Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Specifying and Verifying Static Initialisation in Deductive Program Verifiers

Bachelor Thesis

Patricia Firlejczyk

March 7, 2024

Advisors: João C. M. Pereira, Dr. Marco Eilers,
Prof. Dr. Peter Müller

Department of Computer Science, ETH Zürich

Abstract

Static initialisers are blocks of code that initialise static variables before they are used in the program. They are often used to modularly establish invariants on the static variables regardless of the clients of the module. Static initialisation is prone to errors that are hard to debug, and its implementation varies between different programming languages. Moreover, static initialisers may run at unexpected moments in the execution. In this thesis, we characterise and exemplify the uses of static initialisers in large code collections. We analyse static initialisation in two different settings and present a modular verification methodology for both settings. In the first setting, the time at which static initialisation occurs is only minimally restricted. There, static initialisation fulfils two characteristics that are satisfied by most mainstream programming languages. Because of that, the presented methodology applies to most mainstream programming languages. The second setting is closely related to Java. Our approach presumes the absence of cyclical initialisation dependencies, which, in our experience, are often indicative of latent bugs. We define a modular check to prevent them. To automate the verification, we provide an encoding from an annotated Java program into Viper. Lastly, we evaluate this encoding on some examples.

Acknowledgements

First and foremost, I am deeply thankful to my exceptional supervisors João Pereira and Dr. Marco Eilers for their help and encouragement during my bachelor's thesis. They were always very enthusiastic towards this project, had great ideas during our meetings and patiently answered all my questions. I would also like to thank them for their thorough proofreading, which helped me to improve my thesis. Further, I would like to acknowledge Prof. Dr. Peter Müller for granting me the opportunity to work on this interesting project. Lastly, I am grateful to my boyfriend for providing feedback on my written report, as well as my family and friends for their support during this time.

Contents

Chapter 1

Introduction

Static initialisation refers to the process through which *static variables*, i.e., the variables whose lifetime is the entire program, are initialised. The goal of static initialisation is to set up constants and static variables or execute blocks of code that might be necessary for the proper functioning of the program before these variables or the functionality are used. Some programming languages provide a construct called a *static initialiser*, which contains initialisation logic to perform setup tasks that need to be executed once before the class is used. Static initialisers are present in multiple mainstream programming languages: in Java or C#, static initialisers are used to initialise the static fields of classes, whereas, in Go, static initialisers can be used to initialise the global variables of a package.

In mainstream languages, static initialisers usually have the following two properties guaranteed by the language runtime:

Property 1 Each static initialiser runs to completion before any access to the static variables that are declared in its class, as long as there are no *cyclical initialisation dependencies*. Accesses to its own static variables inside the static initialiser are excluded from that.

Property 2 Each static initialiser runs at most once.

We say that there exists an initialisation dependency of class `A` on `B` if the static initialiser of `A` might trigger the initialisation of `B`. A cyclical initialisation dependency between classes exists if they cyclically depend on each other. We provide an example of such a cyclical initialisation dependency later in this chapter.

Even though most programming languages satisfy the two properties mentioned above, how static initialisation is implemented depends on the programming language or the compiler. In some languages, the exact order of initialisation may even be undefined. This makes it harder for programmers

```
class A {                           class B {
   static char a = 'a';                static char b = 'b';
   static { a = B.b; }                 static { b = A.a; }
}                                   }
```

Listing 1.1: The static initialisers of both classes depend on each other in this Java code, which can lead to reading uninitialised data or even a deadlock in the concurrent case.

to reason about such programs.

There are a lot of examples of problems related to static initialisation. One example from Java is shown in Listing 1.1. In Java, static fields are declared using the keyword `static`, here `A.a` and `B.b`, and the block of code enclosed in curly braces and preceded by the `static` keyword is called a static initialiser. In this example, class `A` accesses a static field declared in class `B` in its static initialiser. If the initialisation of class `B` is not completed before the execution of `a = B.b`, uninitialised data will be read. The same is true in the opposite direction. This is called a cyclical initialisation dependency. As a result, it is unclear which order the static initialisers should be executed in. In Java, in the sequential case, cyclical initialisation dependencies may lead to reading uninitialised data. Thus, Property 2 is violated in their presence.

Another difficulty related to static initialisation is that it is hard to track when static initialisers are running because they may run at unexpected moments in the execution. Since each static initialiser runs at most once, it depends on how the program has progressed so far, if a statement triggers the initialisation of some classes. Additionally, static initialisers often manipulate the global state. So, one has to make sure their side effects don't result in problems for the code that is currently running and that they do not invalidate any assumptions other initialisers make about the state of their static variables. This is exemplified in Listing 1.2. The static variable `A.a` is

```
class A {                           class Main {
   static char a = 'a';                public static void main() {
}                                         f();
class B {                                 print(A.a);
   static { A.a = 'b'; }               }
}                                   }
```

Listing 1.2: The static variable `a` declared in class `A` is modified by the static initialiser in class `B`. We assume that `f()` is a method that does not write to `A.a`. The value of `A.a` in `main()` is determined by whether the execution of `f()` caused `B`'s initialiser to run.

initialised to `'a'` in class `A`. The static initialiser in class `B` overwrites it, so the value of `A.a` depends on whether `B` is initialised. After `B` was initialised, `A` cannot claim any more that `A.a` holds the value `'a'`. The outcome of the program depends on whether the execution of the method `f()` leads to `B` being initialised.

Therefore, methods for reasoning about the correctness of programs that use static initialisation are clearly needed. Currently, there are not many widely accepted solutions to this problem. We present two existing verification approaches in Section 2.5.

The purpose of a static initialiser is usually to get the class into a valid state before it is used. In formal verification, the conditions that must be met for a state to be valid, one would declare as the static class' *invariant*. We are going to present an approach for verifying programs with static initialisers regarding their static class invariants, which ensures the following properties:

- The class invariants are established by the static initialisation code.
- The established invariants are preserved by all program code, including other initialisers.
- An invariant is guaranteed always to hold while another code that relies on it (i.e., that assumes that the invariant holds) is running.
  - An invariant can only be assumed after the class, that establishes it, is initialised.
  - While an invariant is broken, no code can be executed that relies on the validity of this invariant.
- Cyclical initialisation dependencies are ruled out.

In this thesis, we propose an annotation style where users provide a static invariant per class, expressed in a specification language that extends separation logic. To prove that a class is correct against its static invariant, we must first check that the static initialisation code establishes the invariant. We introduce a verification construct, called an `open/close` block, that allows us to assume and potentially temporarily break static class invariants in the code. To assume an invariant, we require that the class is initialised. We need to ensure that through program execution this invariant will never be assumed while it is broken. In our work, we provide a solution to the problem in two different settings. The former is a language in which static initialisers can run concurrently with the main programs at arbitrary points in the execution. This way, its methodology may serve as a model for many different programming languages. The latter setting is strongly based on the Java language and considers only sequential programs. We present a sound verification technique for both settings. A core restriction we impose is that cyclical initialisation dependencies are not allowed; we rule them out modularly using a novel specification construct we call *static levels*.

Thesis structure The structure of this thesis is the following: In Chapter 2, we provide the background knowledge related to this thesis, including how initialisation works in various languages, especially Java. Afterwards, we describe and characterise the applications of static initialisation and provide examples from large, real-world code bases for them in Chapter 3. This helped us find out which restrictions are acceptable in practice. In the following two chapters, Chapters 4 and 5, we present two verification methodologies to reason about static initialisation and the values of static fields at different points in the program. At the end of this chapter, we present a systematic way to automatically verify programs according to the second methodology by encoding them into the Viper language. In Chapter 6, we present some examples encoded into Viper using the methodology from Chapter 5 and we evaluate this approach. Finally, we give a brief conclusion of this work in Chapter 7.

Chapter 2

Background

In this chapter, we provide the background knowledge necessary to understand this thesis. First, we provide an in-depth explanation of Java's approach to static initialisation, with a focus on its initialisation problems. Then, we briefly explain and characterise static initialisation implementations in different programming languages. Afterwards, we explain the concept of verification. Finally, we summarise two existing techniques for reasoning about static initialisation.

2.1 Static initialisation in Java

In Java, a class may contain several static variable definitions and static initialisers. Instead of requiring explicit calls to initialise static variables, in Java, one can place the initialisation logic in a static initialiser. The Java Runtime Environment (JRE) guarantees that a static initialiser is executed at most once and each static initialiser is initialised before the first *active use* of the class it belongs to (unless there are initialisation cycles, this case will be explained later). In Java, a class is considered initialised after all its static variable definition(s) and static initialiser(s) finish executing. Static variables, that are only declared, but no explicit value is assigned to them, are initialised to a *default value* during static initialisation, that is for example `false` for booleans and 0 for numeric types.

Classes are initialised lazily, which means that initialisation happens at runtime just before the first *active use* of that class. Six situations are considered active use, and all will trigger class initialisation if the class has not already been initialised. Examples of active uses are creating a new instance of a class, calling a static method of a class, the initialisation of its subclass or the access (for both reading and writing) of a static field declared by a class. If there are multiple active uses of a class, from Property 2 mentioned in Chapter 1 follows that this class is initialised at most once.

Uses that are not considered active uses are called *passive uses* and do not trigger the class's initialisation [36]. So in Java, if a class is never used, it won't be initialised. The possibly multiple static variable definitions and static initialisers will execute in the same order as they appear in the program, i.e. they will be executed from top to bottom [27].

As mentioned in Chapter 1, Java does not forbid static initialisers of different classes to depend on each other mutually, this means, they can trigger the initialisation of each other. An example of this issue is provided in Listing 1.1. Since the access of `B.b` inside the static initialiser of class `A` is considered an active use of class `B`, the initialisation of class `B` is triggered before its execution. There is a cyclical initialisation dependency between both classes in our example because the static initialiser of `B` also has an active use of `A`. In this example, the cycle exists between classes `A` and `B`. cyclical dependencies, however, may be of arbitrary size.

There are three possible executions of this program:

Execution 1: A single thread initialises both `A` and `B` and the static initialiser of `A` starts before `B`'s.

Execution 2: A single thread initialises both `A` and `B` and the static initialiser of `B` starts before `A`'s.

Execution 3: Classes `A` and `B` are initialised by two threads concurrently.

The Java Language Specification (JLS) specifies that every class `C` has a corresponding initialisation lock `LC`. A thread must acquire this lock before it can initialise class `C`. If thread *T* wants to initialise class `C` but a different thread is already initialising it, *T* will stall and wait until it gets notified that `C` has been fully initialised. In the case that a thread wants to get the lock for a class `C` for which it's currently holding the initialisation lock (this may happen in the presence of cyclical initialisation dependencies or in the static initialiser of class `C`), it will not block. Instead, it will continue with its regular execution (that is, with the statement that has an active use of `C`), even though `C`'s initialisation is not completed. It thus allows the thread to access static variables from a class whose initialisation did not finish [30].

In the first execution of Listing 1.1, a thread enters the static initialiser of class `A`, claims its initialisation lock `LA` and sets the static char `A.a` to `'a'`. Because class `A` has an active use of class `B`, it triggers the initialisation of class `B` before continuing with `A`. As this thread already holds the initialisation lock `LA`, it does not block and even though `A`'s initialisation is not completed, the current value of `A.a`, which is `'a'`, is assigned to `B.b`. The problem in this execution is that the static variable `A.a`, which contains data that has not been fully initialised, is read, i.e., the static initialiser has not completed, and the value stored in the static field `A.a` may be changed by the static initialiser. In the end, both `A.a` and `B.b` hold the value `'a'`.

In the second execution, the opposite happens. Both `A.a` and `B.b` end up holding the value `'b'` at the end of the initialisation. Thus, the outcome depends on the order in which the static initialisers are executed.

In the last execution, the lock `LA` is claimed by one thread and the lock `LB` is claimed by another thread. `A.a` is set to `'a'` and `B.b` is set to `'b'`. However, in both classes the static block cannot be executed, as neither class `A` nor class `B` have finished their initialisation. Both threads wait for the other one to finish; thus, there is a deadlock [13]. A reason why Java does not reject programs with static initialisation dependencies may be that such dependencies are hard to find statically, and without additional input from the user, especially in the presence of subtyping. The authors of a paper called "Initializing Global Objects" [23] argue that cyclical initialisation dependencies are impossible to find modularly, without whole-program analysis.

To summarise, cyclical initialisation dependencies may result in reading uninitialised data in the sequential scenario, while they can lead to a deadlock in the concurrent case. If cyclical initialisation dependencies are forbidden, then static initialisation in Java satisfies both properties mentioned in Chapter 1.

Another problem in Java related to static initialisation is that in the JLS, it is underspecified when static fields in interfaces are initialised. In contrast to static fields in superclasses, they don't have to be initialised before the initialisation of classes that implement them, however, they can. This means the same program can lead to different results under different implementations [13].

2.2 Characterisation of static initialisers

As mentioned in Chapter 1, the implementation of static initialisers can vary significantly between programming languages. In this section, implementations from languages other than Java are described and grouped based on the characterisations they provide.

Lazy and eager initialisation The first characteristic refers to the time when static initialisers are executed and static variables are initialised. There are two common approaches: *lazy initialisation* and *eager initialisation*. In the first approach, static variables are initialised only when the class to which they belong is used for the first time. Statements, which are first uses, trigger the initialisation of the used class. The initialisation of static variables occurs during program execution. A class' static initialiser never executes and its static variables are never initialised if the class is never used as the program is running.

Eager initialisation differs from lazy initialisation in that all static variables are initialised before program execution. As a result, Property 1 which specifies that all static initialisers run before accessing the static variables they initialise is always true, thus at runtime, no checks are required before accessing static variables, which may increase the performance.

Go is a programming language in which global variables are initialised before the main function's first statement. In Go, static initialisers are represented by *init blocks* and can be used to initialise the global variables of a package. For the sake of discussion, a simplified explanation of static initialisation in Go is provided, where we assume that all initialisation expressions for all global variables do not refer to other global variables or invoke methods that do that.

The initialisation in Go happens one package at a time. Across multiple packages, the language specification defines that a package must be initialised before all packages that import it. Inside a package, first, all global variables are declared and assigned the value of their initialisation expression if there is any, otherwise, they are initialised to the default value. After this step is completed, the `init` blocks in all files are executed. Across multiple files, the initialisation order of variables is determined by the order in which the files are compiled. After all variables are initialised, the main method is executed. In Go, all initialisation code is executed exactly once, regardless of whether the variables are used in the program [4].

A problem in Go comes from the fact that `init` blocks from multiple files are allowed to access the same global variable. An example of this problem is explained in Listing 2.1. As previously stated, the order in which static ini-

```go
// file1.go
package main
var X int
func init() {
   X = 5
}
```

```go
// file2.go
package main
func init() {
   X = 3
}
func main() { }
```

Listing 2.1: The init blocks in the files `file1.go` and `file2.go` both write to the global variable `X`. The outcome of this program depends on the order in which the files are passed to the compiler. If the command `go run file2.go file1.go` is used, the init block in `file1.go` is executed last, and after initialisation, `X` will hold value 5. Else, `X` will hold value 3.

tialisers from different files are executed is determined by the command used for compilation. The `init` block from the first file in the command is executed first. If multiple files write to the same variable, its value may depend on the order in which the `init` blocks are executed, so on the used command [4]. The dynamic initialisation in C++ suffers from the same problem, and it is called the *Static initialisation Order Fiasco* [12].

C++, unlike Java or Go, does not have a direct syntax for a static initialiser, but it does have static variables. Like in Go, *non-local static variables* are initialised before the program execution starts. These are the variables defined in the global namespace or class scope, but not within functions. In C++, the initialisation of static variables happens in two stages. If the initial value of a non-local static variable can be evaluated at compile time, this variable is initialised at compile time. This makes the runtime of the program faster. The remaining non-local static variables are zero-initialised at compile time and later dynamically initialised at runtime. Unless otherwise specified, dynamic initialisation of non-local static variables happens before the execution of the main method [5].

C# is an example of a language that uses lazy initialisation for static variables. In C#, *static constructors* are used to initialise static fields, or to perform actions that need to be executed only once. Static fields can also contain their own initialisers. Those initialisers are executed in textual order immediately before static constructors. When an explicit static constructor is declared, static initialisation (i.e., executing the static field initialisers and the static constructor) is triggered by the first access to a static member or the first creation of an instance of this class. When no explicit static constructor is declared, the exact initialisation time of static field initialisers is implementation-dependent. The only guarantee is that static initialisation must occur before any static field of that class is accessed. However, creating an instance of the type does not have to trigger the initialisation in this case [11][2].

In Scala, *singleton objects* are equivalent to classes in Java, except that only one instance of a singleton object can exist. A singleton object can define multiple fields. Scala does not have the direct concept of a static initialiser; instead, each field is bound to its own block of code, its initialiser, which is executed when the field is initialised. An initialiser can also contain code, that is independent of any fields. Scala has two types of fields: mutable fields called *vars* and immutable fields. The immutable fields are additionally divided into ones that are initialised eagerly, called *vals* and the *lazy vals*, which are initialised lazily. Lazy vals are initialised on its first access.

Singleton objects, like lazy vals, are initialised on their first use, i.e., lazily. A first use refers to assigning the singleton object to a variable, calling the object's method or accessing one of its fields. When a singleton object is initialised, all its vars and all non-lazy vals are initialised [19].

In Rust, a program can contain *static items*, which are values whose lifetime is the entire program. Each static item has its static initialiser, i.e. a constant expression. Static initialisers are evaluated at compile time, thus eagerly [9]. Using the library `lazy_init` one can declare immutable variables, which are initialised on their first access [3].

The language Python does not have an explicit static initialiser, however, imports satisfy both static initialisation properties. When the running program reaches an import statement, the entire code in the imported module is run to completion. Imports in Python adhere to Property 2. This means, that even if the program encounters multiple import statements referring to the same module, the module is imported at most once. However, the module from which the program starts running might run up to two times, once in the beginning and once if another imported module imports it (so, in case of a cyclical import dependency). Modules whose import has begun are stored in a cache. Before importing a module, the program first checks the cache to see if it has already been imported. Moreover, the import statement must be executed before a variable or function from the imported module is accessed, otherwise, it results in a `NameError`.

The difference to static initialisers in other languages is that in Python, one must explicitly call the static initialiser by using the import statement. Because of that, the initialisation in Python cannot be classified as lazy or eager [18].

Cyclical initialisation dependencies Another characteristic is whether cyclical initialisation dependencies between static initialisers are rejected. A cyclical dependency between two classes exists if the execution of its initialisers requires the other class to be initialised. In Listing 1.1, a cyclical dependency in Java code is demonstrated. Because given a cyclical initialisation dependency, it is unclear in which order the initialisation should happen, some languages forbid them.

In Go, at compile time, it is checked if the import relation between packages is a partial order, so acyclic. If this is not the case, the program is rejected. Thus, imported packages are always initialised before packages that import them.

The language C# does not forbid cyclical initialisation dependencies between static initialisers, which might lead to reading uninitialised data. Similar to Java, C# uses initialisation locks. In C#, if a thread fails to acquire an initialisation lock held by itself or another thread, to resolve the deadlock it simply returns and continues with its regular execution. As a result, the thread might see an incompletely initialised state in both the sequential and concurrent cases.

In contrast to static constructors, cyclical dependencies between static field initialisers are not allowed. Since static field initialisers are executed in textual order, if one references the contents of a static field defined below, this field has not yet been initialised. Such dependencies are not statically rejected, but an exception is thrown [11].

```
// Main.py
print("in Main.py, start")
import A
print("in Main.py, end")
```

```
// A.py
print("in A.py, start")
import B
print("in A.py, end")
```

```
// B.py
print("in B.py, start")
import A
print("in B.py, end")
```

Listing 2.2: This example shows a cyclical import dependency between two modules.

In Scala, the initialisers of two vals in separate singleton objects can cyclically depend on each other, just like in Java or C#. Again, this might lead to a deadlock or reading uninitialised data [7].

In Python, cyclical imports of modules are not statically rejected. A cyclical import dependency between modules exists if they all cyclically import each other. In Listing 2.2, a cyclical dependency between modules `A.py` and `B.py` is shown. If the program is run from `Main.py`, the output is the following:

```
in Main.py, start
in A.py, start
in B.py, start
in B.py, end
in A.py, end
in Main.py, end
```

The program starts its execution in `Main.py`. On `import A`, it starts executing `A.py`. Inside this file, the import of `B.py` is triggered. However, when `import A` is reached for the second time, due to the cache, the program knows that an import of module `A` has already started, ignores this statement and continues with its execution in `B.py`. However, if one of the files `A.py` or `B.py` would import `Main`, the `Main` module would be executed twice [18].

The example above runs without errors. But if both files `A.py` and `B.py` would define functions and call each other's functions, it would result in an error. As a result, circular dependencies should be avoided in Python as well.

2.3 Verification

Formal verification in programming is a rigorous and mathematical approach to statically prove or disprove the correctness of a program with respect to a formal specification or property [32]. It is usually used to reason if some post-condition Q holds after the execution of a program s (if s terminates) under the assumption that the precondition P holds at the beginning of the execution of the program. The pre-and post-conditions are assertions about the program state. Hoare logic [20] provides a way to reason mathematically about statements of this form, which are often represented as the $\{P\}\, s\, \{Q\}$ Hoare triple. Mathematical axioms and derivation rules are provided that specify which properties can be proven about programs. There are many extensions of Hoare logic that, among other things, allow to reason about parallelism or pointers.

One of these extensions is called *Separation logic* [31]. Separation logic is designed to reason about heap manipulation or concurrent programs. Each heap location is associated with a permission. The program state is extended by a mapping from heap locations to values and a mapping from heap locations to rational numbers in the range $[0, 1]$, which correspond to the amount of permission held. Permissions guard heap accesses by specifying which locations can be accessed by statements or expressions. A field `o.f` can only be written to if the full permission ($p = 1$) to `o.f` is held, and it can only be read from if a non-zero permission is held [15]. Permissions to heap locations are created on object allocation.

2.4 Viper

Verification Infrastructure for Permission-based Reasoning (Viper) [24] is an infrastructure that automates program verification. It includes an intermediate language, also called Viper, which is a sequential, object-based, imperative programming language. At its core, the language contains methods, functions, and specification constructs such as function and method contracts (i.e. preconditions, postconditions and invariants, which are similar to Hoare triples) and predicates. To prove the correctness of a program, Viper verifies the correctness of all methods and functions against their specification.

The language Viper is based on Implicit Dynamic Frames [34], a permission-based program logic similar to Separation Logic. In the Viper language, object creation is done using the `new(...)` statement, which additionally inhales permissions to all fields listed within the parentheses. Fields are declared by the keyword `field` and all objects can contain all fields. Permission to field f of object o is denoted by $\texttt{acc}(o.f, p)$ in Viper. The value $p \in [0, 1] \cup \{\texttt{wildcard}\}$ is optional (if left out, it equals one) and denotes the amount of permission held. If $0 < p < 1$, the permission $\texttt{acc}(o.f, p)$ is called a fractional permission. The `wildcard` keyword denotes an unspecified positive number, which is always strictly smaller than 1 and greater than 0. The expression $\texttt{old[}l\texttt{]}(o.f)$ in Viper refers to the value of $o.f$ on the heap in the program state at label l.

In Viper, an assertion is called self-framing if it contains read permissions to all the locations it reads. For example, the assertion $o.f == 3$ is not self-framing, however, `acc`$(o.f)$ `&&` $o.f == 3$ is, because it contains read permissions to field $o.f$.

Methods in Viper can contain a sequence of statements, and functions can contain a single expression as a body. If no body is provided, they are called abstract. Viper's methods are impure, which means that their execution can modify the program state. Functions, however, are pure and, given the same input, always return the same value. Pre- and post-conditions of methods or functions (which are self-framing assertions) can be specified with the `requires` and `ensures` keywords.

Let A be a Viper assertion that is not necessarily self-framing on its own but in the current program state. The statement `assert` A checks the permission and value properties specified by A. `assume` A assumes the permission and value properties denoted by A. It only executes if the given permission is already held in the current state without modifying the held permissions.

Similar to the `assume` A and `assert` A statements, Viper additionally contains `inhale` A and `exhale` A statements. However, these statements may add to or remove permissions from the program state. The `inhale` A statement is executed by first adding the permissions denoted by A to the program state and then assuming all value constraints in A. For example, `inhale acc`$(o.f)$ `&&` $o.f == 3$ first adds the full permission to $o.f$ into the program state and then assumes $o.f == 3$ to be true. Executing `exhale` A, first asserts all values constraints in A, then asserts that all permissions denoted by A are currently held and finally removes all these permissions. If one of these `assert` statements does not hold, verification fails.

Predicates in Viper are written in the form `predicate P(...)`$\{B\}$, where B, a self-framing assertion, is the predicate body, or they are written without a body as `predicate P(...)`. The latter is called an abstract predicate. In Viper, predicate instances can be held. A predicate instance can be received e.g. by inhaling or folding P.

For a non-abstract predicate P, Viper defines two statements: `unfold` P`()` and `fold` P`()`. If permissions to P are held, `unfold` P`()` is executed by exhaling the predicate instance and inhaling the predicate body. The statement `fold` P`()` exhales the predicate body and then inhales the predicate instance. Let P be a predicate defined as follows: `predicate P(){acc(o.f) && o.f == 3}`. At a program point, where no instance of P is held, the statements `inhale` P`();` `unfold` P`()` are equivalent to directly inhaling `acc`$(o.f)$ `&&` $o.f == 3$. Executing `fold` P`();` `exhale` P`()`, first checks if $o.f == 3$ holds and full permissions to $o.f$ are held in the current program state. If this is the case, permissions to $o.f$ and the predicate instance P are removed.

A detailed description of the language is provided in the Viper tutorial [38].

2.5 Related work

An existing approach providing a modular methodology for specifying and verifying static class invariants in object-oriented programs is described in a paper by K. Rustan M. Leino and Peter Müller [22]. Their methodology supports three major uses of static fields and invariants within the Java library. Leino and Müller propose attaching static class invariants to each class. They define static class invariants as invariants that are described at the class level and enforced by their static initialiser. After a static initialiser is executed, its static class invariant must be established. The authors also propose `expose` blocks, inside which the static class invariants may be temporarily broken, and static ghost fields, that specify if one currently is inside an `expose` block. The static class invariant of `C` may not hold within an `expose C` $\{s\}$ block, but it must be re-established on exit.

Moreover, the authors define a partial order between classes, called validity ordering. The validity ordering is used, among other things, as the initialisation order. It is required to be acyclic. Because of behavioural subtyping, the validity ordering requires that subclasses are initialised before the initialisation of their superclasses. However, this assumption does not hold in languages like Java or C#. So, in the methodology defined in this paper, the static initialiser of a subclass cannot refer to the static fields of the class it extends. Another issue is that this paper does not use separation logic and in general targets a sequential setting. This makes it more difficult to extend this approach to deal with concurrency or heap-allocated data structures.

Another related work is the paper called "Initializing Global Objects: Time and Order" by Fengyun Liu et al [23]. The authors present a modular static analysis to ensure the safe initialisation of global objects. For the static analysis, Fengyun Liu et al. define two small calculi, an abstract domain, and the corresponding declarative rules. In the first calculus, all global variables are immutable, whereas in the subsequent calculus, mutable global variables are supported. The approach proposed in this paper is based on two principles:

- Initialisation of global objects must follow a partial order.
- The state of a global object should be independent of when the global object is initialised.

The first principle disallows cyclical initialisation dependencies. This restriction is enforced by a rule in the static analysis. To enforce the second principle, the authors disallow side effects during the initialisation of global objects. This forbids for example global objects to write to mutable variables in other global objects during initialisation. This approach also has the restriction that during initialisation, global objects are not allowed to read from mutable variables declared in other global objects.

Chapter 3

Uses of static initialisation

The goal of this chapter is to describe how static initialisation is used in practice. Our observations are based on several real-work code bases: the Java Standard library (2018, Oracle) [1], a Minecraft Hack Client code base [35], a Java design patterns code base [37] and the Go Standard library [16]. A special focus lies on the Java Standard library. In the first section, we explain and characterise the different uses of static initialisation that we found. In the following section, we provide and explain some examples of the mentioned uses. The characterisation is based on one in the paper "Modular verification of static class invariants" [22].

3.1 Characterisation of uses of static initialisation

The main purpose of static initialisation is to initialise static variables independently of the clients, such that they do not have to do it themselves. We encountered three main patterns of static initialisation. All of them are explained in a paragraph below.

1. Set up a consistent global state.
2. Initialise a shared pool.
3. Register a callback.

Set up a consistent global state The first application is the most commonly encountered. During static initialisation, a consistent global state is established by initialising the static fields. The consistent global state can be described by static class invariants, which describe what properties hold for the static and instance fields. The code relies on the invariants being maintained during program execution. Furthermore, the code may maintain limitations on how their fields' state changes and this can be captured using history constraints.

Static fields may contain mutable or immutable data. Immutable data is initialised to store shared values. After initialisation, these values are read by other parts of the code. In the code bases we looked at, immutable data was usually of primitive type, and then it is sufficient to declare the fields as `static final` to make them immutable. In the Java Standard library, this pattern can be found, among other things, in the class `Character`, which defines an immutable static `char MIN_VALUE` that holds the smallest value of type `char`, or in the class `Boolean`, where an immutable static field stores a version number. Some classes define even immutable non-primitive static fields. For example, the class `Calendar` defines a read-only set using the `unmodifiableSet()` method from the `java.util.Collections` class. This set stores all available calendar types and is used for exception throwing in the code. Since immutable static fields cannot be modified, the invariants they establish are trivially maintained throughout program execution and can be assumed by everyone in the program.

In the code bases we looked at, mutable static fields are most often used to store mutable data structures, usually sets or maps. The static initialisers set up the static fields to satisfy the static invariant immediately after initialisation. But this is not enough. During program execution, the class must make sure that its static class invariant will never be violated. This is often done by declaring these data structures as private fields and either only accessing them inside this class (example provided in Listing 3.1) or allowing other classes to access them only through public methods (Listing 3.2). These public methods control how others modify the private static fields to make sure that the static class invariant is never violated. The motivation for this additional subdivision is that these public methods enable other classes to establish properties about the class that declares the data structure. In Section 4.2, we will especially look at the usage where other classes access these shared data structures inside their static initialisers and thus their static class invariant relies on the validity of another class. Listing 3.3 exemplifies this pattern.

Initialise a shared pool A shared pool is a data structure (stored as a static field) that holds frequently used items which are re-used during program execution. The goal of this pattern is to reduce computation or memory overhead. This pattern can further be subdivided depending on its primary goal.

The first usage is to store shared objects inside a static data structure to reduce the memory overhead. It is based on the Flyweight design pattern [33]. The data structure is called an object pool in that case. An object pool is a data structure that contains a set of initialised objects kept ready to use, such that they do not have to be allocated and destroyed by clients on demand. It aims to reuse frequently used objects. Instead of creating a new object, the client will request an object from the pool and perform operations on it. Data structures containing object pools are either initialised completely inside the static initialiser, or they are initialised during program execution on the first use of this object. Typically, once an object is added to the pool, the classes defining these data structures make sure that it will never be removed. This pattern can be found, for example, in the Java Standard library, e.g., in the classes `Byte`, `Integer` or `Boolean`. In the next section, an example from the `Byte` class is shown in Listing 3.4.

The second usage of this pattern is to store computationally hard results inside a shared data structure. That way, clients can utilize results from previous computations. This usage reduces the computation overhead, thus improving performance by eliminating redundant computations. In the next section, this pattern is shown in Listing 3.5.

Register callback Registering a callback means passing a function pointer to someone who can call this function in the future. An example that we frequently saw in the Java Standard library is to register a native method. In Java, native methods are methods written in programming languages different from Java. Native methods can be used, for example, to link a library written in another language to some Java code. They can also be used to execute code written in a lower-level language for operations that require high efficiency [29]. Native methods are usually loaded during the class' initialisation inside the static initialiser by calling `System.loadLibrary(`*name*`)` or `registerNatives()`. This pattern can be found for example in the classes `Object`, `System` or `Thread` in Java's Standard library.

3.2 Examples

In this section, we provide examples written in Java of usages of static initialisation corresponding to the previously mentioned patterns. We explain each example briefly and mention what kind of invariant the initialiser establishes. This section includes three examples for the first pattern, two for the second pattern, and one short example for the third pattern.

The first example corresponds to the pattern where a consistent global state with respect to a mutable data structure is established, and the data structure is only used inside the class. This pattern is shown in Listing 3.1 on a very simplified version of the `Finalizer` class from the Java Standard library. In this example, the global state is set up during static initialisation. All class instances assume that the global state is consistent and maintains its consistency.

The class `Finalizer` defines a doubly linked list of `Finalizer` objects, that are awaiting finalization. Each `Finalizer` object contains non-static pointers next and prev, that point to the next and the previous Finalizer objects in the list, respectively. The head of the list is stored as a static field called unfinalized. During static initialisation, unfinalized is initialised to null, because no Finalizer instances exist yet. Moreover, a static final lock is defined, which synchronizes all accesses to the unfinalized list. The invariant of this class is that unfinalized is the head of a doubly linked list of Finalizers that are not yet finalized. Additionally, for a Finalizer f, if f.next == f holds, then f is already finalized.

This invariant is expected and preserved by the constructor and the method runFinalizer(). On object creation, the newly created object f, that is not yet finalized, is set as head of the list and the next and prev pointers are adjusted accordingly. In the end, f.next != f holds. If runFinalizer() is called on an unfinalized Finalizer, first, it is removed from the list, and then, it is finalized. Because all fields are private and the methods accessing these fields preserve the class invariant, the class invariant holds throughout program execution.

The example in Listing 3.2 also defines a mutual data structure, however, the class allows other classes to access this data structure through public methods. This code example is based on a class from a C++/Java framework for robot control systems [21].

The class DataTypeBase defines a private static final map called annotationMap, which maps classes to its unique non-negative annotation index, and a static final atomic integer called counter that stores the highest annotation index used so far (this corresponds to its invariant). annotationMap is initialised to empty and the integer to zero.

The method addAnnotation() can be called with a class as an argument. If this class is not yet in annotationMap, the counter is incremented, it gets assigned the next lowest available index and the mapping from this class to the index is added to the map. The method getAnnotation() retrieves the index corresponding to the input class and returns it. No method within this class removes entries from the map, and the map is private. Therefore, this class also maintains the history constraint that once an entry is added to the map, it remains there.

The last example of this pattern is shown in Listing 3.3. It is taken from the class StackStreamFactory from Java's Standard library. This is an example where the static class invariant of a class (StackFrameTraverser) depends on the invariant of another class. The class StackStreamFactory defines a set of classes, called stalkWalkClasses. The invariant of this class is that stalkWalkClasses contains classes, that must be excluded during stack walking. Moreover, it maintains a history constraint that once a class is added to the set, it won't be removed.

The class StackFrameTraverser is a nested class inside StackStreamFactory. Inside its static initialiser, StackFrameTraverser adds itself to the set stack-

```java
final class Finalizer {

   // Head of doubly linked list of Finalizers awaiting finalization.
   private static Finalizer unfinalized = null;

   // Lock guarding access to unfinalized list.
   private static final Object lock = new Object();

   private Finalizer next, prev;

   private Finalizer() {
      // add as head of the unfinalized list
      synchronized (lock) {
         if (unfinalized != null) {
            this.next = unfinalized;
            unfinalized.prev = this;
         }
         unfinalized = this;
      }
   }

   private void runFinalizer() {
      synchronized (lock) {
         if (this.next == this)      // already finalized
            return;
         // remove from unfinalized list
         if (unfinalized == this)
            unfinalized = this.next;
         else
            this.prev.next = this.next;
         if (this.next != null)
            this.next.prev = this.prev;
         this.prev = null;
         this.next = this;           // mark as finalized
      }

      finalize(this);
   }
}
```

Listing 3.1: The class Finalizer sets up a consistent global state and forbids others to access its static variables by declaring them as private.

```java
private static class DataTypeBase{
    // Lookup for data type annotation index
    private static final HashMap < Class<?>, Integer > annotationMap =
        new HashMap < Class<?>, Integer > ();

    // Last annotation index that was used
    private static final AtomicInteger counter =  new AtomicInteger(0);

    public void addAnnotation(Class<?> ann) {
        synchronized (DataTypeBase.class) {
            Integer i = annotationMap.get(ann.getClass());
            if (i == null) {
                i = counter.incrementAndGet();
                annotationMap.put(ann.getClass(), i);
            }
        }
    }

    public Integer getAnnotation(Class<?> c) {
        return annotationMap.get(c);
    }
}
```

Listing 3.2: The class DataTypeBase sets up a consistent global state by initialising its static variables. They can only be accessed through the methods addAnnotation() and getAnnotation() to ensure that the class invariant is not invalidated.

WalkImplClasses and thus assumes its invariant. Because of the history constraint, StackFrameTraverser can claim that after its initialisation, it will stay inside the set stalkWalkClasses.

An example of an object pool comes from Java's Standard library class Byte and is shown in Listing 3.4. The class ByteCache is a private class inside Java's class Byte. This class defines a final array of Byte objects as a static field. Inside its static initialiser, this array is filled with Byte objects containing all possible Byte values. This array acts as a cache or an object pool for Byte's method valueOf(). The valueOf() method is called with the primitive type byte b and returns a Byte object containing the value b. When a client calls this method, instead of creating a new Byte object on demand, it returns a Byte object from the cache. That way, one can reduce the memory and runtime overhead if the valueOf() method is often called.

The invariant of class ByteCache is that for all $b \in \{0, ..., 255\}$, the entry cache[b] holds a Byte object with value $b - 128$. It is established by the static initialiser. The array entries are never modified by the program, and thus this invariant is preserved. The same pattern can often be found in the Java

```
final class StackStreamFactory {

    // Stack walk implementation classes to be excluded during stack walking
    private final static Set<Class<?>> stackWalkImplClasses = init();

    private static Set<Class<?>> init() { ... }

    static class StackFrameTraverser {
        static {
            stackWalkImplClasses.add(StackFrameTraverser.class);
        }
    }
}
```

Listing 3.3: The static initialiser of the class `StackFrameTraverser` relies on the consistent global state, which is set up by the class `StackStreamFactory`.

```
private static class ByteCache{
    private ByteCache() {}

    static final Byte cache[] = new Byte[-(-128) + 127 + 1];

    static{
        for(int i = 0; i < cache.length; i++) {
            cache[i] = new Byte((byte)(i - 128));
        }
    }
}

public static Byte valueOf(byte b) {
    final int offset = 128;
    return ByteCache.cache[(int)b + offset];
}
```

Listing 3.4: The private class `ByteCache` inside `Byte.java` stores an object pool containing `Byte` objects.

Standard library, for example, in the private class `IntegerCache` inside the class `Integer`.

The second usage of the second pattern was to share computationally hard results. This is shown in Listing 3.5 on a class called `DemoImages` from the Java Standard library. This class defines two static fields. The first one is an array called `names` containing the names of demo images. The second field provides a mapping from these image names to `Image` objects and is called `cache`. Once the method `newDemoImages()` is called, this map is initialised.

```java
public class DemoImages {
    // names of the demo images
    private static final String[] names = {...};
    // mapping from names to images
    private static final Map<String, Image> cache =
            new ConcurrentHashMap<String, Image>(names.length);

    // initialise cache to a mapping from names to frequently used images
    public static void newDemoImages() {
        DemoImages demoImages = new DemoImages();
        for (String name : names) {
            cache.put(name, getImage(name, demoImages));
        }
    }

    public static Image getImage(String name, Component cmp) {
        Image img = null;
        if (cache != null) {
            if ((img = cache.get(name)) != null) {
                // return image from cache
                return img;
            }
        }

        img = // load image called 'name' using cmp
        return img;
    }
}
```

Listing 3.5: The class `DemoImages` uses the pattern, where a shared pool is stored as a static field to increase the performance of the program.

The method `getImage()` returns the image corresponding to the input `name`. If it exists inside cache, it is retrieved from there. Otherwise, the image is loaded and returned. Using the cache increases the performance of this program if the method `getImage()` is often called with an input name inside `names` under the assumption that loading images is computationally hard. The invariant of `DemoImages` is that `cache` is either empty or the entries of `cache` are of the form (img_name, img), where img_name is an entry of the array `names` and img is the corresponding loaded image of type `Image`.

The last example in Listing 3.6 shows the pattern used to register a callback in the class `Object` from Java's Standard library. The native method `registerNatives()` is declared in the class and is called inside `Object`'s static initialiser. What invariant is established depends on the contents of the native

```java
public class Object {

    private static native void registerNatives();
    static {
        registerNatives();
    }
    ...
}
```

Listing 3.6: Inside the static initialiser of the class `Object` a native method is registered.

code, and its verification is out of scope for this thesis.

Chapter 4

Basic methodology

As mentioned in Chapter 2.2, the implementation of static initialisers varies strongly between different programming languages. However, the implementations of static initialisers that we are aware of satisfy the key properties listed in Chapter 1. In this chapter, we introduce the programming language BAUM, a toy language used to model the behaviour of static initialisers as found in many mainstream programming languages. This programming language provides support for modules, static initialisers, and heap-allocated data structures. The language BAUM imposes no restrictions on the initialisation process other than Property 1 and Property 2 introduced in Chapter 1 and additionally, Property 3, which we add below specifically to help us model languages with subclassing. Because of that, it may be used as a model for many different programming languages. Furthermore, we provide a sound and modular specification and verification technique for the language BAUM. The following are the initialisation rules of the language BAUM:

Property 1 Each static initialiser runs to completion before any access to the static variables that are declared in its class, as long as there are no cyclical initialisation dependencies. Accesses to its own static variables inside the static initialiser are excluded from that.

Property 2 Each static initialiser runs at most once.

Property 3 The static initialiser of a subclass runs after the class it extends is initialised.

In this language, the initialisation of a module can start at any time during program execution, as long as the three properties above hold. The initialisation of a module A is modelled as spawning a new initialisation thread *A*, that runs concurrently to the main thread and possibly other initialisation threads. To satisfy Property 1, other threads are only allowed to access static fields declared in module A after thread *A* has completed the initialisation.

If access to a static field by a thread different from its initialisation thread occurs before the class is initialised, the language semantics guarantees that the access blocks until initialisation has been completed.

As seen in Chapter 2.2, under the assumption that no cyclical initialisation dependencies are present in the program, the first two properties hold for most mainstream programming languages. Property 3 holds in languages like Java or C# [11]. This general methodology applies to any programming language that satisfies the three properties. Because the initialisation order in this language is only vaguely specified, the set of possible executions for a program in this language is an over-approximation of the possible executions in a real programming language that satisfies the three properties.

From now on, the set of allowed programs is restricted to programs that do not contain cyclical dependencies between static initialisers. The reason for this restriction is that we did not find any interesting usages of mutual dependencies in static initialisers of different modules. Furthermore, in languages like Java, C#, or Scala, whenever we discovered a bug report related to this (usually resulting in deadlocks), the developers decided to break the mutual dependency. Examples of this issue can be found on GitHub in a project called Netty [8], a Java API called JavaPoet [6] or in Scala bug tracker [10]. A cyclical initialisation dependency always seemed undesirable because it may lead to deadlocks or reading uninitialised data, and in some languages Property 1 only holds in the absence of cyclical dependencies. As such, we opted to disallow it in this thesis.

In this chapter, we first define the syntax and the semantics of the language Baum. Then, we describe and illustrate our modular verification methodology for this language.

4.1 Language

In this section, we present the syntax and the semantics of the language Baum.

4.1.1 Syntax

The syntax of the language Baum is defined in Grammar 4.1. A program consists of possibly multiple modules. Each module M has a unique identifier called ID. By $\mathcal{ID}$ we denote the set of all module identifiers in the program. A module might contain static variables, a static initialiser and some methods. Some mainstream programming languages allow a module to have multiple static initialisers. However, since static initialisers in the

Type	:=	`Int` | `Pointer to Int`	
P	:=	M^*	*Program*
M	:=	`ID` *extend*? $svar\_decl^*$ *sinit*? m_decl^*	*Module*
sinit	:=	`static` {*Stmt*}	*Static initialiser*
svar_decl	:=	`static` *Type* `id` := *Expr*	
extend	:=	`extend ID`	
m_decl	:=	`static m(`Arg^*`)` *Type* {*Stmt*; `return` *Expr*}	*Static method*
Arg	:=	*Type* `id`	*Method argument*

Grammar 4.1: Grammar for language BAUM

same module are usually executed in textual order[1], merging them does not change the semantics of the program. Each static variable has an identifier *id* unique within its module. The identifier *m* ranges over all possible method identifiers and is also unique within its module. Only one method in the whole program can contain the `main` method identifier. There are only two possible types, so each variable can either store an integer or a pointer to an integer.

For simplicity's sake, in BAUM a module might extend at most one other module. The `extends` relation must be a partial order, thus acyclic. Note that the `extends` relation that we defined for the language BAUM is different from inheritance known from for example Java. Here, a module does not inherit all methods and fields from the module it extends. The only restriction we added for subclasses is Property 3 regarding the initialisation order. We define the helper function `extends` : $\mathcal{ID} \to \mathcal{P}(\mathcal{ID})$, which for a module returns the set of modules that it transitively extends.

Grammar 4.2 defines the statements and expressions that our language contains. The language consists of six different expressions that contain reads from local, static, and heap-allocated variables. The type checking for the expressions and statements follows the standard definitions. For example, the expressions in e_1 `binop` e_2 and `unop` e must be integers, while the expression e in `pderef` e must be a pointer. However, a local variable can be of either type.

The statements include method calls, conditional statements, and various types of stores. For example, the statement `pstore` e_1 e_2 means storing the value of e_2 into the location where e_1 points to. We did not include iteration statements in our grammar, but they can be modelled using recursive calls. The grammar can easily be extended to contain an iteration statement.

[1]This holds for Java and Go. Other languages presented in Section 2.2 do not allow for multiple static initialisers in the same module.

Expr	:=	`numeral`	*Integer*
	|	*Expr* `binop` *Expr*	*Binary expression*
	|	`unop` *Expr*	*Unary expression*
	|	`pderef` *Expr*	*Heap variable read*
	|	`sread ID.id`	*Static variable read*
	|	`x`	*Local variable read*
Stmt	:=	`x` := `ID.m(`*Expr**`)`	*Method call*
	|	`if(`*Expr*`) then` *Stmt* `else` *Stmt*	*If-statement*
	|	`sstore ID.id` *Expr*	*Static variable store*
	|	`pstore` *Expr* *Expr*	*Pointer store*
	|	`x` := *Expr*	*Local variable store*
	|	`x` := `pinit` *Expr*	*Allocate memory on heap*
	|	`var` *Type* `x` := *Expr*	*Local variable declaration*
	|	`return` *Expr*	*Return from function call*
	|	*Stmt*; *Stmt*	*Statement composition*

Grammar 4.2: Statements and expressions for language Baum

4.1.2 Semantics

In this subsection, we present the semantics of the language Baum.

Helper function access We define a helper function `access` : $Expr \cup Stmt \rightarrow \mathcal{P}(\mathcal{ID})$ for language semantics. It is defined in Listing 4.1 as pseudocode using OCaml syntax.

This function takes as input a statement or an expression and returns a set of modules whose static variables are accessed by the input. Property 1 specifies that each static initialiser runs to completion before any access to the static variables that are declared in its module. Thus, the function `access` returns the set of modules that need to be initialised before the expression can be evaluated, or the statement can make a step, i.e. a derivation rule can be applied. For example, the statement `Sstore ID.id 0` can only make a step after the module *ID* is initialised. Note that the output of the function `access` given an if-statement as input only depends on the condition expression of the if-statement and not on the statements inside its branches. This is because we do not want to enforce the initialisation of a module if its static variables are only read from or written to inside an unreachable branch and thus will never be accessed during program execution. Therefore, we can make a step with an if-statement if we can evaluate the condition but can't fully execute one of the branches yet. The same applies to sequential composition. A step can be made independently of the initialisation state of all modules.

In contrast to Java, where a method from some class `A` is only called after `A` has been initialised, here, this is not required. The only statements, or

```
let rec access expr : (ID set) =
  begin match expr with
  | (Numeral _) | (Var _) -> {}
  | (Binop e1 e2) -> union (access e1) (access e2)
  | (Unop e) -> access e
  | (Sread ID.id) -> union (extend ID) {ID}
  | (Pderef e) -> access e
  end

let rec access stmt : (ID set) =
  begin match stmt with
  | (x := ID.m(e1,...,en)) -> union (access e1) ...(access en)
  | (If e then s1 else s2) -> access e
  | (Sstore ID.id e) -> union (access e) (extend ID) {ID}
  | (Pstore e1 e2) -> union (access e1) (access e2)
  | (x := e) | (x := pinit e) | (var x := e) | (Return e) ->
    access e
  | (s1; s2) -> {}
  end
```

Listing 4.1: The function access returns a set of IDs whose initialisation must be completed before the execution of the given expression or statement.

expressions, that require `A` to be fully initialised to execute are reads from or writes to `A`'s static variables.

The used helper function `union` unifies an arbitrary amount of sets into one. `extend` is the previously defined function that returns for a module ID the set of modules that it transitively extends.

Program state The current program state is represented by the triple $\sigma = (\sigma_h, \sigma_m, \sigma_l)$. σ_h is a partial function that maps allocated heap addresses to the values they are holding. The second item, σ_m, maps each module identifier to a tuple containing a function that maps the module's static variables to their values and the module's current initialisation state. There are three initialisation states: `not init`, `init` and `ongoing`. A module is in the initialisation state `not init` if its initialisation has not yet begun, `init` if it has been fully initialised and `ongoing` if its initialisation is currently ongoing. The last item, σ_l, is a partial function that associates a value with each local variable. After a local variable is declared, the mapping from the local variable to its value is added to σ_l.

Our semantics includes two types of configurations. The first type takes the form $\langle s, \sigma \rangle$, where s is an instruction, and the instructions s' in s are annotated as follows: $\wr s' \wr_A$, where A is the thread that will execute this instruction. $\wr s';s'' \wr_A$ stands for $\wr s' \wr_A \,;\, \wr s'' \wr_A$, i.e. thread A will first execute instruction s' and then instruction s''. The exact meaning of the subscript is explained below. $\langle s, \sigma \rangle$ means that s is to be executed in state σ. The second type of configuration, σ, represents a final state.

The starting configuration of our program is $\langle \wr x := A.main() \wr_T, (\sigma_h^0, \sigma_m^0, \sigma_l^0) \rangle$. The execution of the program starts by calling the `main` method in the class which defines it, here called `A`. The subscript T on the right of the statement is a fresh ID that does not occur in the set $\mathcal{ID}$ in our program. We will refer to T as the main thread because T will be executing the `main` method. `x` is a local variable never used in the program. σ_h^0 is a partial function that does not yet map any address to a value. Similar to σ_h^0, σ_l^0 does not map any local variables to values yet. For all ID, the first entry of $\sigma_m^0(ID)$ maps all static variables to the zero value, so 0 for integers and `null` for pointers. The second entry of the tuple $\sigma_m^0(ID)$ stores the initialisation state `not init`.

Small-step semantics This paragraph defines the small-step semantics for the execution of the statements. The state $(\sigma_h, \sigma_m, \sigma_l)$ is often abbreviated by σ. The functions `fst` and `snd` return the tuple's first and second entries, respectively. The semantic function $\mathcal{A}$ takes as input an expression e from the language Baum and a state σ of the form described above. It returns the value of e evaluated under the state σ, which is represented as $\mathcal{A}[\![e]\!]_\sigma$.

In our execution model, we have defined that the initialisation of module `A` can start at any time if both conditions listed below hold:

- `A` is in initialisation state `not init`.
- If `A` extends module `B`, then `B` is in initialisation state `init`.

The first point follows from Property 2 and the second one from Property 3.

The initialisation in this language is modelled as a concurrent program. If both listed conditions hold, thread A can spawn and start the initialisation of module `A` in parallel to the main program executed by the main thread T or other initialising threads. Multiple threads can be initialising multiple modules concurrently, and each module has its corresponding initialisation thread. The subscript next to the statement denotes which thread is executing it. If a statement has subscript A different from T, it means that either this statement is inside `A`'s initialiser or this statement is in a method which was invoked from inside `A`'s static initialiser. So the initialisation of module `A` is ongoing at that moment. How initialisation is modelled in the language Baum is shown in the following rule with `snd` $\sigma_m(ID) =$ `not init` and `snd` $\sigma_m(B) =$ `init` for all $B \in$ `extend`(ID) as the side condition $(**)$.

$$\frac{(**)}{\langle s, \sigma \rangle \rightarrow_1 \langle\, \wr s' \wr_{ID} \parallel s, (\sigma_h, \sigma_m[ID \mapsto (\texttt{fst}\ \sigma_m(ID), ongoing)], \sigma_l) \rangle}\ (Init)$$

The statement s' above stands for $sdecl(ID);sinit(ID);finish_{ID}$. $sdecl(ID)$ and $sinit(ID)$ are the static variable declarations and the static initialiser code of the module with ID ID, respectively.

The statement $s_1 \parallel s_2$, where s_1 and s_2 are statements, is introduced for internal use only and cannot be used in the source program. $s_1 \parallel s_2$ encodes concurrent execution of s_1 and s_2, this means, their execution can be interleaved at the granularity of one step defined by $\rightarrow_1$.

Another internally introduced statement is finish_A. This statement marks the end of A's initialisation and on execution, sets A's initialisation state to init.

$$\frac{}{\langle \wr finish_A \wr_A, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m[A \mapsto (\texttt{fst}\,\sigma_m(A), init)], \sigma_l)}\ (Finish)$$

In general, the execution of a statement works as follows: a statement, which is executed by thread A, can only make a step if all modules that it uses (except for module A if A is an initialiser thread) are fully initialised. The function access returns a set of IDs of modules that must be initialised such that the execution of the statement is allowed according to Property 1. Given a state of the form $\langle \wr s \wr_A, \sigma\rangle$ we introduce a second side condition. All configurations of the form $\langle \wr s \wr_A, \sigma\rangle$ having $(*)$ on top of the rule can only be executed if the below-mentioned side condition holds. The side condition has two versions, depending on whether A is the main thread T or an initialisation thread of some module A. If $A \neq T$, the side condition looks as follows:

$$\text{for all } ID \in \texttt{access}(s) \setminus \{A\} \text{ holds } \texttt{snd}\,\sigma_m(ID) = \texttt{init} \qquad (*)$$

From the subscript A next to the statement s follows that A is initialising module A at that moment by initialisation thread A. A statement with subscript A is always allowed to read from or write to A's static variables. However, it is only allowed to access B's static variables if B is fully initialised, for some module B $\neq$ A. This allows A to access its own static variables during the initialisation of A, but it forbids reading variables declared in other classes if they have as initialisation state ongoing or not init, such that no uninitialised data is read. If A is the main thread T, then the side condition $(*)$ changes to

$$\text{for all } ID \in \texttt{access}(s) \text{ holds } \texttt{snd}\,\sigma_m(ID) = \texttt{init} \qquad (*)$$

This is because the main thread is not initialising any module. The semantics of the remaining statements will now be presented. In all rules, A is a placeholder for any thread, i.e., it can be an initialiser thread or the main thread.

First, we look at the execution of the method call x $:= ID.m(e_1, ..., e_n)$, where the method declaration m_decl of $ID.m$ is $m(x_1, ..., x_n)\{s;$ return $e_{ret}\}$. The corresponding rule has the above-defined side condition $(*)$. Therefore, this statement can only make a step after all modules, whose static variables are accessed by the methods arguments, are initialised. As long as this is not the case, execution stalls. The local variables used by the caller and the callee must be distinct from each other such that no variables are captured. The execution of the method call x $:= ID.m(e_1, ..., e_n)$ starts by executing the method body in a state, where the values of the arguments are assigned to the method's local variables. In the end, the return value is assigned to the local variable x. After the method call, the old local state of the caller must be restored. To do so, a restore(σ_l, x) statement is introduced. The first argument is the local variable state from before the call, and the second argument is the local variable to which the return value of the method call is assigned. The restore statement is used internally by the semantics, but it must not appear in the source code.

$$\frac{(*)}{\langle \wr x := ID.m(e_1, ..., e_n) \wr_A, \sigma\rangle \rightarrow_1 \langle \wr s;\ x := e_{ret}\ ;\ \texttt{restore}(\sigma_l, x) \wr_A, \sigma'\rangle}\ (Method)$$

We used the abbreviation $\sigma' = (\sigma_h, \sigma_m, \sigma'_l)$, where $\sigma'_l := \sigma^0_l[x_1 \mapsto \mathcal{A}[\![e_1]\!]_\sigma, .., x_n \mapsto \mathcal{A}[\![e_n]\!]_\sigma]$. σ^0_l is the function defined before, that does not map any local variables to values.

The restore(σ_l, x) statement restores the values of the local variables in σ_l from before the call and leaves x's value, which holds to the method's return value, unchanged.

$$\frac{}{\langle \wr restore(\sigma'_l, x) \wr_A, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m, \sigma'_l[x \mapsto \sigma_l(x)])}\ (Restore)$$

The conditional statement if (e) then s_1 else s_2 is executed as follows: if e holds in the current state (evaluates to a non-zero value), then s_1 is executed, else s_2 is executed.

$$\frac{(*),\ \mathcal{A}[\![e]\!]_\sigma \neq 0}{\langle \wr if\ (e)\ then\ s_1\ else\ s_2 \wr_A, \sigma\rangle \rightarrow_1 \langle \wr s_1 \wr_A, \sigma\rangle}\ (If_1)$$

$$\frac{(*),\ \mathcal{A}[\![e]\!]_\sigma = 0}{\langle \wr if\ (e)\ then\ s_1\ else\ s_2 \wr_A, \sigma\rangle \rightarrow_1 \langle \wr s_2 \wr_A, \sigma\rangle}\ (If_2)$$

Executing the statement sstore $ID.id\ \ e$ writes e's value into the static variable id in module ID. The module's initialisation state fst $\sigma_m(ID)[id \mapsto \mathcal{A}[\![e]\!]_\sigma]$ is abbreviated by $\sigma_{m,fst}$.

$$\frac{(*)}{\langle \wr sstore\ ID.id\ e \wr_A, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m[ID \mapsto (\sigma_{m,fst}, \mathsf{snd}\ \sigma_m(ID))], \sigma_l)}\ (Sstore)$$

Executing the statement `pstore` e_1 e_2 writes e_2's values to the heap location whose address is given by evaluating e_1.

$$\frac{(*),\ \mathcal{A}[\![e_1]\!]_\sigma = Addr}{\langle \wr pstore\ e_1\ e_2 \wr_A, \sigma\rangle \rightarrow_1 (\sigma_h[Addr \mapsto \mathcal{A}[\![e_2]\!]_\sigma], \sigma_m, \sigma_l)}\ (Pstore)$$

The statement $x :=$ `pinit` e allocates a fresh memory location $Addr_x$ on the heap, writes e's value into this location, and lets x point to it.

$$\frac{(*),\ Addr_x \notin Dom(\sigma_h)}{\langle \wr x := pinit\ e \wr_A, \sigma\rangle \rightarrow_1 (\sigma_h[Addr_x \mapsto \mathcal{A}[\![e]\!]_\sigma], \sigma_m, \sigma_l[x \mapsto Addr_x])}\ (Pinit)$$

The statement $x := e$ overrides the value of the previously declared local variable x in σ_l by e's value.

$$\frac{(*)}{\langle \wr x := e \wr_A, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m, \sigma_l[x \mapsto \mathcal{A}[\![e]\!]_\sigma])}\ (Lstore)$$

The statement `var` $x\ :=\ e$ extends the local variable store σ_l by the mapping from the fresh variable x to e's value.

$$\frac{(*),\ x \notin Dom(\sigma_l)}{\langle \wr var\ x := e \wr_A, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m, \sigma_l[x \mapsto \mathcal{A}[\![e]\!]_\sigma])}\ (Ldecl)$$

The four rules listed below represent parallelism. A distinction is made between whether s_1 or s_2 are executed entirely in one step or not.

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \langle s_1', \sigma'\rangle}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_1' \parallel s_2, \sigma'\rangle}\ (Par_1)$$

$$\frac{\langle s_2, \sigma\rangle \rightarrow_1 \langle s_2', \sigma'\rangle}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_1 \parallel s_2', \sigma'\rangle}\ (Par_2)$$

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \sigma'}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_2, \sigma'\rangle}\ (Par_3) \qquad \frac{\langle s_2, \sigma\rangle \rightarrow_1 \sigma'}{\langle s_1 \parallel s_2, \sigma\rangle \rightarrow_1 \langle s_1, \sigma'\rangle}\ (Par_4)$$

The last two rules model sequential composition.

$$\frac{\langle \wr s_1 \wr_A, \sigma\rangle \rightarrow_1 \sigma'}{\langle \wr s_1; s_2 \wr_A, \sigma\rangle \rightarrow_1 \langle \wr s_2 \wr_A, \sigma'\rangle}\ (Seq_1) \qquad \frac{\langle \wr s_1 \wr_A, \sigma\rangle \rightarrow_1 \langle \wr s_1' \wr_A, \sigma'\rangle}{\langle \wr s_1; s_2 \wr_A, \sigma\rangle \rightarrow_1 \langle \wr s_1'; s_2 \wr_A, \sigma'\rangle}\ (Seq_2)$$

4.2 Specification and Verification

This section introduces the methodology obtained for the language Baum.

In this methodology, each class contains a static class invariant and may additionally contain a history constraint (introduced in a paragraph below). A static class invariant is a predicate, which can specify properties of static fields. The static class invariant is not limited to its static fields only, it can also specify properties about static fields declared in other classes. Each static class invariant is enforced at the class level and is established by executing the static variable declarations and the static initialiser of the corresponding class (provided they terminate). After the initialisation of the class, its invariant must hold until program termination and can be assumed from everywhere in the program.

Open/close block To allow a class invariant to be assumed in the program, we introduce the instructions `open A` and `close A` for some module `A`. Together with the statements between these instructions, they form an `open/close` block. This block is only used for verification purposes, and the `open A` and `close A` instructions are ghost code. It gives the statements inside the block access to the static class invariant of class `A`. The meaning of `open A` and `close A` (for some class `A`) is the following: Upon `open A`, `A`'s invariant is inhaled. This means, the permissions specified in the invariant are added to the state, and the value constraints are assumed. On `close A`, `A`'s class invariant is checked to hold again, and the permissions are given away. If `A`'s invariant is broken, verification fails. At any program location, which is not inside the `open A/close A` block, and after `A` has been initialised, one is allowed to open `A` and thus assume its invariant. The idea of the `open A/close A` block is that, inside this block, the invariant of class `A` might be temporarily violated, but has to hold again on exit. However, because we have defined a very general language, allowing the class invariant to be broken would lead to an unsound methodology. A problem in this model is that the initialisation of a class might start at any time as long as Property 1 and Property 3, previously defined in this chapter, hold. The initialisation of a class could start running while concurrently another thread is inside the `open A/close A` block, so when `A`'s invariant might be violated, for some class `A`. However, the running static initialiser may rely on the invariant of `A`. The code example in Listing 4.2 demonstrates this problem.

There are two classes in this example, both containing a static variable. The predicates `inv_A` and `inv_B` represent `A`'s and `B`'s static class invariants, respectively. As explained in Section 2.4, `acc(A.x)` denotes write permissions to `A`'s static field `x` and allows everyone holding this permission to read from and write to `A.x`. In our methodology, one is allowed to assume the invariants of all classes that have been initialised. Given that `A.x` is accessed inside

```
public class A {                    public class B {
// inv_A: acc(A.x) && A.x >= 1      // inv_B: acc(B.y) && B.y >= 3
   static int x = 1;                   static int y;
   static int m() {
      // open A                        static {
      A.x = A.x - 2;                      // open A
      // A's inv. violated                y = 3*A.x;
      A.x = A.x + 2;                      // close A
      // close A                       }
      return 0;
   }                                }
}
```

Listing 4.2: This code example demonstrates why statements between `open A` and `close A` must be atomic. Our methodology rejects this example.

the method `m()`, it can be inferred that `A`'s initialisation did finish before the execution of `A.x = A.x - 2` and `m()` can assume `A`'s invariant. Note that in contrast to Java, in this language calling a method defined in class `A` does not require `A` to be initialised. Inside the `open A`/`close A` block, `m()` first violates `A`'s invariant and then establishes it again. In a sequential execution, this would not yield any problems. However, here, the initialisation of class `B`, whose initialiser relies on the validity of `A`'s invariant, might be running concurrently to the execution of `m()`, in the worst case, between the instructions `A.x = A.x - 2` and `A.x = A.x + 2`, when `A`'s invariant does not hold. This would lead to assuming an invariant while it is violated.

Since the time at which static initialization occurs is not deterministic in the Baum language, we cannot accept any invariant to be violated after it has been established. To resolve this problem, we have made the following restriction:

Rule Between `open A` and `close A` instructions, one can only place a physically atomic statement and possibly multiple ghost code instructions.

Because static initialisation may happen at any time, and we don't want to track for each module if its invariant holds at the moment, it would be unsound to allow class invariants to be broken. If only one atomic instruction can be placed inside the `open`/`close` block, its class invariant can never be broken, because it has to hold again on `close`. With this restriction, it is always sound to assume the invariant of a class after it has been established.

In our small-step semantic model as defined in Section 4.1, every statement that requires exactly one step to execute is atomic. Statements, that can be placed inside an `open`/`close` block are `pinit`, `ldecl`, `sstore`, `lstore` and pstore. In Chapter 5, this restriction is weakened for a language with a more restrictive specification for when static initialisation occurs.

Instructions inside the static initialiser In contrast to statements between open A and close A instructions, the statements inside the static initialiser of a module A do not need to be atomic. This is because a thread different from A's initialisation thread can only use a static field of module A or assume its invariant after the initialisation of A is completed. The thread has to wait for A's static initialiser to run until completion, such that A's initialisation state changes to init. As a result, a thread different from A's initialisation thread can't encounter a partially initialised state of module A during the initialisation of A.

We do not, however, allow method calls inside a static initialiser. A method called from within A's initialiser could rely on A's invariant, and because it is executed by A's initialisation thread, it would be allowed to access A's fields even though A's initialisation state is ongoing. To preserve modularity, we do not want to require methods to reveal which module's invariants they rely on, so we simply forbid method calls inside static initialisers.

In our methodology, accessing static variables requires permissions. Before its execution, the static initialiser is given full permissions to all static variables it declares and is therefore allowed to access them. The only way to transfer these permissions to the rest of the program is via its class invariant. If a static variable should stay mutable, the full permissions must be transferred into the static class invariant, so that others have the opportunity to write this variable. For immutable static variables, fractional permissions can be passed into the static class invariant. If an invariant contains full permissions to a static variable, opening the invariant transfers the permissions temporarily to the client and allows them to write to that static variable. These permissions must be returned on closing the invariant.

History constraints History constraints are predicates that describe how objects evolve. They do so by specifying relationships between some older states and some newer states. In our methodology, a history constraint of a class A must hold over all pairs of states s_{old}, s_{new}, such that s_{old} precedes s_{new} and neither s_{old} nor s_{new} are states within an open A/close A block. Therefore, both states s_{old}, s_{new} satisfy A's invariant. An example of a class containing a history constraint is shown in Listing 4.3. The history constraint specifies that the value of counter must be monotonically increasing.

History constraints must be reflexive and transitive. Additionally, a history constraint defined in class A must be framed by A's class invariant, i.e., the history constraint must not refer to the values of heap locations or static variables for which the invariant does not contain the permissions.

```
public class ObjectCounter {
// inv_ObjectCounter: acc(counter) && counter >= 0
// history_ObjectCounter: old(counter) <= counter
   static int counter = 0;
   ObjectCounter() {
      // open ObjectCounter
      counter = counter + 1;
      // close ObjectCounter
   }
}
```

Listing 4.3: For presentation purposes, we are extending our grammar by a constructor. The class `ObjectCounter` contains a static variable `counter`, that counts the number of objects of this class, i.e., how often the constructor is called. Its invariant specifies that the value of `counter` is always greater than or equal to 0. The class additionally contains the history constraint that the value of `counter` must be monotonically increasing.

Once upon a time Static variables can not only be accessed inside the static initialiser of the module that defines them. If one can get permissions (through opening an invariant) to static variables, one can read from and write to these fields as long as it does not break the opened class invariant. The invariant can be opened inside a static initialiser of another class or inside a method. Inside the `open/close` block, some property regarding these static fields might be established. As seen in Chapter 3, the pattern where static initialisers establish properties with respect to static fields defined in other classes is used in the Java standard library. In Listing 4.4, this pattern is demonstrated in a simple example. The initialiser of class `A` receives the

```
public class A {
// inv_A: acc(m)

   Map<K,V> m = new Map<K,V>();

}
```

```
public class B {
// inv_B: A.m[K] == 1
   static{
      // open A
      A.m.put(K,1);
     // close A
}}
```

Listing 4.4: The static initialiser of class `B` modifies a static map, which is defined in another class. The invariant `inv_B` is invalid.

permissions for its static map `m` and initialises it to an empty map. After the initialisation, `A` transfers the permission `acc(m)` into its class invariant and thus allows others to modify this map. For demonstration purposes, we are extending our grammar defined in Section 4.1 by a map data structure on the heap and assume that inserting or deleting key/value pairs from the map can be done atomically (e.g., using ConcurrentMap [28] in Java). The initialiser of class `B` receives the permissions to `A.m` by opening `A`'s invariant, inserts the element `(K,1)` for some key `K` into the map and then transfers the permissions back to `A`'s invariant.

The first problem in this example is that `B`'s static initialiser must transfer the full permission to `A.m` back into `A`'s invariant. Thus, `B`'s static class invariant cannot contain any permissions to `A.m`, however, we may still want to add information about `A.m` to `B`'s invariant. We call a predicate that is established concerning static fields in class `A` a once-upon-a-time (OUAT) predicate with respect to `A`. The predicate `A.m[K] == 1` that `B` established, is not framed by the permissions in B's invariant, so it cannot be placed directly inside `B`'s invariant.

This problem is resolved by creating an assertion `OUAT(A.m[K] == 1)`. This assertion is assumed to hold after the OUAT predicate is established. It acts as a proof that this predicate was established. OUAT predicates can be established everywhere in the code. If we establish it inside `B`'s static initialiser, the assertion can be transferred into `B`'s class invariant to show others that the corresponding OUAT predicate was established.

A bigger problem comes from the fact that `A.m` is mutable, and thus its entries can be modified from all program locations. After `B`'s initialisation is completed, another class might simply open `A` like `B` did, modify the mapping `K` $\mapsto$ `1` to for example `K` $\mapsto$ `3` and thus invalidate what `B` has established. Even though `B` can claim that once upon a time it inserted the key-value pair `(K,1)` into the map, without additional constraints, it cannot be concluded that this property is preserved. This is where history constraints come into play. As previously stated, history constraints $\varphi(s_{old}, s_{new})$ relate an older state s_{old} to a newer state s_{new}. If a class `A` contains a history constraint φ, $\varphi(s_{old}, s_{new})$ must hold for all states where s_{old} comes before s_{new} and the module invariant is not opened in either s_{old} or s_{new}. Given some history constraint φ and a predicate p, if the validity of both p in state s_{old} and $\varphi(s_{old}, s_{new})$ imply that p holds in s_{new} for arbitrary states s_{old} and s_{new}, then p is said to be stable under the history constraint φ. In our methodology, the history constraint (if provided) is checked on `close`, where the older state is the state before opening the invariant.

Back to our example: `A` can specify as a history constraint that once the key `K` is in the map `m`, and it maps to 1, the mapping `K` $\mapsto$ `1` cannot be removed. Under this history constraint, if the property `A.m[K]==1` is established, it cannot be undone. Thus, the property established by `B` is stable under this history constraint.

To specify that a OUAT predicate p w.r.t. class `A` is established, the verification statement `close A establishing` p is introduced. If p was established, `A`'s invariant and history constraint are not violated, and the above-mentioned conditions hold, on `close A establishing` p, the assertion `OUAT(`p`)` is assumed and directly afterwards `A` is closed. The predicate p must be framed by `A`'s class invariant. Moreover, p must be stable with respect to `A`'s history constraint, and it must be duplicable. A predicate is said to be duplicable if duplicating it won't change its meaning. For example, duplicating the read permission `acc(`$o.f$`,1/2)` for some field f of object o gives us write permission to $o.f$ and thus changes its meaning. Thus, the shown predicate is not duplicable, however, $o.f == 3$ without any permissions is. To enforce OUAT predicates to be duplicable, we disallow them to contain any permissions.

In Listing 4.5, the previous example is encoded using the obtained methodology. `OUAT(A.m[K] == 1)` is proof that `A.m[K] == 1` held at some point while the invariant was closed. Moreover, `A.m[K] == 1` is duplicable and stable under `A`'s history constraint, so the assertion is also a proof that `A.m[K] == 1` will still hold when `A`'s invariant is opened again.

```
public class A {
// inv_A: acc(m)
// history_A: (k in dom(old(m)) => k in dom(m)) &&
// (old(m)[K] == 1 => m[K] == 1)

   Map<K,V> m = new Map<K,V>();
}

public class B {
// inv_B: OUAT(A.m[K] == 1)
   static{
      // open A
      A.m.add(K,1);
      // close A establishing A.m[K] == 1
   }
}
```

Listing 4.5: The same code example as in Listing 4.4, but now the invariant of class `B` contains the assertion `OUAT(A.m[K] == 1)` as a placeholder for the established OUAT predicate `A.m[K] == 1`, which is received from the instruction `close A establishing A.m[K] == 1`. Moreover, class `A` contains a history constraint under which `A.m[K] == 1` is stable.

After a OUAT predicate p w.r.t. class `A` is established, from opening `B`, the corresponding assertion `OUAT(`p`)` can be assumed. The body of the OUAT predicate, however, can only be assumed inside an `open A`/`close A` block, because it might not be self-framing on its own. We can only assume the body of a OUAT predicate where `A`'s invariant is closed because the history constraint does not make any guarantees about the states inside an open `A`/`close A` block. This is demonstrated in Listing 4.6. This example uses the same classes as in Listing 4.5. In this example, you can see that we cannot

```
// open A
A.m.delete(K);
// open B
assert false; // succeeds
// close B
// close A
```

Listing 4.6: The OUAT predicate `A.m[K] == 1` is assumed at a point, where `A`'s history constraint is violated and the OUAT predicate is broken. This would allow us to assert false. Note that all instructions except for `A.m.delete(K)` are ghost code, thus the statement inside the `open A`/`close A` block is still atomic.

assume the body of the OUAT predicate while `A`'s invariant is opened. To solve this problem, asserting `OUAT(`p`)` and assuming both p and `A`'s invariant are combined into one verification statement: `open A using` p. This statement can only be executed if p is a logical consequence of a OUAT predicate w.r.t. `A`, for which the assertion holds. To assume the body of the predicate, on `open A using` p the knowledge that `OUAT(`p`)` holds must be ready.

There are several possibilities to get this knowledge. Since `OUAT(`p`)` is an assertion, it can be received by previously establishing the corresponding OUAT predicate inside the same class member or inside a function that is called there (if this function puts this assertion in its postcondition). Inside a method, the assertion can be received from its precondition. Another possibility is to inhale `OUAT(`p`)` from a class invariant of a class `B` using the following pattern:

```
// open B
assert OUAT(p)
// close B
```

To learn that `OUAT(`p`)` holds, we open `B` before opening `A`. Because `OUAT(`p`)` is an assertion, the knowledge about it gained through opening `B` stays after closing `B`. After that, the OUAT predicate p can be assumed using `open A using` p. Note that opening `B` only assumes the assertion but it does not assume the body of the corresponding predicate.

To summarise, static initialisers or methods can specify that they have established a OUAT p with respect to class `A` by writing `close A establishing` p and assuming the assertion corresponding to p. If inside a static initialiser, this assertion can be put inside its static class invariant. The established OUAT predicate p must be stable with respect to `A`'s history constraint. If the assertion is inside `B`'s class invariant, it can be assumed by opening `B`. Then, using `open A using` p, the predicate body p can be assumed.

An invariant can only be established after class' initialisation As previously mentioned, one can only assume the invariant of a class after it is initialised. Thus, the statements `open A` and `open A using` p can only be executed if `A` is initialised at that point.

Why this rule is necessary is shown using the previously used example in Listing 4.5. The key-value pair `(K,1)` is only added inside `B`'s static initialiser. If the static initialiser of `B` did not run, then `B`'s class invariant might have never been established. Using the map `A.m` only triggers the initialisation of class `A` (and classes that `A` extends) but not the initialisation of class `B`, so one cannot assume that `(K,1)` is in the map `A.m` when using `A`. Thus, `open B` can only be executed and the assertion can only be inhaled after `B`'s initialisation is completed.

If we extended our language syntax by a `while` statement, the example in 4.7 would even be possible. Since the static initialiser will never terminate, it

```
public class A {
// inv_A: false
   static {
      while(true) {}
   }
}
```

Listing 4.7: A non-terminating static initialiser establishes `false` as its static class invariant.

is possible to establish `false` as its invariant. However, in our methodology, since it never terminates, we are not allowed to assume its static class invariant anywhere in the program.

In our small-step semantic model, the initialiser thread of a module `A` can access static variables from modules different from `A` only if their initialisation is completed. So the access to a static variable of some module `B` ($\mathtt{A} \neq \mathtt{B}$) suffices as proof that `B` is initialised. However, `A`'s initialiser thread can always access `A`'s static variables. As a result, accessing `A`'s static variable inside `A`'s static initialiser does not prove that `A`'s initialisation is completed. To resolve this problem, we disallow assuming `A`'s static class invariant there.

In our methodology, method calls from inside a static initialiser are disallowed. Therefore, if we are inside a method, we know that this method is executed by the main thread. The main thread can access a static variable only after the class that declares it has been initialised. From that follows that within a method, access to a static variable is always sufficient as proof of initialisation.

The statements `open A` and `open A using` p (which must not be inside `A`'s static initialiser) can only be executed if one of the following holds:

- A previous statement inside this method or static initialiser accesses a static variable declared by module `A`.
- The statement inside the `open/close` block accesses a static variable declared by module `A`.

The first point is straightforward. Regarding the second point, we know that the statement inside the `open/close` block cannot be executed until `A` has been initialised. Even though the statement placed *after* `open A` triggers `A`'s initialisation, because `open A` and `open A using` p are just verification instructions, the program can be thought of as moving the `open A` statement between the initialisation of `A` and the execution of the enclosed statement. Then it again holds that the invariant of class `A` is only assumed after `A`'s initialisation.

Summary To summarise, these are the assumptions of our methodology:

- The program does not contain any cyclical dependencies.
- Each module must define a static class invariant and might also define a history constraint.
- Everything that is opened must be closed.
- We can open an invariant for a physically atomic instruction everywhere where we know that the initialiser has run (or if the instruction inside the open block has a use of this module). From that follows that the statements `open A` and `open A using` p cannot be placed inside the static initialiser of `A`.
- Opening invariants can be nested. However, one can never nest opening the same module.

These are the guarantees our methodology gives:

- The static initialiser of module `A` must establish its static class invariant (if it terminates).
- After initialisation, invariants and history constraints cannot be broken.

Examples Two examples are encoded using the obtained methodology and presented here.

The first example in Listing 4.8 is `ByteCache` introduced in Section 3.2. Its invariant is that for all $i \in \{0, ..., 256\}$, the entry `cache[`i`]` contains a `Byte` object holding value $i - 128$. The static initialiser, which does not have to be atomic in this methodology, establishes the class invariant inside the for-loop. The method `valueOf()` in the class `Byte` uses `ByteCache`'s class invariant to retrieve its result from the cache. Array lookup can be done atomically. Because the static field `cache` is used inside the `open ByteCache/close`

ByteCache block, opening the invariant of ByteCache is safe there. This code example verifies successfully with respect to the specified class invariant.

```
private static class ByteCache{
// Inv_ByteCache: acc(cache, wildcard) && (forall i:Int :: 0 <= i &&
// i < |cache| ==> acc(cache[i].value) && cache[i].value == i-128)

   static final Byte cache[] = new Byte[-(-128) + 127 + 1];
   static{
      for(int i = 0; i < cache.length; i++) {
         cache[i] = new Byte((byte)(i - 128));
      }
   }
}

public static Byte valueOf(byte b) {
     final int offset = 128;
     // open ByteCache
     Byte B = ByteCache.cache[(int)b + offset];
     // close ByteCache
     return B;
}
```

Listing 4.8: The ByteCache example from Section 3.2 encoded using the obtained methodology. The invariant of the class ByteCache is assumed in Byte's method valueOf(). From this invariant follows that B is a Byte object holding the value b.

The second example in Listing 4.9 is a class called Fib that computes the Fibonacci numbers efficiently using a shared cache to reduce the computation overhead. During initialisation, a private static map cache is created. The cache provides a mapping from natural numbers to their corresponding Fibonacci number. It is initialised lazily, i.e., a mapping is only added after the result has been requested. The invariant of Fib is that the entries of cache are of the form $(n, \texttt{FibPure}(n))$ for natural numbers n, where FibPure is a mathematical function returning the corresponding Fibonacci number and the entries $(0,0)$ and $(1,1)$ are in the map (and serve as base cases). Additionally, the history constraint specifies that once a key is in cache, it will never be removed and the value to which it maps stays unchanged. For the sake of readability, the invariant and the history constraint of Fib are defined using mathematical notation.

$$
\begin{aligned}
Inv_{Fib} : &\ \texttt{acc(cache)} \land \{0,1\} \subseteq \texttt{dom(cache)} \land \\
&\ \forall i \in \texttt{Fib().cache}\ (i \geq 0 \implies \texttt{Fib().cache}[i] == \texttt{FibPure}(i)) \\
History_{Fib} : &\ \forall i\ \ (i \in \texttt{old}[s_{old}](\texttt{Fib().cache}) \implies \texttt{old}[s_{new}](\texttt{Fib().cache}))
\end{aligned}
$$

The static initialiser establishes the static class invariant by initialising `cache` to contain the entries $(0,0)$ and $(1,1)$. The static class invariant is used inside the method `fib()`. Because here, instructions inside an `open`/`close` block must be atomic, the method `fib()` must open and close the invariant three times. On the first `open Fib` instruction, the method checks if the key n is inside the map. On `close Fib`, it establishes the OUAT predicate `b ==> n in Fib().cache`, which is stable w.r.t. `Fib`'s history constraint. Inside the if-branch, from the established OUAT predicate and `Fib`'s history constraint, we can conclude that if b was `true` before, then n must be inside `Fib().cache` now. In this case, the corresponding value v can safely be read. From `Fib`'s invariant follows that v `== FibPure(`n`)`. In the else-branch, the `fib()` function is called twice outside the `open Fib`/`close Fib` block and afterwards, the result is put into the cache inside the last `open Fib`/`close Fib` block. This code example verifies successfully given the provided specification.

```
public class Fib {
// Inv_Fib
// History_Fib

   public static Map<Integer, Integer> cache;
   static {
      cache = new HashMap<Integer, Integer>();
      cache.put(0,0);
      cache.put(1,1);
   }

   static int fib(int n) {
      // open Fib
      bool b = cache.contains(n);
      // close Fib establishing b ==> n in Fib().cache
      int v = 0;
      if (b) {
         // open Fib using b ==> n in Fib().cache
         v = cache.get(n);
         // close Fib
      }
      if (!b) {
         v = fib(n-1) + fib(n-2);
         // open Fib
         cache.put(n,v);
         // close Fib
      }
      return v;
   }
}
```

Listing 4.9: `Fib` example from Section 3.2 encoded using three `open`/`close` blocks. The OUAT predicate `b ==> n in Fib().cache` is established and assumed inside the method `fib()`.

Chapter 5

Methodology for Java

This chapter provides a verification methodology for the Java language. The language defined in Section 4.1 imposes only the most general restrictions on static initialisation. Because of that, the obtained methodology is quite restrictive. This chapter deals with a simplified version of the Java language. The syntax and semantics of this language are based on Java. One difference between our general model and the Java model is that Java uses lazy initialisation. A static initialiser only runs directly before the first use of the class. Thus, if a statement does not have a use of a class or only uses of classes that are already initialised, we can be sure that it will not trigger its initialisation on execution. Moreover, static initialisation does not run concurrently with the program execution. The methodology for the Java language model allows us to include more than one statement inside the `open`/`close` block. Listing 5.1 provides an example that is valid in the Java setting, but not in our general model.

```java
public class B {
   // inv_B: x >= 0
   int x = 0;

   static void main() {
          // open B
          B.x = B.x - 5;   // (1)
          B.x = B.x + 5;   // (2)
          // close B
   }
}
```

```java
public class C {
   // inv_C: true


        static{
          // open B
          B.x = 3 * B.x;
          // close B
        }

}
```

Listing 5.1: In contrast to the methodology for the language BAUM, the Java language allows us to place multiple instructions inside the `open B`/`close B` block.

In this example, you can see that the main method in class `B` contains two statements between the `open B` and `close B` instructions. The statement (1) breaks `B`'s invariant and (2) establishes it again. In the methodology for the language Baum, we do not allow this code example, because the instructions inside the `open B`/`close B` block are not atomic. The initialisation of module `C`, a module, that relies on the invariant of class `B`, could happen between instructions (1) and (2). So, `C`'s static initialiser could assume `B`'s invariant at a point where it does not hold.

However, in the single-threaded Java setting, we can allow this example, since we can prove that no code that relies on `B`'s invariant may run between statements (1) and (2). Statements (1) and (2) may only trigger the initialisation of class `B`. Since `B` is already initialised before the execution of `main()` before opening `B`, no class will be initialised in between. Since static initialisation is more strongly defined in Java than in the Baum language, it allows us to lift the restriction about atomicity and define a methodology that accepts a larger set of programs.

This methodology may only be applied to programs that do not contain cyclical initialisation dependencies. While this is not true for all programs in Java, we also provide a modular way of proving the absence of cyclical initialisation dependencies using static levels, which we introduce in Section 5.2.

First, in Section 5.1, we define the language used in this chapter. Then, we present a verification methodology catered to this language. In the end, we provide an encoding of our methodology into Viper and explain the design choices.

5.1 Language

In this section, we present the definition of our Java-like language. The syntax of this language is identical to the one of Baum defined in Subsection 4.1.1. We refer to Grammar 4.1 and Grammar 4.2 for its definition.

Next, we define the semantics of this language. The implementation of static initialisation in Java varies significantly from the semantics defined in Subsection 4.1.2.

Helper function trigger_init Similar to the helper function `access` from Section 4.1, we define a helper function `trigger_init` : $Expr \cup Stmt \rightarrow ID\ list$. In contrast to the return type of the function `access`, which is a set of IDs, the return type of this function is a list of IDs. The function `trigger_init` returns a list of modules whose initialisation the input statement may trigger (if this module is not initialised yet), and the initialisation is triggered according to the order defined in the list. The ordering is defined according to the evaluation order in Java. For example, binary expressions are evaluated from left to right in Java. This means that the binary expression `A.a + B.b`, where `A` and `B` are classes containing the static fields `a` and `b` respectively, triggers first `A`'s initialisation, then `B`'s. And the method call `A.m(B.b)` to static method `m` in class `A` triggers first the initialisation of the class of its argument (in this case, `B`) and then the one of its base class (here, `A`) [26]. If for example the static initialisers of both classes `A` and `B` write to the same static variable of a different module, the order of initialisation has an impact on the outcome of the program. The function `trigger_init` is defined in Listing 5.2 as pseudocode using OCaml syntax.

```
let rec trigger_init expr : (ID list) =
  begin match expr with
  | (Numeral _) | x -> []
  | (Binop e1 e2) -> remove_dupl ((trigger_init e1) @ (trigger_init e2))
  | (Unop e) -> trigger_init e
  | (Sread ID.id) -> (extend ID) @ [ID]
  | (Pderef e) -> trigger_init e
  end

let rec trigger_init stmt : (ID list) =
  begin match stmt with
  | (x := ID.m(e1,...,en)) -> remove_dupl ((trigger_init e1) @ ... @
        (trigger_init en) @ (extend ID) @ [ID])
  | (If e then s1 else s2) -> trigger_init e
  | (Sstore ID.id e) -> remove_dupl ((trigger_init e) @ (extend ID) @ [ID])
  | (Pstore e1 e2) -> remove_dupl ((trigger_init e2) @ (trigger_init e1))
  | (x := pinit e) | (Return e) | (x := e) -> trigger_init e
  | (s1; s2) -> []
  end
```

Listing 5.2: The function `trigger_init` returns a list of IDs that must be initialised before the expression can be evaluated, or the statement can make a step.

The used helper function `extend` : $ID \rightarrow ID\ list$ returns for a module ID the list of modules which it transitively extends. If ID_1 precedes ID_2 in the list, then it is a superclass of ID_2. We defined it this way because, in Java, superclasses are initialised before the classes they extend [25]. Another helper function is `remove_dupl`. It takes a list and returns a list with all duplicate elements removed except for the first one, while keeping the element order intact (e.g. `remove_dupl [1,2,3,2] = [1,2,3]`).

Program state The definition of a program state here is identical to the one defined in Subsection 4.1.2. Our semantics again includes the two types of configurations: $\langle s, \sigma \rangle$ and σ for a statement s and state σ. However, the statement s is now not annotated with any thread name. This is because our semantics rely on Java's static initialisation procedure, where in a sequential program initialisation occurs sequentially. In this chapter, all programs are executed by only one thread, so the annotation $\wr . \wr_A$ of statements for some thread A is omitted.

As previously, the starting configuration of our program is $\langle x := A.main(), (\sigma_h^0, \sigma_m^0, \sigma_l^0) \rangle$. The mappings σ_h^0, σ_m^0 and σ_l^0 are defined like in Subsection 4.1.2.

Small-step semantics The execution of the statements proceeds differently than in the language Baum. Apart from the program execution being sequential, Java uses lazy initialisation. This means classes are initialised on their first use. In our language, a module `A` is considered *used by statement s* if s contains an access to a static variable declared in class `A` (using `sacc` or `sstore`) or if it is a call to a method defined in `A`. The execution of a statement s works as follows: before s is executed, it triggers the initialisation of all modules it uses that are not initialised yet. The initialisation is triggered according to the order given by $\texttt{trigger\_init}(s)$. So, the class that is at the top of the list and is not yet initialised is selected, and its initialisation is started. The execution of s follows. This is defined in the following rule:

$$\frac{(**)}{\langle s, \sigma \rangle \rightarrow_1 \langle s'; s, (\sigma_h, \sigma_m[ID \mapsto (\texttt{fst}\, \sigma_m(ID), ongoing)], \sigma_l) \rangle}\ (Init)$$

The side condition $(**)$ specifies that ID is an entry of $\texttt{trigger\_init}(s)$ such that $\texttt{snd}(\sigma_m(ID)) = \texttt{not init}$ and $\texttt{snd}(\sigma_m(ID')) \neq \texttt{not init}$ for all ID' that come before ID in the list $\texttt{trigger\_init}(s)$. According to the JLS, the initialisation of a module whose initialisation state is `ongoing` is not triggered.

The statement s' is defined as $s' := sdecl(ID); sinit(ID); finish_{ID}$, where $sdecl(ID)$ and $sinit(ID)$ are the static variable declarations and the static initialiser code of the module with ID ID, respectively.

According to the JLS, a statement s can only be executed after all modules it uses are initialised, or their initialisation is ongoing (by the same thread), this means if their initialisation states are either `init` or `ongoing`. The uses of a statement s are computed by $\texttt{trigger\_init}(s)$. Given a state of the form $\langle s, \sigma \rangle$, we introduce a second side condition, which summarises what is written above:

$$\text{for all } ID \in \texttt{trigger\_init}(s) \text{ holds } \texttt{snd}\ \sigma_m(ID) \in \{\texttt{init}, \texttt{ongoing}\} \qquad (*)$$

All configurations of the form $\langle s, \sigma \rangle$ having $(*)$ on top of the rule can only be executed if the above-mentioned side condition holds.

The remaining rules are very similar to those defined in Section 4.1, except that everything is executed by the same thread. Thus, rules for parallel composition are no longer necessary. All rules are listed below.

$$\frac{}{\langle finish_A, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m[A \mapsto (\texttt{fst}\ \sigma_m(A), init)], \sigma_l)}\ (Finish)$$

$$\frac{(*)}{\langle x := ID.m(e_1, ..., e_n), \sigma\rangle \rightarrow_1 \langle s;\ x := e_{ret}\,;\ \texttt{restore}(\sigma_l, x), (\sigma_h, \sigma_m, \sigma_l')\rangle}\ (Method)$$

We use the abbreviation $\sigma_l' := \sigma_l^0[x_1 \mapsto \mathcal{A}[\![e_1]\!]_\sigma, .., x_n \mapsto \mathcal{A}[\![e_n]\!]_\sigma]$. The method declaration of $ID.m$ is $m(x_1, ..., x_n)\{s;\ \texttt{return}\ e_{ret}\}$

$$\frac{}{\langle restore(\sigma_l', x), \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m, \sigma_l'[x \mapsto \sigma_l(x)])}\ (Restore)$$

$$\frac{(*),\ \mathcal{A}[\![e]\!]_\sigma \neq 0}{\langle if\ e\ then\ s_1\ else\ s_2, \sigma\rangle \rightarrow_1 \langle s_1, \sigma\rangle}\ (If_1) \qquad \frac{(*),\ \mathcal{A}[\![e]\!]_\sigma = 0}{\langle if\ e\ then\ s_1\ else\ s_2, \sigma\rangle \rightarrow_1 \langle s_2, \sigma\rangle}\ (If_2)$$

$$\frac{(*)}{\langle sstore\ ID.id\ e, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m[ID \mapsto (\texttt{fst}\ \sigma_m(ID)[id \mapsto \mathcal{A}[\![e]\!]_\sigma], \texttt{snd}\ \sigma_m(ID))], \sigma_l)}\ (Sstore)$$

$$\frac{(*),\ \mathcal{A}[\![e_1]\!]_\sigma = Addr}{\langle pstore\ e_1\ e_2, \sigma\rangle \rightarrow_1 (\sigma_h[Addr \mapsto \mathcal{A}[\![e_2]\!]_\sigma], \sigma_m, \sigma_l)}\ (Pstore)$$

$$\frac{(*),\ Addr_x \notin Dom(\sigma_h)}{\langle x := pinit\ e, \sigma\rangle \rightarrow_1 (\sigma_h[Addr_x \mapsto \mathcal{A}[\![e]\!]_\sigma], \sigma_m, \sigma_l[x \mapsto Addr_x])}\ (Pinit)$$

$$\frac{(*)}{\langle x := e, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m, \sigma_l[x \mapsto \mathcal{A}[\![e]\!]_\sigma])}\ (Lstore)$$

$$\frac{(*),\ x \notin Dom(\sigma_l)}{\langle var\ x := e, \sigma\rangle \rightarrow_1 (\sigma_h, \sigma_m, \sigma_l[x \mapsto \mathcal{A}[\![e]\!]_\sigma])}\ (Ldecl)$$

$$\frac{\langle s_1, \sigma\rangle \rightarrow_1 \sigma'}{\langle s_1; s_2, \sigma\rangle \rightarrow_1 \langle s_2, \sigma'\rangle}\ (Seq_1) \qquad \frac{\langle s_1, \sigma\rangle \rightarrow_1 \langle s_1', \sigma'\rangle}{\langle s_1; s_2, \sigma\rangle \rightarrow_1 \langle s_1'; s_2, \sigma'\rangle}\ (Seq_2)$$

5.2 Methodology

In this section, we provide a methodology for the simplified Java language defined in Section 5.1. This methodology is strongly based on the one defined in Chapter 4 for the general language, however, it has much weaker restrictions. We extend the syntax of our language by constructors that allow us to create objects and by non-static methods. The non-static methods of a superclass are inherited by the classes that extend it. As a simplification, static fields are not inherited here. Additionally, methods can have `void` as return type.

In this methodology, as in the previous chapter, each class must define a static class invariant and may also specify a history constraint. Again, OUAT predicates with respect to other classes can be established in the code. During the static initialisation of a class, the static class invariant is established.

Verification statements The verification statements `open A`, `close A`, `close A establishing` p and `open A using` p for some class `A` and predicate p defined in Section 4.2 are reused, and its semantics remain the same. The actions that are taken on their execution are summarised here:

`open A` The invariant of class `A` is assumed.

`close A` The invariant and, if present, the history constraint of `A` are checked to hold. The history constraint is checked between the states on `open A` and on `close A`.

`close A establishing` p The OUAT predicate p is asserted to hold. The assertion `OUAT(`p`)` is assumed to hold and immediately afterwards `A` is closed.

`open A using` p If p is a logical consequence of a OUAT predicate p' w.r.t. `A` and `OUAT(`p'`)` holds, then `A`'s invariant and p are assumed to hold.

The greatest difference to the previous methodology is that in this methodology, static class invariants are allowed to be broken temporarily after they have been established. We allow multiple statements to be placed inside an `open`/`close` block. These statements may break the invariant inside the `open`/`close` block, but they must re-establish it on `close`. Inside a static initialiser, multiple instructions are still allowed. Additionally, method calls from static initialisers or `open`/`close` blocks that satisfy the static level restrictions are now accepted, as explained later.

As previously said, the statement `close A establishing` p is reused from the previous chapter. Here, it is even more important that asserting p and closing `A` are summarised into one statement. This is because multiple statements are now allowed within the `open A`/`close A` block. If asserting p and assuming `OUAT(`p`)` would be implemented as a single instruction and not combined with closing `A`, the established predicate could be violated before `A` is closed but after `OUAT(`p`)` was assumed. Because the history constraint of `A` is only checked between opening and closing `A`, closing `A` would succeed. This would result in an unsound methodology. In our methodology, if p holds while closing `A`, and p is stable w.r.t. `A`'s invariant, it will hold until program termination.

Static levels We wanted to weaken the restriction from Chapter 4 which disallows method calls inside a static initialiser or an open/close block. The first idea was that each method must be annotated with the names of all classes whose invariants it assumes or whose initialisation it might trigger. However, this would reveal too many implementation details and thus break information hiding. So, to allow for method calls inside an open/close block and still hide the implementation of a method, we introduce static levels. The idea comes from the fact that methods called from inside a static initialiser are often helper functions that do not access static fields or assume anything about the class that called them, such as Java's `HashMap.put()` method.

In our methodology, each statement, method, constructor, and static initialiser must be annotated with a static level, which is a non-negative integer. The *static level of a class* refers to the static level of its static initialiser. The main idea of static levels is the following: If something has a static level of a, it can only trigger the initialisation or assume the invariant of classes whose static level is less than or equal to a. A class having a static level of a means that triggering the initialisation of this class and executing its static initialiser can only trigger the initialisation or assume the invariant of classes with a static level less than or equal to a. This means that the execution of a statement with static level a does not lead to assuming the invariant of classes with a static level strictly greater than a. Then, at a program point where the invariant of a class `A` with static level a is violated, everything having a static level strictly smaller than a won't produce any problems because it does not rely on `A`'s invariant (and invariants of other classes having static level $\geq a$). Two such points, at which the invariant of class `A` might not hold are inside an `open A`/`close A` block and inside `A`'s static initialiser.

```
// open A
// only statements with
// static level < a allowed
// close A
```

```
public class A {
   static{
// only statements with
// static level < a allowed
```

Now, we will explain the static level constraints. These constraints provide inequalities for all statements, methods, constructors, and classes. They have to be resolved to get valid static levels. We will check in Viper as described in Section 5.3 if a static level assignment is valid. If no valid assignment of static levels under the given constraints is found, the program will be rejected.

The static level of `open A` and `open A using` p equals the static level of class `A` because these instructions assume `A`'s invariant. The static level of `close A` and `close A establishing` p are set to 0.

The static level of all statements not having any use of a class is set to 0, because they neither assume a class invariant nor do they trigger any class initialisation.

Let s be a statement that is neither a method call nor does it create an object, but it reads from or writes to static variables. Because accessing a static field might trigger the initialisation of the class that declares it, executing s might lead to the execution of the code inside its static initialiser, which might assume some invariants. This might be problematic if the statement is inside an `open`/`close` block. So, the static level of the class' static initialiser must be taken into account. The static level of s is at least as big as the static level of all classes whose static variables it accesses if it cannot be proven that the initialisation of this class is ongoing or already finished (and thus won't be triggered any more).

However, for all these classes whose static variables s accesses, it actually can be proven, that s does not trigger its initialisation inside an `open`/`close` block[1]. In the presented methodology, each class receives the permissions to the static fields it declares at the beginning of its static initialiser. To enable reads from or writes to its static fields outside this static initialiser, these permissions must be transferred into its static class invariant because a static initialiser cannot leak permissions otherwise. Inside the static initialiser, we know that the initialisation of this class is already ongoing (and thus will not be triggered any more). Outside the static initialiser, a static variable can only be accessed if the necessary permissions are held. Since initially these permissions were put into the class invariant of that class, they can only be received if the class was previously opened at least once. And the methodology only allows opening a class after its initialisation. Thus, if a statement contains reads from or writes to a static variable, it can be proven that the initialisation of all the classes whose static variables it accesses is ongoing or has already finished. That is why the static level of s is set to 0.

Let s be a method call to the static method `A.m()` (a call to a constructor of class `A`, so the statement `new A()`, is treated like a call to a static method). The

[1] s can trigger the initialisation of some classes. However, in a successfully verified program, such a statement s can only be the first statement within the `open`/`close` block, as explained later. In this case, the initialisation can take place before the `open` statement. This means that it happens outside the `open`/`close` block, which is not problematic. No class is initialised between the `open` statement and s.

static level of *s* is greater than or equal to the static level of the method `A.m()`. How static levels of methods are defined is described below. The reason for this is that if the code inside a method assumes some invariant, then calling this method will also lead to assuming this invariant.

The static level of *s* is additionally greater than or equal to the static level of class `A` if we cannot prove that executing *s* does not trigger `A`'s initialisation. This is because calling `A.m()` is a use of class `A` and thus may trigger the execution of `A`'s static initialiser. So, its level must be taken into account. The static level of a method call is independent of the methods' arguments because they have a static level of 0.

Let *s* be a method call to a non-static method *o*`.m()`, where the object *o* has static type `A`. The static level of *s* is greater than or equal to the static level of `m()` defined in class `A`. In contrast to static method calls, the static level of *s* is independent of the static level of class `A`. This is because, for the method call to succeed, *o* cannot be `null`. Let *o* be an object of dynamic type `B`, for some subtype `B` of `A`. The object *o* had to be created via `new B()`, which is a use of class `B` and might trigger `B`'s initialisation. And since `A` is a superclass of `B`, it must be initialised before `B`. So the call *o*`.m()`, for a non-null object, never triggers `A`'s initialisation.

The static level of a static method is the maximum over the static levels of all statements it contains.

The static level of a non-static method is greater than or equal to the maximum over the static levels of all statements it contains. An additional constraint comes from behavioural subtyping. If the method overrides another method, its static level must be less than or equal to the static level of the method it overrides.

Let class `B` extend class `A` and both define a method `m()`. The previous constraint is necessary because if we call a non-static method `o.m()` on some an object `o` with static type `A`, the method `m()` in class `B` might be executed (if `o` has dynamic type `B`). However, since statically we only know for sure `o`'s static type, the static level of this method call statement is set to the static level of method `m()` in class `A`, which corresponds to `o`'s static type. So, since instead of `m()` from class `A`, the overriding method might be called, this method can't have a static level greater than `m()`'s in class `A`. By the same reasoning, overriding methods of subtypes can have weaker preconditions and stronger postconditions than their corresponding supertype methods. Even though the grammar does not support object creation or subclassing, it can be easily extended to do so.

The static level of a static initialiser in class `A` must be strictly greater than the static levels of all statements it contains (using the knowledge that no statement there might trigger `A`'s initialisation, since it is already ongoing) and it is also greater than or equal to the static level of a class it extends, if there is one.

The reasoning behind the strict inequality in the first condition is that inside a static initialiser of class `A`, the invariant of `A` does not hold. So, no statement is allowed to assume it. Additionally, the strict inequality forbids cyclical static initialisation dependencies. It is not possible to assign valid static levels if two static initialisers mutually depend on each other. This is exemplified in Listing 5.3.

```
class A {
// Inv_A: acc(a)
   static char a = 'a';
   static {
      // open B
      a = B.b;
      // close B
   }
}
```

```
class B {
// Inv_B: acc(b)
   static char b = 'b';
   static {
      // open A
      b = A.a;
      // close A
   }
}
```

Listing 5.3: Let SL_A and SL_B refer to `A`'s respectively `B`'s hypothetical static levels. The instruction `open B` has static level SL_B. Since it appears inside `A`'s static initialiser, $SL_A > SL_B$ must hold. The same argument for `B`'s static initialiser gives the constraint $SL_B > SL_A$. It can be seen, that in this example, no valid assignment of static levels is possible.

The second constraint (which says that the static level of a superclass must be less than or equal to the static level of the subclass) follows from the property that the initialisation of superclasses is triggered before the initialisation of the classes that extend them. A statement s, that has a use of class `A`, can be viewed as a statement that not only has a use of `A`, but also a use of its superclass. Therefore, executing s must take into account the static level of the superclass. Strict inequality between the static level of a subclass and the class it extends is not required. This is because the initialisation of the class and its superclass run sequentially, and the superclass does not see a partially initialised state of the subclass, so, it is different from triggering the initialisation of the superclass from within the static initialiser of the superclass.

Given a valid static level assignment, the restriction that method calls are not allowed inside static initialisers or `open`/`close` blocks can now be lifted. Static levels not only rule out cyclical initialisation dependencies. Instructions inside an `open A`/`close A` block are allowed if their static level is strictly less than the static level of class `A`. A direct consequence of this is that an invariant can never be opened twice without being closed in between.

An invariant can only be established after class' initialisation As in the previous chapter, we are only allowed to assume an invariant of a class after it is established. The reason is again that the invariant is established by its static initialiser, so we cannot assume that an invariant holds before the static initialiser runs.

Before, the language semantics guaranteed that a thread different from `A`'s initialisation thread, for some class `A`, can only access `A`'s static variables when `A`'s initialisation state is `init`. Here, however, we are using Java's language semantics. A thread is allowed to read static variables declared in class `A` not only if `A`'s initialisation state is `init` but also if it is `ongoing` (by the same thread). As was shown in Listing 1.1, in Java, it is possible to read uninitialised variables. Thus, in an arbitrary program, a statement outside `A`'s static initialiser, which accesses a static variable or calls a method (a constructor is treated like a static method) defined in class `A`, cannot be taken as proof that `A` is initialised.

However, in a program with valid static level assignments, if a method or a static initialiser different from `A`'s contain a statement s, which has a use of class `A`, and they contain an `open A` statement, then it holds that the initialisation of class `A` is completed directly before the execution of s. The same applies to `open A using` p but is omitted here for presentation purposes. Then, we can use s to prove that opening `A` is sound. This holds because of static level constraints.

The reasoning is the following: when an `open A` statement is inside a method or a static initialiser, the static level of this class member must be at least as big as `A`'s static level. Therefore, no such method can be called and no such static initialiser can be triggered from inside `A`'s static initialiser. Also, no method or constructor called or triggered from inside `A`'s static initialiser is allowed to call or trigger this class member. Moreover, `open A` cannot be placed inside `A`'s static initialiser. Thus, since this class member cannot be executed while `A`'s initialiser is running, and we are dealing with sequential programs, it must run after the initialisation is completed.

That is why if a class member contains `open A` and a statement s that uses `A`, we know that if all static level assignments are valid, `A`'s initialisation must be completed before s is executed.

If a statement s can be seen as a proof of initialisation depends on its relative position to the `open A` statement. The first option for s being a proof of initialisation is if s lies before `open A`, and the second option is that s is the first (not ghost code) statement inside the `open A`/`close A` block. Again, if s is the first instruction after `open A`, since `open A` is just an annotation, it can be executed directly before the execution of s and after the initialisation of `A`.

Note that the remaining statements inside the block are not taken into consideration. Why this is the case is shown in Listing 5.4. Inside the `main()`

```
public class B {                    public class C {
// inv_B: x >= 0                    // inv_C: true
   int x = 0;                          static {
                                          // open B
   static void main() {                   B.x = 3 * B.x;
      // open B                           // close B
      B.x = B.x - 5;   // (1)          }
      C.f();
      B.x = B.x + 5;   // (2)          public static void f() {
      // close B                          return;
   }                                   }
}                                   }
```

Listing 5.4: The instruction `C.f()` triggers the initialisation of class `C` inside the `open B`/`close B` block.

method, there is an `open B`/`close B` block. Since the first instruction inside the block accesses a static variable defined in class `B`, we can assume that `B` is initialised, and open `B`. Inside the block, the statement (1) breaks `B`'s invariant. Because `C` is not initialised yet, the statement `C.f()` triggers its initialisation. This causes the static initialiser of `C` to assume a violated invariant. In the end, `B.b` holds the value `-10`.

It would be unsound to assume that `C`'s initialisation happened before `open B` and thus before the execution of `B.x = B.x - 5`. We see that this would result in `B.b` holding a different value than in a correct execution.

Now, since we cannot prove that `C` is initialised, the static level of `C.f()` is not only greater than or equal to the static level of the method `f()` in `C` (which is 0) but also greater than or equal to `C`'s static level. Since the `open B` statement is inside `C`'s static initialiser, its static level is strictly greater than `B`'s. Thus, `C.f()` cannot be placed inside this `open B`/`close B` block. Note that if `C` were already initialised before the `open B` instruction, our methodology would allow placing this statement inside an `open B`/`close B` block.

In this methodology, there is also another way to prove that a class is initialised. In this semantics, calling a method `A.m()` defined in class `A` triggers `A`'s initialisation, so `A`'s initialisation state has definitely started before this method is executed. If additionally `A.m()` contains an `open A`/`close A` block inside its body, then it can be proven that `A` is initialised before the execution of this method. Again, this only holds if a valid static level assignment exists for the program. We know that the static level of `A.m()` is at least as big as `A`'s static level since the method's body contains an `open A` statement. As a result, from inside the static initialiser, we cannot call `A.m()`. Thus, the call to `A.m()` must happen after `A`'s initialisation. As a result, in Listing 5.4, it even holds that `A` is initialised before the execution of the `main()` method.

Consistency criteria This paragraph lists the criteria that must apply for an annotated Java program so that it can be encoded to Viper according to the translation in Section 5.3.

- Each class must contain a self-framing (explained in Section 2.4) static class invariant.
- A OUAT predicate w.r.t class `A` must be pure (i.e., not contain permissions to any resources) and framed by `A`'s class invariant, but it does not have to be self-framing on its own.
- A history constraint of class `A` must be framed by `A`'s class invariant.
- If a OUAT predicate w.r.t. `B` is claimed to be established, then `B` must contain a history constraint.
- The `open`/`close` instructions form a block. Thus, each `open A` must be followed by a matching `close A` instruction.
- Each static initialiser, method, and constructor must be annotated with a static level.

Examples Here, we present three code examples written in Java and annotated with our methodology.

The first example in Listing 5.5 we provide to demonstrate static level assignments. In this example, class `A` establishes its invariant under the assumption of `B`'s invariant. The problem in this example is that inside the method `B.m()`, the method `A.n()`, which might trigger the initialisation of `A`, is called while `B`'s invariant is violated. As a result, the static initialiser of class `A` might assume `B`'s invariant while it does not hold. Our methodology rejects this example due to static levels.

Let SL_A, SL_B, $SL_{A,n}$, and $SL_{B,m}$ denote the static levels of classes `A`, and `B` and methods `A.n()` and `B.m()` respectively. As explained above, the statements `b = 3`, `A.a = B.b`, `B.b--` and `B.b++` all have static level 0. The statement `open B`, since it assumes `B`'s invariant, has static level SL_B. Thus, the static initialiser of class `A`, and therefore also class `A`, have a static level strictly larger than SL_B ($SL_A > SL_B$). By the same reasoning, $SL_{B,m} \geq SL_B$ must hold. Since in this example, it cannot be proven that `A` is initialised before the `open A`/`close A` block in method `m()`, the statement `A.n()` has static level $SL_{A,n} \geq \max(SL_{A,n}, SL_A) > SL_B$. Because all instructions inside an `open B`/`close B` block must have a static level strictly smaller than SL_B, the statement `A.n()` cannot be placed there and the verification fails.

The example in Listing 5.6 is a simplified version of the example introduced and explained in Section 3.2. Because we are only looking at sequential

```java
class A {
// inv_A: acc(A.a) && A.a == 3
   static int a;
   static {
      // open B
      A.a = B.b;
      // close B
   }

   public static void n() {}
}

class B{
// inv_B: acc(B.b) && B.b == 3
   static int b = 3;

   public static void m() {
      // open B
      B.b--;
      A.n();
      B.b++;
      // close B
   }
}
```

Listing 5.5: Code example whose verification fails, because A.n() cannot be placed inside an open B/close B block.

programs, we removed the synchronisation and modified the atomic integer to a regular integer. For the sake of readability, the invariant of class DataTypeBase is defined here using mathematical notation.

$$Inv_{DataTypeBase}: \texttt{acc(counter)} \land \texttt{acc(annotationMap)} \land \texttt{counter} \geq 0 \land \\ (\forall c \in \texttt{annotationMap}\ \texttt{acc}(\texttt{getAnnotation}(c))\ \land \texttt{getAnnotation}(c) < \texttt{counter}) \land \\ (\forall c_1, c_2 \in \texttt{annotationMap}\ c_1 \neq c_2 \implies \texttt{getAnnotation}(c_1) \neq \texttt{getAnnotation}(c_2))$$

During the static initialisation of DataTypeBase only static variables of its class are accessed, and DataTypeBase does not extend any class. Thus, its static level can be set to any positive number. We set its static level to 1. Since the methods addAnnotation() and getAnnotation() are inside the class DataTypeBase, its invariant can be safely assumed. Inside both open DataTypeBase/close DataTypeBase blocks, all instructions have static level 0, so they are also allowed. Because both methods open DataTypeBase, their static level must be at least 1. Inside the method addAnnotation(), a class is assigned the lowest available annotation index and this mapping is added

```
private static class DataTypeBase {
// SL_DataTypeBase: 1
// Inv_DataTypeBase
        private static final HashMap < Class<?>, Integer > annotationMap =
            new HashMap < Class<?>, Integer > ();
            private static int counter =  0;

        public void addAnnotation(Class<?> ann) {
        // SL_DataTypeBase_addAnnotation: 1
            // open DataTypeBase
            Integer i = annotationMap.get(ann.getClass());
            if (i == null) {
                annotationMap.put(ann.getClass(), new Integer(counter));
                counter++;
            }
            // close DataTypeBase
        }

        public Integer getAnnotation(Class<?> c) {
        // SL_DataTypeBase_getAnnotation: 1
            // open DataTypeBase
            Integer i = annotationMap.get(c);
            // close annotationMap.get(c)
            return i;
        }
}
```

Listing 5.6: A sequential version of the class DataTypeBase from Section 3.2 encoded using our obtained methodology

into the annotationMap. The method getAnnotation() assumes the invariant of DataTypeBase and returns the annotation index corresponding to the input class.

The last example in Listing 5.7 is a design pattern called Singleton [33]. A class using the Singleton pattern defines one static field called instance. This static field holds the only instance of this class. To achieve that at most one instance of the class exists, the constructor is hidden. There are two possibilities to implement this pattern. In the first one, instance is already initialised during class initialisation. Here, we present the second one: the static field instance is not initialised until getInstance() is called.

The invariant of this class contains the permissions to the static field instance. The class contains the history constraint, which specifies that once the field instance is not null, it will not be modified. When getInstance() is called for the first time, a freshly created Sigleton object is assigned to

```
class Singleton {
// SL_Singleton: 1
// Inv_Singleton: acc(instance)
// History_Singleton: old[start](instance) != null ==>
//                     old[start](instance) == old[end](instance)

        private static Ref instance = null;

        private Singleton() {} // SL_Singleton_cons: 0


        public static Ref getInstance() {
            // SL_Singleton_getInstance: 1

            // open Singleton
            if (Singleton.instance == null) then
                Singleton.instance = new Singleton()
            Ref instance = Singleton.instance;
            // close Singleton
            return instance
        }
}
```

Listing 5.7: Class implementing the Singleton pattern encoded using our methodology.

`instance`. This is done inside an `open Sigleton`/`close Sigleton` block. The constructor is empty, so its static level can be set to 0. The static level of the `Singleton` class can be set to a number higher than 0, for example to 1, then a `Singleton` instance can be created inside the `open Singleton`/`close Singleton` block. This example verifies successfully.

5.3 Viper encoding

In this section, we explain the translation of an annotated and type-checked Java program into a Viper program. The core idea is to translate all Java methods and static initialisers as Viper methods and the static class invariants as Viper predicates. At the end of each static initialiser, we check using the `fold` statement if the class invariant was established. On `open A`, the invariant of class `A` is assumed using the `inhale` and `unfold` statements, in between it might be violated, and on `close A` it is checked to hold again (`fold` and `exhale`). Static level checks precede the Java statements. They make sure that the static level assignment is valid and that no cyclical initialisation dependencies between classes exist. To not run into naming issues, we assume that no signature of a method or constructor contains an underscore.

A restriction that is made inside this section is that OUAT predicates can only be established inside a static initialiser and a class can establish at most one OUAT predicate w.r.t. one particular class (but might establish multiple OUAT predicates w.r.t. different classes). With this restriction, an injective mapping from pairs of classes to established OUAT predicates can be defined. This makes the translation easier.

A new verification statement `open A using` p `from B` is introduced here, which replaces the statement `open A using` p. The new statement combines opening `B`, learning that the predicate was established, closing `B` and then using the statement `open A using` p to assume p and `A`'s class invariant and assert that the OUAT predicate was established into one. It can only be used if `B` established a OUAT predicate w.r.t. `A` of which p is a logical consequence. Writing `open A using` p `from B` makes it clear which OUAT predicate was intended.

Now, the precise translation is explained. For most bullet points, a detailed paragraph is provided below. For each class `A`, the following must be created:

- For all static fields of class `A`, a Viper field is added. E.g. `field a : Int`, if `A` contains an integer field called `a`.
- Abstract `function A() : Ref`. It returns a reference that represents class `A`. A static field `a` of class `A` can be accessed by `A().a` if the necessary permissions are held.
- Abstract `function A_init() : Bool`, which returns `true` if the initialisation of class `A` has started or is already finished (and thus won't be triggered any more).
- `function SL_A() : Int {`s`}`, where s corresponds to `A`'s static level (a non-negative integer), defined according to Section 5.2.
- If the static initialiser of `A` does not establish any OUAT predicate, create `predicate invariant_A()` with `A`'s invariant (a self-framing predicate) as its body.
- Else, for each class `B`, such that `A`'s static initialiser establishes a OUAT predicate p with respect to `B` (this holds, if the annotation `close B establishing` p can be found in `A`'s static initialiser):
  - Create abstract `function ouat_A_B() : Bool`. It represents the assertion that specifies whether the OUAT predicate was established. The function returns `true` if the OUAT predicate with respect to class `B` was established by `A`'s static initialiser.
  - Create the macro `define OUAT_A_B (`p`)`, which replaces `OUAT_A_B` by the actual contents of the predicate.

  - Create `method check_stability_OUAT_A_B()`, which checks the stability of `A`'s OUAT predicate w.r.t `B`'s history constraint. The body of this method is provided in a paragraph below.
  - Once, create `predicate invariant_A()` whose body is `A`'s invariant. There, replace all assertions corresponding to OUAT predicates `OUAT(`p`)` with the corresponding functions `ouat_A_B()`.

- If `A` contains a history constraint, create a macro with signature `define history_A(start, end)` that acts as a placeholder for the actual history constraint of class `A`. Create `method check_history_A()`, a method, that checks if this history constraint is reflexive and transitive. This method is defined below.
- `method initialise_A()` contains all static variable initialisations of class `A` and the code from `A`'s static initialiser. The details are provided in a paragraph below.
- Create `method A_m(...)` for each (static or non-static) method `m` of class `A`. The exact translation of the method is described below. Additionally, create the function `function SL_A_m() : Int {`s`}` returning `A.m()`'s static level s.
- For each non-static method `m()` in class `A` that directly overwrites the method `m()` defined in `A`'s subclass `B`, create `method check_A_B_m()`. This method checks if `m()` adheres to the rules of behavioural subtyping.
- If `A` contains a constructor, create `method constructor_A() returns (A_instance:Ref)`. Without loss of generality, we assume that `A` contains only one constructor without parameters. Additionally, create the `function SL_A_con() : Int {`s`}`, which returns the static level s of `A`'s constructor.

Static level checks The integer sequence `SL_bound_stack` is used to mechanize static level checks. It can be viewed as a stack, on which we push and pop static levels during the execution. The element inserted last into `SL_bound_stack` is referred to as the top of `SL_bound_stack` and can be read using `SL_bound`. The following macro, once defined in the program, sets `SL_bound` as a placeholder for the top of `SL_bound_stack`.

```
define SL_bound(SL_bound_stack[|SL_bound_stack|-1])
```

A program invariant is that `SL_bound` corresponds to an integer, such that all statements must have a static level which is strictly less than the current value of `SL_bound`. During program execution, the value of `SL_bound` might change. Each class member initialises `SL_bound_stack` at the beginning of the corresponding Viper method. Since inside the static initialiser of some class `A` all statements must have a static level strictly smaller than `SL_A()`, in the beginning of the static initialiser, the integer `SL_A()` is pushed onto the empty stack `SL_bound_stack`. In a method `m()` (constructors are treated like methods) defined in class `A`, all statements must have a static level smaller than or equal to `SL_A_m()`. Since for all integers a, b, $a \leq b$ is equivalent to $a < b + 1$, in the method `m()`, `SL_bound` is initialised to `SL_A_m()+1`.

The sequence `SL_bound_stack` is extended upon entry of each `open/close` block. Inside the `open A/close A` block, additionally, each statement must have a static level strictly smaller than `SL_A()`, so strictly smaller than min(`SL_A()`, `SL_bound`). Because, like all statements, `open A` (which has static level `SL_A()`) must have a static level strictly smaller than `SL_bound`, the integer min(`SL_A()`, `SL_bound`) equals `SL_A()`. Therefore, it is sufficient to push `SL_A()` onto `SL_bound_stack`. On `close A`, the top of `SL_bound_stack` is popped, because the additional static level constraint no longer holds.

On `close`, the old static level bound must be recovered. Because `open/close` blocks can be nested, and the depth of `open/close` blocks is only limited by the number of classes in the program, the static level bound must be represented by a sequence that can be extended an arbitrary number of times.

function A_init() This abstract function returns a boolean that indicates if the initialisation of `A` *has started* and thus won't be triggered any more. It is used to prove that the execution of any statement will not trigger `A`'s initialisation if `A_init()` returns `true`. `A_init()` is assumed to return `true` after each statement that has a use of class `A`.

Another usage of this function is that one is only allowed to assume the invariant of class `A` after the initialisation of `A` *is completed*. In a successfully verified program (concerning the methodology defined in this chapter), if a method or a static initialiser contains the statement `open A` and `A_init()` returns `true`, then it holds that the initialisation of class `A` is completed at that point. The reasoning behind this is explained in Section 5.2. Thus, if `A_init()` holds in a class member containing `open A`, then `A`'s initialisation did not only start but is completed. Thus, it is sufficient to check if `A_init()` returns `true` before assuming `A`'s invariant.

method check_history_A() The macro `history_A(`*start*`,` *end*`)` is a placeholder for `A`'s history constraint, which relates the state at the label *start* to the state at the label *end*. Each history constraint must be reflexive and transitive. By definition, the history constraint P_A is reflexive if $P_A(l, l)$ holds in all states l. It is transitive, if for all states l_1, l_2, l_3, $P_A(l_1, l_2)$ and $P_A(l_2, l_3)$ implies $P_A(l_1, l_3)$.

The code below checks both properties of the history constraint of class `A`. Note that the states l_1 and l_4 are equal since the verification statement `label` *l* does not modify the state.

```
method check_history_A() {
    inhale invariant_A()
    unfold invariant_A()
    label l1
    label l4
    assert history_A(l1, l4)            // reflexivity
    fold invariant_A()
    exhale invariant_A()


    inhale invariant_A()
    unfold invariant_A()
    label l2
    fold invariant_A()
    exhale invariant_A()


    inhale invariant_A()
    unfold invariant_A()
    label l3
    assume history_A(l1, l2) && history_A(l2, l3)
    assert history_A(l1, l3)            // transitivity
}
```

method check_stability_OUAT_A_B() `OUAT_A_B(`*L*`)` refers to the OUAT predicate at the label *L*. Therefore, it is the same as `A`'s OUAT predicate with respect to `B` with the exception that all field accesses and calls to heap-dependent functions, e.g. `o.f`, are replaced by expressions that refer to the heap access of that field or heap-dependent function at label *L*, i.e. `old[`*L*`](o.f)`.

As explained in Section 4.2, a OUAT predicate with respect to class `A` must be stable with respect to `A`'s history constraint such that it cannot be invalidated once it has been established. The definition of stability of `OUAT_A_B` with respect to `A`'s history constraint is that if `OUAT_A_B` holds in state *start* and the history constraint holds between *start* and *end*, then `OUAT_A_B` must hold in state *end*. This means, that whenever the history constraint of `B` holds, once `OUAT_A_B` is established, it cannot be invalidated. The code below verifies this property.

```
method check_stability_B_OUAT_A() {
    inhale invariant_B()
    unfold invariant_B()
    label start
    fold invariant_B()
    exhale invariant_B()
    inhale invariant_B()
    unfold invariant_B()
    label end
    assume history_B(start, end)
    assume OUAT_A(start)
    assert OUAT_A(end)
}
```

method initialise_A() This paragraph explains and provides the translation of the static initialisation logic. The encoding is explained first.

The meaning of the sequence SL_bound_stack is explained in the paragraph on static level checks. In the static initialiser of class `A`, it can be proven that the initialisation of class `A` has begun, so `A_init()` is inhaled. Moreover, in Java, a subclass is initialised after the class it extends. Thus, on execution of the static initialiser of class `A`, the initialisation of all its superclasses has already started (if there are no cyclical initialisation dependencies, it can be proven, that the initialisation actually is completed). That is why `C_init()` is inhaled for all classes `C` that `A` transitively extends. `assert SL_C() <= SL_A()` makes sure, that the static level constraint, that the static level of a subclass is greater than or equal to the static level of the superclass, holds. If `A` does not extend any class, these statements can be left out. The static initialiser receives full permissions to all static fields declared in its class at the beginning of its execution. The comment "initialisation code" is a placeholder for all static variable definitions of class `A` and the code from `A`'s static initialiser. These Java instructions are translated according to the paragraph below on translating statements into Viper. Since the static initialiser of class `A` must establish `A`'s invariant, in the end, we check if `invariant_A()` holds by folding the predicate.

The encoding of `method initialise_A()` now follows.

```
method initialise_A() {
    var SL_bound_stack : Seq[Int] := Seq(SL_A())

    inhale A_init()
    // for all classes C that A transitively extends:
    inhale C_init()
    // for class C that A directly extends:
    assert SL_C() <= SL_A()

    // for all static fields a of A:
    inhale acc(A().a)

    // initialisation code

    fold invariant_A()
}
```

method A_m(...) The method's arguments and return type are set according to Java's method definition. The encoding below is provided for a method without input and output parameters. The method might contain arbitrary pre- and post-conditions. They are translated into Viper using the `requires` and `ensures` keywords.

Inside `A_m()`, `A_init()` is inhaled, since calling a method declared in class `A` triggers `A`'s initialisation. Because superclasses are initialised before subclasses, `C_init()` is inhaled for all classes `C` transitively extended by `A` (if any). The meaning of the sequence SL_bound_stack is explained in the paragraph on static level checks. The instructions from method `m()` are translated according to the paragraph on translating statements.

```
method A_m() {
    // A_m might have a non-void return type and arguments
    var SL_bound_stack : Seq[Int] := Seq(SL_A_m() +1)

    inhale A_init()
    // for all classes C that A transitively extends:
    inhale C_init()

    // translated instructions from method m()
}
```

method check_A_B_m() The method below checks if `m()` adheres to the rules of behavioural subtyping.

From behavioural subtyping follows that overriding methods of subtypes may have weaker pre-conditions and stronger post-conditions than its corresponding supertype methods. Let `A` extend `B`. We check the previously stated conditions by invoking method `A_m()` inside the method `check_A_B_m()`, which has the same pre- and post-conditions as `B_m()`. p_{pre} and p_{post} denote the pre- respectively post-condition of method `m()` in class `B`. If verification succeeds, then `A_m()` must have a weaker precondition than p_{pre} and a stronger postcondition than p_{post}. The method `check_A_B_m()` additionally checks, if the static level of `A_m()` is smaller than or equal to the static level of method `B_m()`. The reasoning behind these checks is explained in Section 5.2 in the paragraph on static levels.

```
method check_A_B_m()
    requires p_pre
    ensures p_post
{
    assert SL_A_m() <= SL_B_m()
    A_m()
}
```

method constructor_A() The translation of a constructor is very similar to the translation of a static method. The only difference is that a constructor returns a newly created non-null instance. The translation is provided below.

```
method constructor_A() returns (A_instance:Ref)
ensures A_instance != null
{
    var SL_bound_stack : Seq[Int] := Seq(SL_A_cons() +1)

    A_instance := new()

    inhale A_init()
    // for all classes C that A transitively extends:
    inhale C_init()

    // translated instructions from A's constructor body
}
```

Translating opening and closing A First, the simple case is explained, which corresponds to the following annotated Java code:

```
// open A
instructions I
// close A
```

This paragraph explains the Viper code provided below. Let I be the statements within the `open A`/`close A` block. Our methodology says that assuming the static invariant of class `A` is only allowed if `A` is initialised. Because of that, `assert A_init()` is inserted before `A`'s invariant is inhaled. However, if the first statement of I (ignoring ghost code) contains a use of class `A`, the initialisation of `A` happens certainly before the instructions I. That's why instruction (1) in the code below is not necessary in that case. If the first statement of I contains a use of class `A`, we replace instruction (1) by `inhale A_init() && C_init()` for all classes `C` that `A` transitively extends. After the static level check succeeds and the new lowest static level is pushed onto the stack, `A`'s static class invariant is assumed using the `inhale` and `unfold` statements.

The instructions marked with (2) are only necessary if `A` contains a history constraint. They assert that the history constraint was not violated between the `open A` and `close A` instructions. The index at both labels is an integer guaranteed to be unique across the whole program, so no labels are duplicated. The first 6 instructions correspond to opening `A` and the last 5 instructions correspond to closing `A`. Instructions I' are to the instructions I modified according to the section on translating statements.

Before closing `A`, the most recently added static level is removed from `SL_bound_stack`. Folding and exhaling the invariant asserts that `A`'s invariant holds on `close A`.

```
assert A_init()                                        // (1)
assert SL_A() < SL_bound
SL_bound_stack := SL_bound_stack ++ Seq(SL_A())
inhale invariant_A()
unfold invariant_A()
label openA_index                                      // (2)

// instructions I'

label closeA_index                                     // (2)
assert history_A(openA_index, closeA_index)            // (2)
SL_bound_stack := SL_bound_stack[0..(|SL_bound_stack|-1)]
fold invariant_A()
exhale invariant_A()
```

Translating opening A using a OUAT predicate Next, the translation of `open A using` p `from B`, where p is a boolean expression, is explained. The code is presented below. The matching `close A` statement is translated exactly like shown in the previous paragraph, so its translation is left out here. Let I be the instructions that follow `open A using` p `from B` in the Java code.

`open A using` p `from B` is translated as first opening and closing `B` with no instruction in between. Because the functions corresponding to OUAT predicates are pure, the knowledge about its output gained from opening `B` still exists after closing `B`. The `open B` and `close B` instructions are translated as explained in the previous paragraph, however, since no instructions are in between, the history check and extending the `SL_bound_stack` are not necessary. Again, the statement marked with (1) in the code below is only necessary if the first statement of I does not contain a use of `B`. Otherwise, it is replaced by `inhale B_init() && C_init()` for all classes `C` that `B` transitively extends. From `open B`, we learn that `ouat_B_A()` returns `true`, which means that the OUAT predicate `OUAT_B_A` was established.

Then, `A` is opened. No `assert A_init()` and `assert SL_A() < SL_bound` checks are necessary because for the OUAT predicate w.r.t. `A` to be established by `B`, the invariant of `A` must be opened inside `B`'s static initialiser. And this can only be done if `A` is initialised at that point, and has a static level strictly smaller than `SL_B()`. The boolean expression p must be a logical consequence of the OUAT established by `B` with respect to `A`. Also, `B` must have `ouat_B_A()` in its invariant, which tells that it has established this predicate. If that all holds, p can be assumed to hold. From the consistency criteria follows that if there exists a class that has a OUAT predicate with respect to `A`, `A` must define a history constraint. So the statements, that are marked with (2) in the previous paragraph, are now necessary.

```
assert B_init()               // open B, (1)
assert SL_B() < SL_bound
inhale invariant_B()
```

```
unfold invariant_B()

fold invariant_B()          // close B
exhale invariant_B()

SL_bound_stack := SL_bound_stack ++ Seq(SL_A())
inhale invariant_A()
unfold invariant_A()
label openA_index
assert ouat_B_A() && (OUAT_B_A ==> p)
inhale p
```

Translating closing A establishing a OUAT predicate Finally, the translation of `close A establishing` p in `B`'s static initialiser, where p is a boolean expression, is explained. The translation of the corresponding `open A` equals the one explained in the paragraph on translation opening and closing `A`. The only difference between `close A establishing` p and a regular `close A` statement are the statements marked with $(*)$. Directly before closing `A`, the boolean expression p, that `B` claims to have established, is asserted to hold. If this assertion succeeds, the function `ouat_B_A()` is inhaled. This must be done because `B`'s invariant contains `ouat_B_A()`.

```
assert p                  // (*)
inhale ouat_B_A()         // (*)
label closeA_index
assert history_A(openA_index, closeA_index)
SL_lt := SL_lt[0..(|SL_lt|-1)]
fold invariant_A()
exhale invariant_A()
```

Translating statements Some Java statements are preceded by static level constraints, as explained in Section 5.2 in the paragraph on static levels. Moreover, after a statement, that has a use of some module `A`, `init_A()` can be assumed.

Each Java statement s is translated into Viper as follows:

- s is a method call to the static method `B.m(...)` (we treat the constructor like a static method):
  1. `assert (B_init() || SL_B() < SL_bound) && SL_B_m() < SL_bound`
  2. call `B_m()`
  3. `inhale B_init() && C_init()` for all classes `C` that `B` transitively extends

- *s* is a method call `a.m(...)` to a non-static method, where `a` has static type `B`:
  1. `assert SL_B_m() < SL_bound && a != null`
  2. call `B_m()`
- All other Java statements are translated directly into Viper without additional asserts and inhales surrounding them because they do not trigger the initialisation of any class and do not assume any invariants.

Chapter 6

Evaluation

In this chapter, we evaluate the methodology described in Chapter 5. To evaluate it and the Java to Viper encoding, we manually translated some annotated Java examples to Viper using the rules defined in Section 5.3 and verified the encoding there. Some of these code examples were presented in earlier sections of this thesis. All of them were verified successfully. The translated examples can be found on GitHub [17]. The corresponding annotated Java programs are provided as a comment at the beginning of each file. We discuss some interesting properties of the examples later in this chapter.

Table 6.1 includes the annotation and verification overhead and the verification time of all encoded examples. We obtained these benchmarks on

Name	Lines of code	Lines of annotation	LOC encoding	Verification time (s)
byte_cache (List. 3.4)	13	21	62	1.64
clients	8	18	71	1.7
cond_perm (List. 6.2)	22	33	65	1.2
cyclic_dep (List. 1.1)	8	12	48	0.94
Fib (List. 4.9)	17	25	90	2.04
inheritance	19	27	61	1.06
OUAT (List. 4.4)	14	27	114	1.72
service	15	24	61	1.54
singleton (List. 5.7)	10	17	73	1.32
SL_check	22	36	78	1.1

Table 6.1: This table contains the data corresponding to the examples on GitHub.

a machine running Windows 10, having an Intel(R) Core(TM) i7-8550U CPU processor and 16.0 GB of RAM. The encoding was verified using the Viper version 24.01 and Viper's verifier Silicon. The number of lines of annotation refers to the total number of lines of code of the examples annotated with our methodology for Java, including static level annotations. LOC encoding refers to the number of lines of Viper code. The verification time is averaged over five runs and was measured on Visual Studio Code. It can be seen in Table 6.1 that the verification time is low in all encoded examples.

Limitation In this thesis, we have defined static level constraints, which once resolved, provide a valid static level assignment. Static levels allow certain method calls inside a static initialiser or an `open`/`close` block, without the method having to specify which invariants it assumes and which class's initialisation it might trigger. They resolve the challenge that while an invariant is broken, no code can be executed that relies on the validity of this invariant. The only restriction we make on instructions inside a static initialiser or an `open`/`close` block comes from static level constraints. Inside a static initialiser, we do not allow calls to functions that rely on the invariant of the static initialiser. That prevents us from reading uninitialised data. In some cases, this might be restrictive, because the other class or the called method might only rely on already initialised parts of the class invariant. This is exemplified in Listing 6.1. The method `f()` only relies on the static

```java
public class A {
// inv_A: acc(i) && i == 6 && acc(j) && j == 6
   static int i = 6;
   static int j = A.f(); // 6

   public static int f() {
      return A.i;
   }
}
```

Listing 6.1: This example cannot be verified in our methodology. The function `f()`, which relies on `A`'s invariant, cannot be called from within `A`'s static initialiser.

variable `A.i`, which is placed in the code above the function call and thus `acc(i) && i == 6` is already established at the time when `f()` is called. Therefore, no uninitialised data is read in this case and `f()` could assume the first part of `A`'s static class invariant. However, this example cannot be verified in our methodology, because we make no distinction on which parts of the invariant are assumed.

Challenging examples The file `cyclic_dep.vpr` shows an example of two classes having a cyclical initialisation dependency. Static levels provide a way to rule out all programs containing cyclical dependencies. Inside a static initialiser, all instructions must have a static level strictly smaller than the class's level. Because of this constraint, in a program with cyclical initialisation dependencies, one cannot generate correct static levels. Independent of the static level assignment, given a cyclical dependency, at least one static level constraint will be violated, and this constraint will be detected by our encoding in Viper.

Another program rejected by our methodology is presented in the file `SL_check.vpr`. There, a method is called inside an `open B`/`close B` block, which could trigger the initialisation of a class that assumes `B`'s invariant inside its static initialiser. This would lead to re-opening an already opened invariant. Static levels enforce that an invariant is never assumed while it may be violated by only allowing instructions with a static level strictly smaller than `B`'s inside an `open B`/`close B` block.

In a program with no static initialisers or static class invariants, all static levels can be set to 0. This is, because only `open`/`close` blocks and static initialisers introduce strict inequality constraints. The other static level constraints provide no strict inequalities. Without strict inequalities, all static levels can be the same, so can be set to 0 and all static level checks become trivial.

We added support to our methodologies, such that in both of them, classes can establish properties not only concerning their own static fields but also static fields declared in other classes. Methods can also establish properties related to static fields. We call these properties OUAT predicates. An encoded example can be found in the file `OUAT.vpr`, where the class `B` establishes a OUAT predicate with respect to class `A`.

The file `cond_perm.vpr` presents an encoding of two classes, where an invariant even contains permissions to a static field defined in the other class. The corresponding code example is shown in Listing 6.2. `B`'s static initialiser negates `A.b`, writes to `A.m` and transfers the full permission to `A.m` inside its invariant if `A.b` was previously `true`. That way, `B` can hold write permissions to `A.m` inside its invariant.

All the examples introduced in Chapter 3 (except for registering native code, which is out of scope for this thesis) can successfully be verified using the methodology for Java in Viper concerning the specifications explained in that chapter. Thus, this methodology seems to apply to all typical applications of static fields found in Java programs.

```
class A{
// SL_A: 0
// inv_A: acc(A().b) && (A().b
// ==> acc(A().m) && A().m == 42)

   static bool b = true;
   static int m = 42;


}
```

```
class B {
// SL_B: 1
// inv_B: acc(B().b) && (B().b
// ==> acc(A().m) && A().m == 43)

   static bool b;
   static {
      // open A
      B.b = A.b;
      if(B.b) {
         A.b = false;
         A.m++;
      }
     // close A
   }
}
```

Listing 6.2: Class invariant of class B contains permissions to static fields declared in class A.

Chapter 7

Conclusion

In this thesis, we collected and analysed the uses of static initialisers in large code collections like the Java standard library. For each use, we presented examples that challenge the state of the art. Moreover, we found that cyclical static initialisation dependencies are very problematic in practice and are often indicative of bugs.

We examined how static initialisation is implemented in various programming languages, with a particular focus on Java. Additionally, we analysed the implementations of static initialisation in various programming languages based on two features, namely whether they do lazy or eager initialisation and whether cyclical initialisation dependencies are rejected. We discovered two properties, Property 1 and Property 2, that the majority of the programming languages possess. Based on these two properties, we defined the programming language BAUM. This language imposes minimal restrictions on static initialisation, thus it can be used to model the static initialisation procedure of many programming languages.

Our second language is based on Java and thus uses lazy initialisation. For each language, we presented a modular verification methodology for proving static class invariants, which are expressed in a specification language that extends separation logic. Both languages not only allow static initialisers to establish properties concerning the static fields defined in their class, but static initialisers and methods can establish properties concerning all static fields in the program, which was not supported in previous work. Apart from registering a callback, the methodology for Java covers all the typical uses of static initialisation that we found before. The methodologies make the restriction that there are no cyclical dependencies in the program. In practice, we find this restriction to not be problematic at all, given that, in our experience, cycles are always indicative of latent bugs. We believe that both methodologies are sound. A formal proof of soundness is left for future work.

In the end, we provided an encoding from an annotated Java program into Viper. Using this encoding, we implemented some test cases into Viper and evaluated our approach. All translated examples were verified correctly.

Future work In this thesis, we are only looking at sequential programs. In the future, we will also consider concurrent programs. For the class invariants, we used a specification language that extends separation logic, such that the extension to concurrent programs should be feasible.

In this thesis, we developed a verification methodology for Java, a language that uses lazy initialisation. In the future, one could extend the general methodology for the language Baum to a methodology catered to languages that use eager initialisation.

Another possible direction consists of automating the translation from an annotated Java program to a Viper program using the translation steps defined in Section 5.3.

To facilitate the encoding, we introduced the restriction that only static initialisers can establish OUAT predicates. Moreover, each class can establish at most one OUAT predicate with respect to a different class. This restriction can be dropped by introducing a different mapping between the OUAT predicates and abstract functions in Viper.

A design choice that we made in the methodology was to treat the `open` and `close` instructions and all instructions enclosed by them as a block. That way, the rule that each `open` must have a matching `close` is directly enforced during type checking. However, one could consider the opening and closing not as a block but as two separate instructions. The semantics of these instructions would stay the same. A code example is shown in Listing 7.1. In the example on the left, there are two `close A` statements, however, `A`'s invariant is opened only once. Therefore, they do not form a block. But still, on each execution path, the `open A` statement is eventually followed by `close A`. This example could not be verified directly by our methodology. However, on the right, we show how the separate `open` and `close` instructions can be rewritten into a block form. Using `open` and `close` as separate instructions and not as a block makes the check that all opened invariants are eventually closed less straightforward. It must be checked that, on each execution path, an `open` instruction is followed by a corresponding `close`. This check can be implemented using obligations [14]. Obligations are separation logic resources that must be consumed before the end of the program. On `open A`, an obligation specific to `A` is created. This obligation specifies that `A`'s invariant must eventually be closed. `close A` can only be executed if the obligation corresponding to `open A` is held, and it consumes this obligation. As a result, it is not possible to close an invariant twice without opening it in between, and each `open A` is eventually followed by a `close A`.

```
class A {
// inv_A: acc(A.a) && A.a >= 0
//        && acc(A.b)
   static int a = 0;
   static bool b = false;

   static int m(int d) {
      // open A
      if(A.b) {
        A.a = d*d;
        // close A
        return 7;
      }
      else {
        A.a = A.a + d*d;
        // close A
        return 0;
      }
   }
}
```

```
static int m(int d) {
   // open A
   int temp = A.b;
   if(temp) {
     A.a = d*d;
   }
   else {
     A.a = A.a + d*d;
   }
   // close A

   if(temp) {
     return 7;
   }
   else {
     return 0;
   }
}
```

Listing 7.1: The code example on the left shows an extension of our methodology, where `open A` and `close A` instructions are used as two separate instructions and not as a block. The code example on the right shows how the method `m()` can be rewritten, such that it can be verified in our methodology.

Bibliography

[1] Adoptopenjdk. https://github.com/AdoptOpenJDK/openjdk-jdk11/tree/master/src/java.base/share/classes/java/lang. Accessed: 2024-02-19.

[2] Ca1810: Initialize reference type static fields inline. https://learn.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca1810. Accessed: 2024-02-04.

[3] Crate lazy_init. https://docs.rs/lazy-init/latest/lazy_init/. Accessed: 2024-02-08.

[4] The go programming language specification. https://go.dev/ref/spec#Program_initialization_and_execution. Accessed: 2023-09-17.

[5] Initialization - cppreference.com. https://en.cppreference.com/w/cpp/language/initialization. Accessed: 2024-01-28.

[6] Java api. https://github.com/square/javapoet/issues/637. Accessed: 2024-03-04.

[7] Lazy vals initialization. https://docs.scala-lang.org/scala3/reference/changed-features/lazy-vals-init.html#. Accessed: 2024-02-08.

[8] The netty project. https://github.com/netty/netty/issues/5720. Accessed: 2024-03-04.

[9] The rust reference - static items. https://doc.rust-lang.org/reference/items/static-items.html. Accessed: 2024-02-08.

[10] Scala 2 bug tracker. https://github.com/scala/bug/issues/7646?orig=1. Accessed: 2024-03-04.

[11] Static constructors (c# programming guide). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-constructors. Accessed: 2024-02-04.

[12] Static initialization order fiasco. https://en.cppreference.com/w/cpp/language/siof. Accessed: 2023-09-25.

[13] Egon Börger and Wolfram Schulte. Initialization problems for java? *Softw. Concepts Tools*, 19(4):175–178, 2000.

[14] Pontus Boström and Peter Müller. Modular verification of finite blocking in non-terminating programs. 07 2015.

[15] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, pages 55–72, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[16] Go developers. Go standard library. https://github.com/golang/go/tree/master/src. Accessed: 2024-03-04.

[17] Patricia Firlejczyk. Verifying static initialisation. https://github.com/pfirlejczyk/Verifying-static-initialisation. Accessed: 2024-03-05.

[18] Python Software Foundation. The import system. https://docs.python.org/3/reference/import.html. Accessed: 2024-02-08.

[19] Simon Fritsche, Malte Schwerho, and Peter Müller. Verifying scala's vals and lazy vals, 2014.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[21] Robotics Research Lab. Datatypebase.java. https://www.finroc.org/browser/rrlib_serialization-java/rtti/DataTypeBase.java?rev=21. Accessed: 2024-02-21.

[22] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John S. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2005.

[23] Fengyun Liu, Ondřej Lhoták, David Hua, and Enze Xing. Initializing global objects: Time and order. *Proceedings of the ACM on Programming Languages*, 7:1310–1337, 10 2023.

[24] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[25] Oracle. Expressions. https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html. Accessed: 2024-03-06.

[26] Oracle. Initialization of classes and interfaces. https://docs.oracle.com/javase/specs/jls/se8/html/jls-12.html#jls-12.4. Accessed: 2024-03-06.

[27] Oracle. Initializing fields. https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html. Accessed: 2024-02-07.

[28] Oracle. Interface concurrentmap<k,v>. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentMap.html. Accessed: 2024-02-16.

[29] Oracle. Java native interface specification. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. Accessed: 2024-02-21.

[30] Oracle. Loading, linking, and initializing. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html#jvms-5.5. Accessed: 2024-02-03.

[31] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[32] Alok Sanghavi. What is formal verification? *EE Times_Asia*, 2010.

[33] Alexander Shvets. *Dive Into Design Patterns.* Refactoring.Guru, 2018.

[34] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1), may 2012.

[35] Minecraft Developer Team. Minecraft-hack-client-1.8. https://github.com/CrxsCode/Minecraft-Hack-Client-1.8/tree/ea2901180d51806c432d7acd85ecf430f9a6167c. Accessed: 2024-03-04.

[36] Bill Venners. The lifetime of a type. https://www.artima.com/insidejvm/ed2/lifetype.html. Accessed: 2023-09-20.

[37] ZanXusV. java-design-patterns. https://github.com/ZanXusV/java-design-patterns/tree/master. Accessed: 2024-03-04.

[38] ETH Zürich. Viper tutorial. https://viper.ethz.ch/tutorial/#predicates. Accessed: 2024-02-10.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

SPECIFYING AND VERIFYING STATIC INITIALISATION IN DEDUCTIVE PROGRAM VERIFIERS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
FIRLEJCZYK	PATRICIA

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Möhlin, 07.03.2024	PFirlejczyk

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.