminate. The two threads will never continue, and if there are no other runnable threads the system is deadlocked.

We can check for this situation by analysis on the wait graph $\mathcal{W}$ we introduced in section 5.5. As a reminder, the nodes of the graph coincide with the threads, and there is an edge from thread t_a to t_b if and only if t_a is waiting for t_b.

MMC checks for potential deadlocks when joining two threads using simple cycle detection in $\mathcal{W}$. The pseudo code for this algorithm is given in listing 5.7 on page 72.



IStorageVisitable
void Accept(*IStorageVisitor*)
ICleanable
bool IsDirty()
void Clean()
IContained
IActiveState Container
IThreadPool
ISparseThreadList Threads
int CurrentThreadId
IThreadState CurrentThread
Queue RunnableThreads
int RunnableThreadCount
IDirtyList DirtyThreads
int NewThread(*IMethodState*, ObjectReference)
void TerminateThreads(int)
void JoinThreads(int, int)
int FindOwningThread(*IDataElement*))
ThreadPool
...
ISparseThreadList
IThreadState this[int]
int Add(*IThreadState*)
IContained
IThreadState
int State
int WaitingFor
bool IsAlive
bool IsRunnable
ICallStack CallStack
IMethodState CurrentMethod
ObjectReference ThreadObject
IDirtyList DirtyFrames
SparseThreadList
...
ThreadState
...
ICallStack
...
IMethodState
...
CallStack
...

Figure 5.9: Design of the thread pool

6
Testing and Benchmarking

In this chapter, we will present experimental measurements of MMC being run on several examples written in C#. To give an impression of the performance of a similar tool, we translated these examples into Java, and run these on JPF. We made sure not to introduce more complexity in the translation of C# to Java. The two languages are very similar, so usually the translation is trivial.

In the rest of this chapter, we will first describe the test setting (section 6.1). Then, we will present some test results, and discuss what lessons can be learned from them (section 6.2).

6.1 Test Setting

The MMC and JPF tests performed for this chapter were run on a single CPU personal computer running Linux on an AMD Thunderbird 700 CPU with 768MB of RAM. We are running Mono and MCS version 1.1.13.6, which is the latest development version at the time of writing. For JPF we use the Sun Java Runtime standard edition, version 1.5.0_07-b03.

Of course, the measurements given in this chapter are to be interpreted relatively. The exact numbers individually do not carry much meaning.

For the automation of the tests, we have used the m4 macro processor. This software is avaiable on most Unices, or can be obtained at [31]. We will not discuss m4 here, but refer the reader to the same website for a full documentation.

Listing 6.1 contain an m4 macro, taken straith from the m4 documentation [31], that allows us to evaluate a body of text multiple times. We will use it to print multiple copies of the same text, substituting an incrementing number for a placeholder. The result m4 prints out is translated as usual by MCS.

```
define(`forloop',
  `pushdef(`$1', `$2')_forloop(`$1', `$2', `$3', `$4')
     popdef(`$1')')
define(`_forloop',
  `$4`'ifelse($1, `$3', ,
  `define(`$1', incr($1))_forloop(`$1', `$2', `$3', `$4')')')
```

Listing 6.1: m4 macro for expanding a loop.

	10 functions			100 functions			
	10	100	1000	1	10	100	1000
Mono	0.095	0.097	0.097	0.13	0.12	0.13	0.15
MMC	0.22	0.51	3.9	0.47	0.79	4.1	36
MMC w/o cache	0.88	5.8	58	0.69	7.1	56	–
JPF	0.29	0.73	2.8	0.35	0.73	2.8	24

Table 6.1: Run-time of power calculator in seconds.

6.2 Test Results

In this section we will present the tests we have done to analyse the performance of MMC. We have only been able to perform two tests, described in sections 6.2.1 and 6.2.2 respectively.

6.2.1 Execution Performance

First of all, we tested how much of the performance of the application was taken away by running it on MMC rather than directly on Mono. We constructed a simple application that calculates bas to the power of exp, found in listing 6.2.

We varied the number of iterations and functions (using m4). The results in table 6.1. Below "10 functions" (FUNCTION_COUNT) are the results of performing 10, 100 and 1000 iterations (ITER_COUNT) of calling CalcPower1, CalcPower2, ... CalcPower10 consecutively. All tests have been run twice, and then averaged. For MMC, only the exploration times are measured.

From the table, we can conclude MMC is many times slower than Mono. This does not come as a surprise, since Mono is specially optimized for speed. It translates the CIL image to native code, which makes it scale much better than MMC. But of course, omparing a model checker tool to a run-time environment, notably the run-time that model checker run on, in terms of speed is not a fair competition.

By default MMC caches the instruction executor for each instruction it executes, thus preventing a new one needs to be allocated for each instruction. We have shown what happens when this functionality is switched off. This example is quite an extreme case in terms of how many times each line

```
define(‘func_name’, ‘CalcPower$1’)

static class PowerCalculator {

forloop(‘N’, 1, FUNCTION_COUNT, ‘
  public static long func_name(N)(long bas, int exp) {

  long result = 1;
    while (exp > 0) {
      while ((exp & 1) == 0) {
        bas = bas * bas;
        exp = exp >> 1;
      }
      result = result * bas;
      exp--;
    }
    return result;
  }
’)
  public static void Main(string[] args) {

    long result = 0;
    for (int i=0; i < ITER_COUNT; ++i) {
forloop(‘N’, 1, FUNCTION_COUNT, ‘
      result = func_name(N)(14, 8);’)
    }
  }
}
```

Listing 6.2: C# code of the power calculator example.

of code is executed, but we even see a speed-up when we only perform one call to CalcPower. This can be explained by the fact that this method itself contains a loop, so we benefit from caching these instruction executors.

JPF performs very similar to MMC on this example. The somewhat faster run-times could be ascribed to Java being a faster run-time environment than Mono. JPF suffers from the same scalability problems as MMC, and run-times are more or less linear to the size of the executed code.

6.2.2 Locking: Dining Philosophers

The problem of the dining philosophers is a traditional example often used to explain deadlocks. The deadlock happens because of incorrect nesting of locking code. The C# code of the example can be found in listing 6.3.

The performance of MMC and JPF when running this example is highly dependant on the scheduling algorithm used. MMC scheduler does not consider fairness at the moment, which gives us very good results on this particular test.

Number of threads	2	3	5	10
MMC	0.35 / 65	0.39 / 37	0.44 / 182	0.70 / 372
MMC w/o sharing	0.34 / 26	0.35 / 106	0.37 / 59	0.46 / 114
JPF	0.84 / 20	1.11 / 64	2.3 / 376	82 / 14204
Number of threads	**12**	**20**	**50**	**80**
MMC	0.90 / 448	1.53 / 752	7.7 / 1892	19 / 3032
MMC w/o sharing	0.50 / 136	0.85 / 224	2.7 / 554	6.4 / 884
JPF	19.2 / 57188	–	–	–

Table 6.2: Run-time and number of states for dining philosophers example.

This is what happens. The main thread is run until it terminates, spawning n threads $t_1 \dots t_n$. Next, t_1 is run until it re-visits a state, which will happen after one iteration, and the first lock has been acquired. Then, t_2 is scheduled, which does the same, i.e. it is rescheduled the moment it has acquired its "left" lock. Eventually, all threads will own one lock. The moment t_1 is scheduled in again, it tries to pick up its right fork, which is taken by t_2, so it waits for t_2 to release the lock. Next, t_2 is run, which has to wait for t_3, and so on. Eventually, t_n will wait for t_1 (which is waiting for t_2) and the system is deadlocked.

Even though the results are not entirely fair, we present them anyway. The reader has been warned that this example turns out to be very well suited for MMC. The results are presented in table 6.2. For each test we varied the number of threads (DINER_COUNT), and measured the run-time of the exploration as well as the number of states. The data is presented as "time in seconds / number of states".

We also measured what happens when we consider all access to the heap safe. That is, we assume no objects are shared. This is a very optimistic setting, but it safe for this example. Not surprisingly, it results in a serious reduction in run-time and the number of states.

We aborted JPF's exploration of a system of 20 philosophers after it demanded 1.3GB or RAM, upsetting our test system. JPF produces backtraces that take several minutes to be printed in a typical terminal application. The presented run-times do not include the time it took to print these backtraces. It may be interesting for the practical use of both MMC and JPF to investigate how to create more succinct error traces.

We investigated how MMC performs when it does not stop when a deadlock is found. Note that these measurements are not comparable to the JPF, since like MMC it does stop if it finds a deadlock. For a system of 3 philosophers, and with sharing disabled it took MMC 29 seconds to explore a state space of 11564 states. We tried a system of 4 philosophers, but aborted the exploration after 23 minutes and more than 220500 states.

```
using System.Threading;
class DiningPhilosopher {
  const int philosopher_count = DINER_COUNT;
  static object[] m_forks;

  class Philosopher {
    int m_left;
    int m_right;
    public void Dine() {
      while (true) {
        lock (m_forks[m_left]) {
          lock (m_forks[m_right]) {
            System.Console.WriteLine("eating...");
          }
        }
      }
    }
    public Philosopher(int id) {
      m_left = id;
      m_right = (id+1) % philosopher_count;
    }
  }

  public static void Main(string[] args) {
    m_forks = new object[philosopher_count];
    Thread[] philosopher_threads =
        new Thread[philosopher_count];
    for (int i=0; i < philosopher_count; ++i) {
      m_forks[i] = new object();
      Philosopher p = new Philosopher(i);
      philosopher_threads[i] =
          new Thread(new ThreadStart(p.Dine));
    }
    for (int i=0; i < philosopher_count; ++i)
      philosopher_threads[i].Start();
  }
}
```

Listing 6.3: C# code of the dining philosophers example.

This already concludes our chapter on experimental results. We feel that far more experimentation is needed to get a clear picture of where MMC stands in terms of performance. However, from the examples we tested, we can conclude MMC's performance is more or less equivalent with that of JPF.

Also, it has become very clear that partial order reduction techniques are a necessity rather than an optimalization for model checking ever to be feasible on real software. We would encourage further development and experimentation of these techniques.

7 Future Work

At the time of writing a decent amount of work has already gone into the MMC. A VM capable of checking for deadlocks and assertion violations has been developed, and several techniques have been applied to cut back the number of generated states, as well as compress both the stored states and the backtracking data.

However, there are still several things that could use improvement. And several things that could really help MMC become a better tool to find bugs have not been implemented at all.

This chapter suggests a couple of ideas for future developers of MMC to focus their attention on. It is to some extend a reflection of our personal view on the project, and where problems and bottle-necks are to be expected.

7.1 Partial Order Reduction

In multi-threaded environments, one of the hardest problems is controlling access to shared data. This is traditionally a field where human developers often fail to get all the detail straight, and model checkers come in handy to check if the system behaves as expected in even the rarest of cases. This is why MMC calls the rescheduler when a thread performs an operation on a non-local variable, introducing a state.

Although this is necessary in the general case, we do not want to state-space to grow more than necessary. Unfortunately, many applications use the operations that get or set a field very often. However, in many cases, there is only one thread having access to (for example) an object, and it is perfectly safe to access it without the introduction of a new state.

Aalysis of the code is needed to see which interleaving can safely be skipped in the exploration, and which are still necessary to check. Holzmann and Peled explored the approach of static code analysis in [27]. Here, identifying statements that are unconditionally safe (cf. safe and unsafe instructions in MMC) yields impressive improvements in both the size of the state space, and run-time of the verification process.

In the JPF, sharing analysis is done in the mark and sweep algorithm. Although both concerns are not strictly related, the first (marking) phase of the garbage collector already checks all threads for object references. While the marking process is executed, its easy to see if an allocation has already been seen by another thread. During the recursive marking phase, all allocations reachable via shared allocation are also marked as being shared.

Although this is pragmatically a nice solution, it is not very elegant from a design viewpoint, and we would have implemented this differently. The idea, however, is probably easier to implement than static code analysis, and is very promising in reducing the generated state space.

Implementing one of the above techniques to detect sharing of allocations should severely reduce the number of times a load or store field instruction is found to be unsafe, thus reducing the number of times the rescheduler needs to be called. This will reduce the number of created states, and in turn allow the MMC to check bigger client applications.

7.2 Detailed Back Traces

At the moment, if the MMC finds a deadlock or assertion violation, a call stack is presented to the user. This is perhaps somewhat too concise. Printing the backtrack stack yields a trace to the bug. It could be sugar-coated in many ways, but tracing the bug back will probably stay tedious.

To allow more detailed, user-readable traces, we need to be able to translate the CIL code back to the original programming language (programming directly in CIL is possible, but is a game best played by compilers), as well as "play back" what happened, printing line numbers as we go. Once the first of these two requirements is implemented, the other should be easy to add.

The Mono C# compiler can output DWARF [12] debugging information. This file can probably be used to get the required data. We currently do not know of any DWARF reader library written in C# or another CLI-language, but there is a library written in C, and bindings are probably easily created.

Many available tools (notably `gdb` and its front-ends) are able read DWARF debugging data, so it might be interesting for MMC to output an error trace in that format. This allows the user to use an existing debugger to step through the program to the error.

7.3 Exception handling

At the moment we do not handle exceptions (raising and handling) at all. Of course, this is a feature that we certainly would like to have, since thrown exceptions may cause behavior the programmer did not forsee.

CIL instructions can throw exceptions, documented in [28]. For example, an `dd` or `convert` operation may throw an overflow exception. The following is a suggestion how to implement this in the MMC.

Each instruction executor declares a method that checks the conditions for throwing an exception. This can easily be done on a per instruction basis because each instructions is defined in a separate class (see section 4.4). If an exception is to be thrown (by an instruction such as `add`, or an explicit `throw` or `rethrow` instruction), an exception object should be allocated on the heap, and a reference to it should be pushed on the evaluation stack of the caller.

Next, the return value of the executor should move the program counter to the handling block in the method. The Cecil (cf. section 2.2) library provides us with a convenient list of exception handlers, so we can easily search for the correct handler. If such a handler is not found, we issue a warning to the user, because this is probably unwanted behavior.

7.4 Data and I/O Abstraction

Currently, MMCcan only check closed systems. That is, user interaction is not supported, as well I/O operation. This is a big handicap, since the larger part of applications perform some kind of I/O.

A possible way to elimiate this limitation is to provide a framework for writing stubs for simulation of I/O. The implementation of a write stub could for example log its arguments, and then return. Simulating a read operation is somewhat harder. The approach implemented in the JPF is to provide a code instrumentation class, called Verify, which is used to non-deterministically choose a value [40].

The following is a suggested implementation of such a class for MMC, based on the implementation of JPF's Verify.

To get an actual non-deterministic value in the instrumentation class, it should instruct the VM like the class library does, i.e. by calling a method that is only declared but not fully defined. These declarations are to be marked as internal call. We then add a handler for this internal call to MMC.

The implementation of a handler that returns a non-deterministically chosen value requires that we adjust our scheduler. We store the state on the backtrack stack, just like we do when we choose a thread to run, and choose the first possible return value for the handler. Later, if we backtrack to this state, we choose the next possible value. We continue this process until all possible values have been explored.

For a boolean this means the execution is branched into two possible continuations, i.e. one where the value is true, and one where it is false. For an integer this is somewhat more troublesome, as trying all 2^{32} possibilities is clearly not feasable.

A nicer option (at least from a formal point of view) is data abstraction, or symbolic execution [1, 29]. In this case, the data is not "real" but rather a placeholder, say x. This placeholder would point back to the place where it was declared, for the purpose of providing useful feedback to the user.

This involves quite a few changes to the MMC. Not in the least of which, it requires changes in the explorer. The following is a sketch of what implications adding data abstraction to the model checker would have.

We already have an object structure of data elements, so we can add abstract versions of the data we want to abstract. For example, we could add an AbstractInt4 as a sub-class of Int4. This abstract version should behave exactly like a normal number. For example, if we add the abstract x to an integer 4, the result should be the abstract $y = x + 4$. This is probably the least of our problems.

Now let us assume the program uses this abstract value y for a conditional branch, for example the check $y > 1$ is made. We should introduce a scheduling point here (i.e. introduce a state where we can backtrack to), and let the explorer explore the path where this condition holds, and the one where it does not. Choosing any of these path (e.g. the case $y > 1$ holds) restricts (or refines) the value of x (in this case, $x > -3$).

Analysis and refining of these abstract values could become an art on its own, although the person adding such functionality to the MMC can probably find help in the work that has already been done in this field, e.g. by using the Omega library [35], which is also used in [1].

Additionally, the explorer needs to be adjusted so it is able to schedule not only based on which thread is to be run, but also which path is chosen when doing a conditional branch based on abstract values. Our example case was quite simple, but this could really become a problem when multiple abstract values are involved.

In this chapter we have given some suggestions on possible improvements for the MMC, ranging from easy ones (exception handling) to very difficult (data abstraction). The current design allows all but the last suggestion to be implemented without causing too much fuss. The latter (data abstraction) would involve a lot of work and will probably open up a whole world of interesting work to be done on its own.

8 Conclusion

We have written a software model checker for Mono (MMC). The approach is heavily based on and inspired by the Java PathFinder, developed by Willem Visser et al. At the moment, MMC is capable of early deadlock detection and checking for assertion violations.

We adopted the concept of implementing a model checking virtual machine (VM) pioneered by JPF, capable of systematically exploring the state space of a software application. The exploration is performed by iteratively executing instructions that are read from the compiled (binary) code. To have full control over this process, we manage all VM structures ourselves.

The exploration is done in a depth-first fashion. To detect cycles in the exploration graph, we store visited states, and compare each new state to all stored ones. Backtracking is done by keeping a stack of the states that form the current path being explored.

To make state storage and comparison more efficient, we implemented a technique called recursive indexing. We optimized this technique by keeping a record of each changed part in the VM. The improvements are two-fold. First, we only collapse the changed parts when storing a state. Second, we only store the changed values on the backtrack stack, thereby reducing the number of structures we need to restore. JPF takes a similar approach, but does not optimize the backtrack as severely.

Even with this technique on board, storing the state after every instruction will result in an infeasible number of states. We implemented a rudimentary form of partial order reduction by only storing the state if we perform an instruction that is observable outside of its thread. That is, if it reads or changes shared data. We call such an instruction unsafe. Instructions that only touch local structures are safe to be executed at any time.

We group one unsafe instruction together with zero or more unsafe ones, and treat this whole set as one instruction, thereby severely reducing the number of stored states. This technique is also applied in JPF, although for this feature, we were inspired by Microsoft's XRT.

The number of states can be further reduced by applying a technique called heap symmetry reduction. The heap is a data structure that hold dynamically allocated objects. It is a linear list of addressable allocations. We aim to keep this list in a canonical form, so symmetric heaps are never created.

The chosen approach is two-fold. First, garbage collection is used to detect unused allocations, which are removed from the VM. We implemented both the mark and sweep and reference counting mechanism. Second, when we create a new allocation on the heap for the first time, we remember where we put it. The next time we create that allocation, we put it in the same place. This way, the order in which the allocations are created yield the same heap. This will result in more identical states, and therefore a smaller state-space.

The heap symmetry reduction we apply is also implemented in the JPF. Further reductions might be possible using the *k*BOTS algorithm implemented in Bogor, but this has not been investigated yet.

We feel that the codebase of MMC is quite readable for people that are not familiar with it. In fact, readability and ease of design has been high on our list of priorities. We have re-factored the source code several times to keep it easy to understand as the amount of functionality grew.

The applied idiom for each task is the same every time, and we tried to keep entanglement of classes to a bare minimum. Most, if not all, classes can be substituted by different implementations. Subsituting new implementations is currently not possible, though. This is an aspect that could use improvement.

As a result, we feel MMC is a useful tool in an academic environment where ease experimentation with different implementations is an important virtue. Specifically, MMC can serve as an exploration engine for new techniques to be tested. We would stronly encourage further research in the field of partial order reduction applied to software, since we feel it is to a large extend these techniques that will make model checking of large and realistic software projects feasible in the future.

Bibliography

[1] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstract subsumption checking. In Antti Valmari, editor, *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 163–181. Springer, 2006.

[2] Thomas Ball and Sriram K. Rajamani. The Slam Project: Debugging System Software via Static Analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.

[3] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logics.* Cambridge University Press, 2002.

[4] Blast website. `http://embedded.eecs.berkeley.edu/blast`, 2006.

[5] Bogor Website. `http://bogor.projects.cis.ksu.edu`, 2006.

[6] Cecil Website. `http://www.mono-project.com/Cecil`, 2006.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking, 2002.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[9] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions inmodel checking. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 147–158, London, UK, 1998. Springer-Verlag.

[10] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[11] Edd Dumbill and Niel M. Bornstein. *Mono: A Developer's Notebook.* O'Reilly, July 2004.

[12] The DWARF Debugging Standard. `http://dwarf.freestandards.org`, 2005.

[13] C# Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, June 2006.

[14] Common Language Infrastructure. http://www.ecma-international.org/publications/standards/Ecma-335.htm, June 2006.

[15] E. Allen Emerson. Automated Temporal Reasoning about Reactive Systems. In *Banff Higher Order Workshop*, pages 41–101, 1995.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[17] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD in computer science, University of Liege, November 1994.

[18] Patrice Godefroid. Exploiting Symmetry when Model-Checking Software. In *FORTE XII / PSTV XIX '99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 257–275, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[19] Wolfgang Grieskamp, Nikolai Tillmann, and Wolfram Schulte. Xrt: Exploring runtime for .net - architecture and applications. *SoftMC 2005*, 2005.

[20] John Hatcliff and Matthew Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. *Lecture Notes in Computer Science*, 2154, January 2001.

[21] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[22] K. Havelund and W. Visser. Program model checking as a new trend, 2002.

[23] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. *Lecture Notes in Computer Science*, 2648:235 – 239, January 2003.

[24] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[25] Gerard J. Holzmann, Patrice Godefroid, and Didier Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proceedings of the twelth international symposium on protocol specification, testing and verification*, pages 349–363. North-Holland, 1992.

[26] Gerard J. Holzmann Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. of the 3th International SPIN Workshop*, 1997.

[27] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[28] ECMA International. Common Language Infrastructure (CLI), Partitions I to VI, 2005. http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[29] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing, 2003.

[30] Flavio Lerda and Willem Visser. Addressing Dynamic Issues of Program Model Checking. *Lecture Notes in Computer Science*, 2057:80–??, 2001.

[31] M4 macro processor, version 1.4.5. `httP;//www.gnu.org/software/m4`, 2006.

[32] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[33] The Mono Project. `http://www.mono-project.com`.

[34] Microsof .Net. `http://www.microsoft.com/net`.

[35] Omega library. `http://www.cs.umd.edu/projects/omega`.

[36] Robby, Matthew B Dwyer, John Hatcliff, and Radu Iosif. Space-Reduction Strategies for Model Checking Dynamic Software. *Electronic Notes in Theoretical Computer Science*, 3, 2003.

[37] Peter Sestoft and Henrik I. Hansen. *C# Precisely*. The MIT Press, 2004.

[38] Elisabeth A. Strunk, M. Anthony Aiello, and John C. Knight . A survey of tools for model checking and model-based development. Technical Report CS-2006-17, University of Virginia, June 2006.

[39] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, 2000.

[40] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engeneering*, 10(2):203–232, April 2003.