SpideyBC

Static Resource Analysis of Safety-Critical Java Applications

Java™

sw101f13

Mikkel Todberg
Jeppe Lund Andersen

Thesis Title:
SpideyBC – Static Resource Analysis of Safety-Critical Java Applications

Area:
Embedded and Distributed Systems

Field of Study:
Software Engineering

Project Period:
1^{st} February 2013 - 31^{st} May 2013

Project Group:
sw101f13

Participants:
Mikkel Todberg
Jeppe Lund Andersen

Supervisor(s):
René Rydhof Hansen
Andreas Dalsgaard

Number of Pages:
91

This report documents the design and development of a software tool for statically analysing memory usage in Safety-Critical Java (SCJ) applications. The project acts as a natural extension of our previous work, *A study of Safety-Critical Java and its Specification Applied*[36], in which we investigated the SCJ specification draft from September 2012 and developed a *level 1* compliant use-case library of the *Cubesat Space Protocol*. From this work, we highlighted the difficulty of being able to properly specify the required storage parameters – an aspect that especially proved difficult for SCJ newcomers and to embedded system development in general.
The developed tool, *SpideyBC*, draws on well established concepts from static program analysis including adaptions of acknowledged WCET techniques such as the *Implicit Path Enumeration Technique* (IPET). With SpideyBC, the developer can analyse one or more Java methods such as the `handleAsyncEvent` methods in order to find the maximum dynamic memory consumption and worst-case JVM stack sizes. The results are presented in a report that in a convenient and visual way shows information regarding worst-case execution paths, call graphs, control flow graphs, stacks etc. Furthermore, by using this tool, developers can analyse all methods that allocates in a private memory area, the mission memory area or the immortal memory area in order to get an indication of the worst possible storage size for the memory region in question – thus making the developer able to specify the respective storage parameters of an SCJ application.

PREFACE

This project has been developed by two Software Engineering master's students from Aalborg University in the spring of 2013. The report documents the master's thesis in the period of 1^{st} of February 2013 until 31^{st} of May 2013 and is the conclusion of the Software Engineering study.

The contents of this report is a combination of theoretical and practical material. As a result, the reader is expected to have a background knowledge that is equivalent to having completed 9^{th} semester of Computer Science or Software Engineering.

We use the following conventions throughout this report, unless otherwise stated:

- "We" refers to the reader and authors. When used in relation to referencing the previous work by the authors, "we" refers to the authors exclusively
- The introduction of new terms or names are represented in cursive
- All references to classes, variables, methods and other implementation specific entities are represented in monospace – for example `java.lang.Object`
- Source code listings may omit parts of the code that are not relevant for a particular illustration. The omitted parts will be replaced with "..."

Source material referenced in this report will be denotated with a number in square brackets that represents an entry in the bibliography. Consulting this entry provides information about the source. For example, [32], is an article written by Tórur Biskopstø Strøm and Martin Schoeberl in 2012 and is about a desktop 3D printer in Safety-Critical Java.

All source code for the software product and evaluation related configurations are available on our GitHub repository[2].

Finally, we would like to thank our supervisors René Rydhof Hansen and Andreas Engelbredt Dalsgaard for supervising our previous project and this master's thesis.

Enjoy reading! Group sw101f13

CONTENTS

CHAPTER

ONE

INTRODUCTION

Embedded systems constitute approximately 99% of the worlds production of microprocessors[8]. When these exhibit real-time properties and are applied in critical environments, it becomes a necessity to verify correctness throughout execution. The software used in aircrafts is an example of a safety-critical system. In the United States, the *Federal Aviation Administration* (FAA) requires aircraft software to be certified against the rigid *Software Considerations in Airborne Systems and Equipments Certification* (DO-178B) standard[13]. As an example, for the stringent certification *level A*, object code that is not directly traceable to source code, must undergo additional verification processes – this is also a requirement for all compiler made optimisations[12, 16].

One challenge in certifying safety-critical applications is that certification often must be performed on the exact same object code that is used in the final product. In avionics, it is a requirement that the code used in validation is also the code that ends up in the aircraft. Thus, one must not intertwine application code and measurement code for worst-case execution time analysis, and remove the measurement code afterwards[21, 35]. In other words, "Test what you fly and fly what you test"[35].

Static program analysis deals with examining items of software without executing the program in question[38]. The object code is the artifact from which the desired information about the program is deduced. This is in direct contrast to dynamic analysis. There are, however, limitations in the use of static analysis as there is no general solution to the halting problem[40]. Thus one must often choose between an over- or under approximation. Nonetheless, static analysis is applied for many types of problems, notably for determining worst-case execution time of real-time systems.

The *Safety-Critical Java* (SCJ) specification, under JSR-302[34], is an attempt to leverage the benefits of Java for developing safety-critical applications amendable for certification against rigid standards. In order to achieve this, SCJ differs from Standard Edition Java in different ways. SCJ uses a scoped memory model instead of a garbage collected heap, which requires the programmer to have knowledge on the memory requirements of the application. It is therefore necessary to be able to reason about resource consumption.

In this master's thesis, we look at static program analysis of bytecode compiled Java applications in terms of analysing resource usage. Note that resource usage is not only restricted to dynamic memory, but can also cover e.g. exceptions and locks. Furthermore, we will show how we can use and adapt well-established techniques for other types of program analysis for this purpose.

1.1 The Problem

In our previous work in relation to Safety-Critical Java and its specification, we identified several areas for future work[36]. Empirical evidence from this study showed how it was necessary to analyse and reason about the resource usage of SCJ applications. In this case, the resource in question was memory. In the following, we clarify why this topic is relevant and point out the currently cumbersome method of manually having to reason about memory usage through source code inspection.

1.1.1 Explicitly Stating Memory Requirements in SCJ

One of the main issues we encountered in our previous work, was the difficulty of specifying storage parameters for periodic and aperiodic handlers (see Section 2.2 for a brief recap of the significant parts of SCJ). As a result of the scoped memory model, the developer is required to explicitly specify backing store sizes for all memory regions. This is illustrated in Listing 1.1 for the private memory region of a periodic event handler. Furthermore, depending on the underlying platform other memory requirements may be necessary. Currently, these value must be found manually through source code inspection. This can be error prone, considering that the specified value must be *safe* to avoid a run-time exception.

Listing 1.1: Specifying memory requirements for a periodic handler in SCJ

```
1  PeriodicEventHandler peh = new PeriodicEventHandler(
2   new PriorityParameters(10),
3   new PeriodicParameters(new RelativeTime(0, 0), new RelativeTime(10, 0)),
4   new StorageParameters(1024, null)) {
5     public void handleAsyncEvent() {
6       /*
7        * This handler is invoked every 10 ms
8        * and has a backing store of 1024 bytes
9        */
10    }
11  };
12 peh.register();
```

In the previous study, working with the scoped memory model was found to be one of the largest deviations when coming from Standard Edition Java. Assistance in the use of this memory model was considered to be of great benefit.

1.1.2 Pushing for Garbage Collection

Analysing the resource usage in terms of memory is, however, not only beneficial for determining backing store sizes of memory regions in SCJ. The problem with the scoped memory model, is that it significantly changes the way a SCJ program will be written compared to a standard Java application. This was also highlighted in a RepRap 3D Printer use case[32]. The memory model introduces issues, e.g. with dangling references that the programmer must ensure will not occur. In the latest version of the DO-178B standard from the fall of 2012, requirements for the use of garbage collection have been specified. Being able to employ automatic memory management in SCJ, with the use of garbage collection, is now a possibility. This also requires knowledge of memory resource usage by handlers.

Investigating garbage collection in SCJ has previously been done[30]. The authors use a *time based* approach where the garbage collector runs in its own thread and is scheduled along with the remaining handlers (*mutators*). Determining the lowest possible period for the collector while guaranteeing the system never runs out of memory, requires knowledge of the maximum amount of bytes allocated in each handler release

along with their frequencies. As with the previous stated issue of specifying backing store sizes, this currently requires the programmer to manually perform this analysis.

1.2 Contribution

From the described problem, the main contribution of this master's thesis is to present how to perform static resource analysis on Java bytecode and an implementation of a tool that captures this idea. The master's thesis will be based on the following thesis question:

> *How can the principles of static program analysis be applied for resource analysis of Java applications?*

In order to work with this problem, we delimit ourselves to the following:

- The primary focus will be memory as the resource, but with the opportunity of extending the analysis to other types of resources
- The analysis must provide a safe upper bound on memory usage
- We focus on support for SCJ applications, but work at the bytecode level. This allows analysis, in theory, to be performed on any source program that compiles to Java bytecode such as a Scala program
- We work on the basis of the SCJ specification draft from 6^{th} of December, 2012[34]. In our previous work, we worked on a previous version. Anything based on our previous work where changes have been made that affects this, will be emphasized in this master's thesis.
- We use the SCJ level 0 and level 1 compliant JVM implementation for the *Java Optimized Processor* (JOP)[27] that was also used in our previous work

In addition, we will focus on solving the following subsidiary tasks:

- Provide an easy to use interface for the analysis tool
- Present the analysis results in a clear and understandable way for the developer to use

As a part of our previous work[36], the network-layer delivery protocol *Cubesat Space Protocol* (CSP) was implemented as a library under SCJ restrictions. This has later been integrated in a full Watchdog use case. As a part of evaluating the implemented analysis tool, this CSP based Watchdog will be presented and used as a program under analysis. Furthermore, the public available implementation for the RepRap 3D Printer in SCJ will also be used to evaluate the tool.

1.2.1 Approach

A notable example of a type of static analysis performed on many real-time applications is worst-case execution time (WCET) analysis. We argue that existing methods for this type of analysis can be transferred to and applied for resource analysis. When performing WCET analysis, the desired result is a cost for the program under analysis – in this case execution time. Memory usage can be considered a cost in the same way. To show the applicability of these methods, we begin with a survey of different methods for finding the WCET of a program. With a clarification of these methods, we use this to determine an approach for answering the stated question. The following is an outline of the remaining chapters that reflects the approach for solving the problem:

Chapter 2 We begin with a brief summary of the prerequisites based on our previous work. This will describe concepts of embedded, real-time and safety-critical systems as well as the core concepts of Safety-Critical Java. Furthermore, we provide more detailed information on the storage parameters in SCJ as these are necessary to understand for this master's thesis

Chapter 3 After this, we will cover theory of static analysis. We examine concepts that are necessary in order to realise the indented analysis

Chapter 4 With our approach being based on looking at methods for determining WCET, this chapter will build upon the static analysis and examine techniques in this area

Chapter 5 Doing memory analysis of Java bytecode requires knowledge on the underlying JVM. This chapter will provide details on the underlying JVM that executes bytecode

Chapter 6 and 7 With the necessary theory in place, this chapter describes requirements, design and implementation of a tool that performs the intended analysis. Furthermore we provide an evaluation of the tool by using it on two SCJ use cases – the CSP based Watchdog and the RepRap

Chapter 7 and 8 We end this master's thesis with a reflection on the work done as well as ideas for future work

CHAPTER

TWO

PREREQUISITES

This chapter provides a brief recapitulation of the essential prerequisites based on our previous work, *A Study of Safety-Critical Java and its Specification Applied*[36]. The parts on embedded and real-time systems were primarily based on the book *Real-Time Systems and Programming Languages*[8]. The part on Safety-Critical Java was based on the specification draft from September 2012, but is compliant with the current version from December 2012[34]. For more details on the subjects, we refer to these materials.

In addition to a brief recap, the aspect of specifying memory requirements in SCJ applications will be described. This is based on the `StorageParameters` class and virtual methods, that must be overridden, that are used to parameterise the applications memory requirements as was also seen in problem outline in Section 1.1. This serves to identify the different concrete parameters of the class along with their purposes. Ultimately, this is what the resulting analysis and tool should assist the developer in specifying.

2.1 Real-Time and Embedded Systems

An embedded system can be thought of as a subsystem residing in a larger system. In this report, we use the following definition of on an *embedded system*:

> "Any information processing activity or system which participates in a larger system and interact directly with the real world."[8]

Examples of embedded systems are anti-braking systems, video game consoles, traffic lights and dishwashers. Those systems that *must* respond to external input before some specified time has elapsed are referred to as, *real-time systems*. We define a real-time system as:

> "Any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period."[8]

The distinction between an embedded system and a real-time system can be seen by considering a traffic light and a breaking system for cars. Assume that the traffic light solely operates on a timer. This makes the traffic light absent of the real-time property. The vehicles breaking system on the other hand, must respond to external user input within a specified time when the driver activates it.

Real-time systems can be categorised as being either *hard*, *soft* or *firm*. For hard real-time systems it is imperative that the system responds before reaching its respective deadlines, while soft real-time systems occasionally are allowed to miss deadlines,

however, doing so can cause operational degeneration. Refer to [36] for more information on these topics.

2.1.1 Scheduling

A scheduling algorithm in a real-time setting must guarantee that all participating tasks abide their temporal requirements. Note that the term, *task*, encompasses an underlying thread together with scheduling information, which could be its period, deadline, priority and release offset. In concurrent programs, the scheduler will create an ordering of task execution such that the functional output remains the same if the program is correct. The definition that we use for scheduling is thus:

> "The activity of restricting the non-determinism found in concurrent systems by ordering the execution of tasks, such that they meet their temporal requirements."[34]

Several scheduling strategies exists including *Value-Based Scheduling* (VBS), *Earliest Deadline First* (EDF), *Cyclic Executive* (CE) and *Fixed-Priority Scheduling* (FPS).

Before putting a newly developed real-time system into use, the system must be verified as being schedulable. Testing for schedulability involves predicting the worst-case behaviour of the system under the particular scheduling algorithm. For each task this entails obtaining its worst-case execution time (WCET). After obtaining the WCET values, which are safe (over-)approximations, a test for schedulability can be performed such as the *Response-Time Analysis* (RTA) in case the scheduling algorithm is FPS. Refer to [36] for more information about scheduling, scheduling algorithms and schedulability tests.

As previously stated, methods for determining WCET will be examined in order to utilise this for the analysis required in this project. Chapter 4 will go into further details on this topic.

2.2 Safety-Critical Java

Safety-Critical Java (SCJ) applications are developed under a strict programming model that, amongst other things, places restrictions on available language constructs. The intuition behind the platform is to encapsulate specific responsibilities into logical structures each having distinct purposes. The basic building blocks are:

- Safelet
- Mission(s)
- Event handler(s)

A SCJ application must define one or more *missions*, each claiming responsibility of one or more encapsulated *event handlers*. The event handlers can either be of *periodic* or *aperiodic* type. Periodic event handlers are periodically executed with defined release cycle times, whereas aperiodic event handlers are triggered by events and are without any lower bound on inter arrival times. As an example, the software of an unmanned rover taking earth samples on Mars can consist of missions for navigation, communication, earth sampling and driving. Furthermore the communication mission could contain two event handlers for periodically checking pending inbound transmissions and aperiodically sending sample results back to base whenever these have been obtained. The *safelet* is the application defining class creating the missions and setting the overall storage size.

2.2.1 Compliance Levels

Different types of applications vary in terms of size and complexity as they must adhere to different functional requirements and be applied in various domains. It is furthermore of great interest to reduce this complexity as it reduces the cost and time of the certification process. SCJ defines three levels of compliance for this purpose, with the levels ranging from 0 to 2 – 0 being the lowest offering the least amount of programming flexibility. The main features of the levels are:

Level 0: The scheduler is a cyclic executive where each mission is broken down into fixed size computation parts and run in sequence. The computation parts within missions (event handlers) must be distributed amongst a number of *frames* (*minor cycles*) whose duration must be set. It is thus up to the developer to create the schedule (*major cycle*). Task priorities are ignored and synchronization mechanisms not needed. It is, however, advised to assign priorities and use synchronised methods in case the application should run at a higher compliance level. `Object.wait`, `Object.notify` and `synchronized` blocks are prohibited

Level 1: The scheduler is a fixed-priority scheduler with priority ceiling emulation. This opens up for concurrency and thus also preemption. In order to ensure mutual exclusion, `synchronized` methods can be used. Both periodic and aperiodic event handlers are allowed at this level

Level 2: Same scheduler and event handler types as in level 1, but with additional support for *no-heap* real-time threads. The largest difference between level 1 and 2 applications, is the ability to create nested missions in level 2. In addition, `Object.wait` and `Object.notify` are allowed

2.2.2 Memory Model

SCJ uses the scoped memory model based on that of RTSJ. As a result, there is no garbage collection and the general notion of a heap is gone. In the scoped memory model objects are allocated in an allocation context (*scope*) and are bulk deallocated when the context exits.

SCJ defines the three named memory areas, *immortal memory*, *mission memory* and *private memory*. The immortal memory area contains static fields and objects created during the *initialization phase*. Objects that are allocated in this area will persist throughout the entire execution and are accessible by all underlying event handlers, which makes it an ideal place to store shared data structures. Mission memory area exists throughout the lifetime a missions. Allocations are done during the *mission initialization phase* and are available for all handlers that execute under the mission. Lastly, when an event handler is released, an anonymous private memory area is allocated, which persists only throughout the handlers job execution. In addition, it is possible to enter any number of private nested scopes.

When any of these scopes exits, every enclosed object gets deallocated. For event handlers this happens when the `handleAsyncEvent` terminates, for missions, when it is terminated and when the application terminates, the immortal memory area is deallocated.

Figure 2.1 illustrates the different memory regions of a level 1 SCJ application with one mission having two periodic event handlers and one aperiodic event handler.

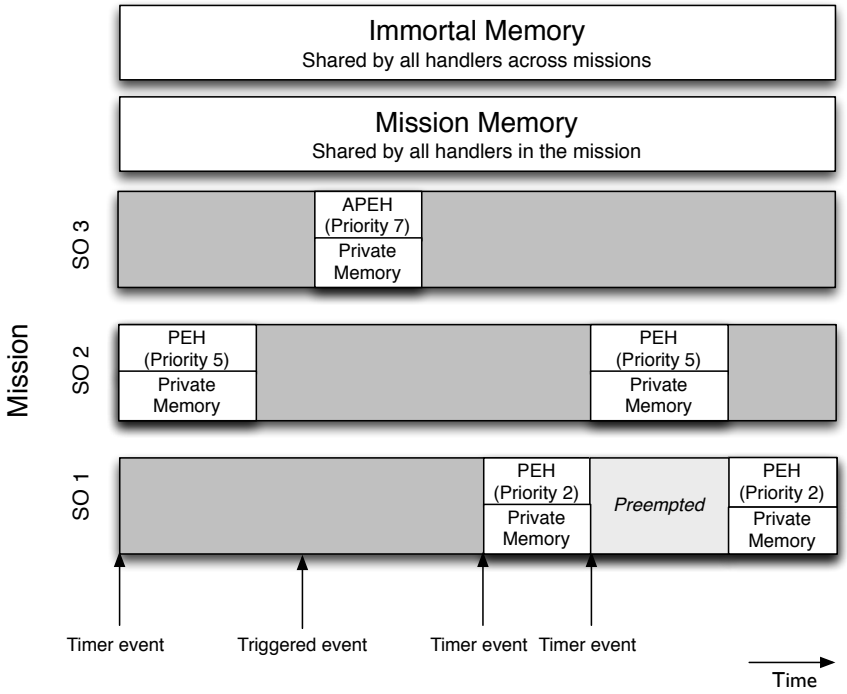


Figure 2.1: The memory areas of a level 1 SCJ application

2.2.3 Storage Parameters

Listing 2.1 shows an extract of the `Safelet` interface and the abstract `Mission` class. An implementation of the `Safelet` interface requires the programmer to implement the `immortalmemorySize` method and thus specify the size of the immortal memory area in bytes. If the available memory is less than the requested size, safelet initialisation aborts. For each created mission, the developer must specify the missions memory size (in bytes) in terms of implementing the abstract `missionMemorySize` method. This number determines the size of the `MissionMemory` area and includes all encompassed backing store memory areas. In other words, the storage used for all contained `ManagedSchedulable` objects within the mission, will use this memory[1].

Listing 2.1: Method signatures in the `Safelet` interface and the abstract `Mission` class for specifying the immortal and mission memory sizes

```
1  @SCJAllowed
2  public interface Safelet<MissionLevel extends Mission> {
3      public long immortalMemorySize();
4      ...
5  }
6
7  @SCJAllowed
8  public abstract class Mission {
9      abstract public long missionMemorySize();
10     ...
11 }
```

Listing 2.2 shows constructors in the `PeriodicEventHandler` and `AperiodicEventHandler` classes. Besides specifying common task information, instances of `StorageParameter` must be supplied.

[1] A `ManagedSchedulable` object could for example be a periodic or an aperiodic event handler.

Listing 2.2: The constructors in `PeriodicEventHandler` and `AperiodicEventHandler` both accepting instances of the `StorageParameter` class

```
1  public abstract class PeriodicEventHandler extends ManagedEventHandler {
2      public PeriodicEventHandler(
3          PriorityParameters priority,
4          AperiodicParameters release,
5          StorageParameters storage) { // <-- difficult part
6              ...
7      }
8      ...
9  }
10
11 public abstract class AperiodicEventHandler extends ManagedEventHandler {
12     public AperiodicEventHandler(
13         PriorityParameters priority,
14         AperiodicParameters release,
15         StorageParameters storage) { // <-- difficult part
16             ...
17     }
18     ...
19 }
```

In order to instantiate an instance of `StorageParameters`, several storage parameters must be known. The constructor of `StorageParameters` is shown in Listing 2.3. In Line 3, the backing store size represents the worst-case scope usage (in bytes) by the associated `ManagedSchedulable` object. The term backing store is used for a range of memory allocated that e.g. a handler is allowed to use (including nested scopes). The `sizes` array in Line 4 must contain configuration parameters for the virtual machine such as the native stack size and Java stack size. It follows that the number and meaning of these values are vendor specific. The `messageLength` in Line 5 covers the memory dedicated to the `ManagedSchedulable ThrowBoundaryError` exception including method identifiers in the stack backtrace. The `stackTraceLength` in Line 6 is the number of elements in the `StackTraceElement` array which is associated with the `ThrowBoundaryError` exception. The `maxMemoryArea` in Line 7 is the maximum amount of memory per release in the private memory area. Finally, `maxImmortal` and `maxMissionMemory` in Line 8 and 9 represents the maximum size the `ManagedScheduble` occupies in immortal memory and mission memory respectively.

Listing 2.3: The constructor in the `StorageParameter` class

```
1  public class StorageParameters {
2      public StorageParameters(
3              long totalBackingStore,
4              long[] sizes,
5              int messageLength,
6              int stackTraceLength
7              long maxMemoryArea,
8              long maxImmortal,
9              long maxMissionMemory) {
10         ...
11     }
12     ...
13 }
```

Note that the SCJ implementation in the JOP repository only uses the first parameters in the constructor, namely `totalBackingStore` and `sizes`. In the remaining parts of the report, we will thus focus on these two parameters as a consequence of using JOP as the target platform.

CHAPTER

THREE

STATIC PROGRAM ANALYSIS

The following is based on unpublished lecture notes by Michael Schwartzbach[31] and the book *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*[21]. Additional sources are referred to as usual. In the previous chapter the essential prerequisites were described. Before continuing with more details on current methods for WCET analysis, we first describe the foundations of static program analysis. The existing WCET analysis methods use this theory and will be used in our analysis as well.

3.1 Background

Static program analysis is an important topic in several areas of computer science. For a given program P, it can be of great interest to know different properties of P. Questions that we may wish to ask can for example be:

- Does P ever throw an exception?
- Does a variable ever change value after its initial assignment?
- Will a null value ever be dereferenced?

For the purpose of this master's thesis, the relevant question to ask about a program would be how much dynamic memory a program allocates in the worst case at run-time? Unfortunately, many such properties cannot be determined by a program analysis without tradeoffs, which is a result deeply manifested in theory of computability. Despite such properties being undecidable, it is possible to make approximations that still provide useful results. Figure 3.1 illustrates the types of approximations an analysis can be, compared to the exact behaviour of the program. Furthermore, for some desired analysis, this can be proved to be sound, meaning that the analysis will only produce valid facts. That is, intuitively, the analysis works as intended. An analysis that is both sound and complete captures the exact behaviour of the program, but is undeciable.

Consider an example of the problem of determining whether a specific type of exception will occur in the execution of some function in a program. An analysis of this may respond *no* if it is certain that such an exception will not occur, or *yes* if such an exception may or may not occur. Even though the results are an over-approximation, being able to state that no exceptions of a certain type would occur in that function, is a strong property to be able to say about a program. Similarly, analysis for determining cases where null value deferencing will happen are often also over-approximations as they are used to establish that null pointer exceptions cannot occur in the program. Given the nature of many problems, an analysis that is an

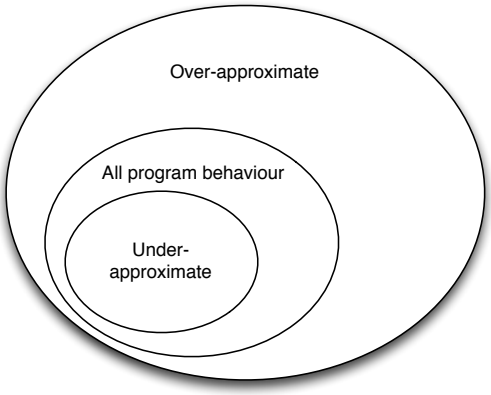


Figure 3.1: Types of approximations in relation to the exact program behaviour

over-approximation will be most appropriate. Moreover, the goal is to be as *precise* as possible and, in the case of an over-approximative analysis, avoid as many false positives as possible.

With these ways of avoiding the undecidable nature of many of the analysis problems, static program analysis is applied in several ways. Notable examples of application are in compilers for optimisation purposes and verification of programs. In the following sections we describe some of the core concepts in static program analysis that are used underneath the types of static program analysis that will be discussed in subsequent chapters, as well as, used in this project.

3.2 Control Flow Graphs

Besides the already mentioned sources in the beginning of this chapter, this section is also based on *Control-Flow Analysis*[5]. The type of analysis problem we work with, relies on the concrete execution flow through the program. Static analysis that requires knowledge of the control flow of the program under analysis, uses a *control flow graph* (CFG). These types of analysis are also called *flow-sensitive analysis*. On the other hand, if the results for a sequence of statements $S_1S_2..S_n$ remains the same for all permutations of the sequence, the analysis is called *flow-insensitive*. Informally, a CFG is a representation of the program and its possible execution paths or the flow. A CFG is constructed using control flow analysis (CFA). This will not be discussed in further details, but we note that algorithms are available for this purpose.

Before introducing a CFG formally, we begin with the fundamental *basic block*. Basic blocks are abstractions over the different program execution points and they constitute the underlying building blocks of a CFG. A basic block is defined as the following:

Definition 3.1. *"A basic block is a maximal sequence of instructions that can be entered only at the first instruction and exited only from the last instruction."[21]*

As an example, a basic block may represent a sequence of instructions with the first instruction being the jump target of a preceding basic block with its last instruction being the jump instruction. It is important to note that only the last instruction in a basic block may branch to other parts in the program.

With basic blocks as the building blocks, a programs flow is represented by a control flow graph, defined as the following:

Definition 3.2. *A control flow graph is a directed graph* $G = (V, E, i)$*, where node*

V corresponds to basic blocks and edges $E \subseteq V \times V$ connect two nodes $v_i, v_j \in V$ iff v_j is executed immediately after v_i. $i \in V$ represents the start node, called source, which has no incoming edges: $\nexists v \in V : (v, i) \in E$.[21]

A special type of a CFG can represent both the control flow within a single function and across function calls. Such a CFG is called an *interprocedural control flow graph* (ICFG). In an ICFG, call nodes (callers) are connected to entry nodes in the target functions (callees). Note that multiple targets may exist in the case of e.g. function pointers or dynamic dispatching. Exit nodes are similarly connected to return nodes for the calling function.

With a CFG representing the control flow of a program, a *path* through a CFG is defined as the following (using an ICFG):

Definition 3.3. *A path π through the control flow graph $G = (V, E, i)$ is a sequence of basic blocks $(v_1, ..., v_n) \in V^*$, with $v_1 = i$ and $\forall j \in 1, ..., n-1 : (v_j, v_{j+1}) \in E$. [21]*

Finally, a program is defined as the following:

Definition 3.4. *All possible paths through the control flow graph $G = (V, E, i)$ starting at i and ending in a sink constitute a program φ. A sink represents a block $s \in V$ with $\nexists v \in V : (s, v) \in E$.[21]*

As can be derived from these definitions, the goal, for our purpose, is to find the path π through a given program, which allocates the most memory resources. It should be noted that we work with bounded applications meaning that the maximum depth of recursive calls and loop iterations are known in advance.

3.2.1 Call Graphs

A different graph data structure often found in use, is a *Call Graph* (CG). Compared to a CFG, a CG describes which functions can be called by each function. A call graph is defined as follows[4]:

- Each node corresponds to a single function in the program
- Each call site (a place where a function can be invoked) corresponds to a node
- If a function f may be called at call site s, then there is an edge going from s to f

Note that different variants exist. Often there is no node for individual call sites, but simply an edge going from the node of a function, f, to each node that f calls (or may call in the case of e.g. function pointers). A CG can be derived from an ICFG, as this includes calling relationships among functions as part of its basic blocks.

3.2.2 Interprocedural and Intraprocedural Analysis

An analysis can work on a single function at a time or on an entire program (whole program analysis). An analysis is said to be *intraprocedural* if it works on a single function at a time[4]. In the case of function calls, worst-case assumptions have to be made in regard to parameters for intraprocedural analysis. An *interprocedural* analysis works on the program flow and works across boundaries of a function. This could e.g. be necessary for an analysis related to the use of data objects pointed to by pointers. An interprocedural analysis will track information from a call site to its target(s). As an example, if a function is called with an argument, the argument is included in the subsequent analysis of the basic blocks within in that function – for intraprocedural analysis, this would not be possible as functions are analysed in isolation with worst-case assumptions being made about a functions formal arguments. An interprocedural analysis can be an approach to increase precision, as worst-case assumptions does not necessarily have to be made at every call site.

3.2.3 Context-Sensitivity

An interprocedural analysis using a ICFG or a CG, can take contexts into account. This is called a *context-sensitive analysis*[4]. As an example, the context of a node within the CG could include call information, as well as any other relevant information for that matter, making it possible to tell from which other function the function is called. Similarly a *context-insensitive analysis* does not take contexts into account. Because the behaviour of a function often depends on the originating call site, precision can be improved by a context-sensitive analysis that include call information. This, however, requires additional analysis information to be maintained.

3.2.4 Example

Let us consider an example of a control flow graph for a program. Consider a simple Java program that calculates the nth Fibonacci number in an iterative approach. Such a program could be written as provided in Listing 3.1.

Listing 3.1: Calculating the nth Fibonacci number

```
1  int FindNthFibonacciNumber(int n) {
2      if (n == 0) return 0;
3      if (n == 1) return 1;
4
5      int current = 0;
6      int previous = 1;
7      int result = 0;
8
9      for (int i = 3; i <= n; i++)
10     {
11         result = current + previous;
12         current = previous;
13         previous = result;
14     }
15
16     return result;
17 }
```

A possible control flow graph for the `FindNthFibonacciNumber` method may be constructed as is illustrated in Figure 3.2 (note that for illustration purposes we keep the level of basic blocks on the actual Java source code in this example rather than on bytecode instructions).

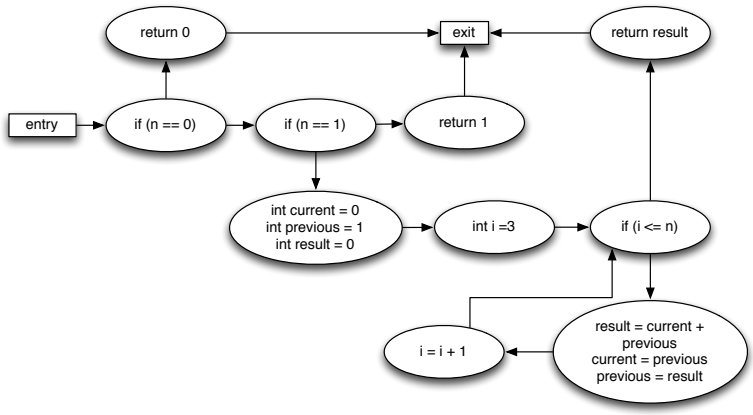


Figure 3.2: A possible CFG for the Java program provided in Listing 3.1

Control flow graphs appear in many places concerning static program analysis, and these will be extensively used throughout the remaining project.

3.3 Pointer Analysis

Pointer analysis is a type of static analysis that determines possible run-time values of pointers[17]. Pointer analysis is relevant for programs produced in languages that has pointer functionality, such as C. For Java and Java bytecode programs, this is also relevant as object references can be considered simple pointers. Consider the sample Java program provided in Listing 3.2. Here the invocation of the `toString` method in Line 26 depends on the referenced object type at run-time. This is where a pointer analysis can provide information about what the reference variable `o` may reference, and thereby ultimately we can say something about which `toString` methods may be invoked.

Listing 3.2: What can variable `o` reference?

```
1  public class A  {
2      @Override
3      public String toString() {
4          return "This is A";
5      }
6  }
7
8  public class B  {
9      @Override
10     public String toString() {
11         return "This is B";
12     }
13 }
14
15 public class ToStringPrinter {
16     public static void main(String[] args) {
17         Object o = null;
18         switch(args[0]) {
19             case "A":
20                 o = new A();
21                 break;
22             case "B":
23                 o = new B();
24                 break;
25         }
26         System.out.println(o.toString());
27     }
28 }
```

Similar to many of the other analysis problems previously presented, a complete analysis of pointer targets is also undecidable[17], thus different approaches that makes approximations have been proposed. These approximations are conservative and sound over-approximations. The result of a pointer analysis is, for each pointer variable p, the set of (possible) targets that p may point to. This information is then typically used for other subsequent analysis. Because others are dependant on the pointer analysis for their own analysis, the precision of the approximation is important. However, other properties are also important such as the speed if working on projects with many SLOC. One may therefore wish to sacrifice some precision for speed for certain types of analysis. We here present the intuition behind the well known *Andersens* algorithm that performs points-to analysis. We refer to [26] that makes a comparison between the different significant types of pointer analysis for additional approaches.

3.3.1 Andersen-Style Points-To Analysis

Andersens algorithm is a flow-insensitive and context-insensitive approach[26]. The algorithm is originally made for analysing pointers in the C programming language[6]. However, it has also been applied in the analysis of Java applications for analysing object dereferencing as well as examined with support for context-sensitivity[19].

Andersens can be described as the creation of a conceptual points-to graph that is maintained and updated as statements are examined. Each node v_i corresponds to pointer variable p_i in the program. A directed edge e from node v_1 to v_2 indicates that pointer variable v_1 may point to v_2 at some point in the execution. This points-to graph is created under a number of constraints specified for the possible pointer manipulations.

We will now show Andersens points-to algorithm and how it is applied for the Java programming language. This is slightly less complex than for C as only simple object references are available. We refer to Appendix A for a similar example in C that illustrates Andersens algorithm in its original form.

Algorithm and Example for the Java Language

We here use set constraints and notation from [31] to describe Andersens algorithm. Let $\langle type \rangle$ and $\langle size \rangle$ denote an application type name and a constant value (or variable having an integer constant value) respectively. Let $\langle type \rangle\_object\_instance - i$ and $\langle type \rangle\_array\_instance - i$ denote an object instance and an array instance instantiated at site i respectively. For each reference variable id in the program, $[\![id]\!]$ denotes the set of possible targets to which id may reference.

The analysis assumes that all reference manipulations in the program are one of the following four kinds (derived from [31]):

1. $id = new\ \langle type \rangle()$
2. $id = new\ \langle type \rangle[\langle size \rangle]$
3. $id_1 = id_2$
4. $id = null$

The following constraints are specified for each of these reference manipulations (except null value assignment), that is used to create the points-to graph:

$$\begin{aligned} id = new\ \langle type \rangle() : &\quad \{\langle type \rangle\_object\_instance{-}i\} \subseteq [\![id]\!] \\ id = new\ \langle type \rangle[\langle size \rangle] : &\quad \{\langle type \rangle\_array\_instance{-}i\} \subseteq [\![id]\!] \\ id_1 = id_2 : &\quad [\![id_2]\!] \subseteq [\![id_1]\!] \end{aligned}$$

The result is a points-to function that for a reference variable p, returns possible targets that p may reference. The points-to function is defined as follows:

$$pt(p) \quad = \quad [\![p]\!]$$

The general intuition behind these constraints, is that in the case that a reference variable v_1 is assigned another reference variable v_2, all possible targets of v_2 are possible targets of v_1. We refer to [31] that provides details on *The Cubic Algorithm* that can be used for creating the points-to graph based on these constraints. The worst-case time complexity of creating and solving these constraints is $O(n^3)$.
Consider the following sequence of statements with reference manipulation from a Java program:

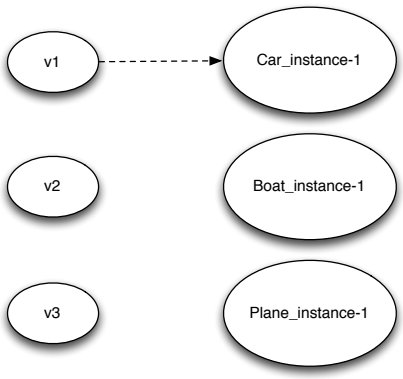
```
Vehicle v1, v2, v3;
v1 = new Car();
v2 = new Boat();
```

```
v3 = new Plane();
v1 = v2;
v3 = null;
v3 = v1;
```

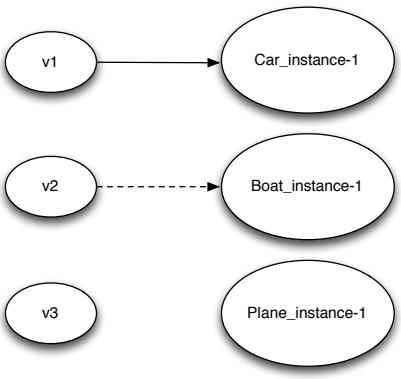
Assume we are interested in knowing possible targets of `v1`, that is, we wish to know the set returned by the function $pt(v1)$. The following constraints are produced for the program:

$Car\_object\_instance{-}1 \subseteq [\![v1]\!]$
$Boat\_object\_instance{-}1 \subseteq [\![v2]\!]$
$Plane\_object\_instance{-}1 \subseteq [\![v3]\!]$
$[\![v2]\!] \subseteq [\![v1]\!]$
$[\![v1]\!] \subseteq [\![v3]\!]$

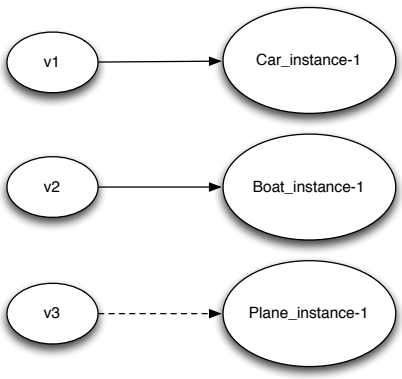
Let us consider how the resulting points-to graph is constructed, under these constraints. Figure 3.3(a) illustrates the inclusion of the newly allocated `Car` object instance as a possible target for the variable $v1$ (dashed edges indicates new relationship added). The same principles applies for Figure 3.3(b) and 3.3(c), such that $v2$ and $v3$ now includes a `Boat` and `Plane` object instance respectively.



(a) From constraint $Car\_object\_instance - 1 \subseteq [\![v1]\!]$



(b) From constraint $Boat\_object\_instance - 1 \subseteq [\![v2]\!]$

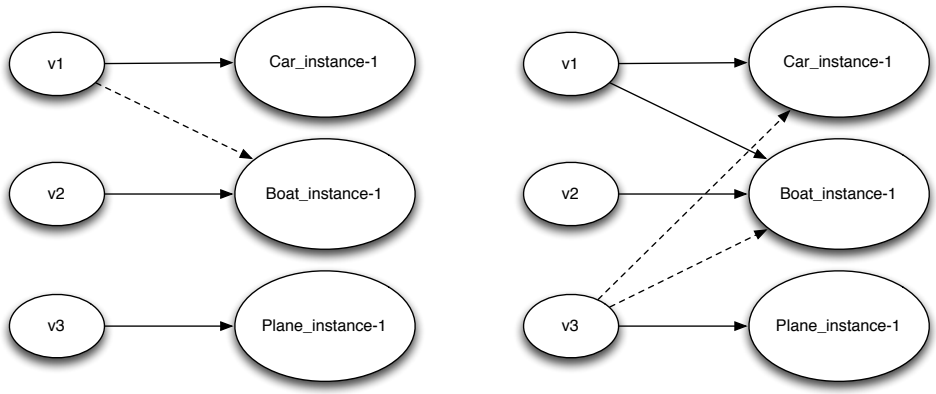


(c) From constraint $Plane\_object\_instance - 1 \subseteq [\![v3]\!]$

Figure 3.3

The fourth constraint, illustrated in Figure 3.4(a), lets $v1$ point to all targets of $v2$

making $v1$ point to a `Boat` object instance as well. The final constraint makes $v3$ include the `Car` and `Boat` object instances. This can be seen in Figure 3.4(b).



(a) From constraint $[\![v2]\!] \subseteq [\![v1]\!]$

(b) From constraint $[\![v1]\!] \subseteq [\![v3]\!]$

Figure 3.4

This result also shows how Andersens algorithm is an over-approximation. From inspecting the source code, we can surely say that $v3$ will reference $Boat\_instance{-}1$ through $v1$s assignment to $v2$. Yet, the analysis will report that $Car\_instance{-}1$ is also a possible target, as a result of being an over-approximation. Being flow-insensitive, the order of statements does not have any impact on the results. The results will be the same for all sequence permutations of the statements.

CHAPTER

FOUR

WORST-CASE EXECUTION TIME ANALYSIS METHODS

This chapter is based on the article *The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools*[39] and *WCET Analysis of Java Bytecode Featuring Common Execution Environments*[14]. Furthermore, the WCET overview is based on our previous work[36]. In the previous chapter we introduced static program analysis. *Control flow graphs* were introduced as a way to represent a program, and *pointer analysis* as a type of static analysis to determine possible targets of pointers or references. This chapter presents an overview of the currently used methods for WCET analysis. These (with the exception of measurement-based approaches) rely directly on the introduced concepts from static program analysis. We begin with a brief outline of the problem at hand and how different approaches attempt to address the problem based on our previous work. We then go into details with the methods that are based on a static analysis approach. The goal is to provide an overview of options for how we can perform our analysis using existing well known techniques.

4.1 WCET Analysis Overview

As it was described in Section 2.1.1, it is necessary to know the worst-case execution time of a real-time task in order to determine schedulability. Intuitively, a guaranteed bound on the execution time could be found by providing the tasks with the worst-case input and record the execution time. However, the worst-case input is in general hard to derive and therefore not feasible. Worst-Case Execution Time (WCET) analysis of real-time programs is the process of determining an upper bound on the execution time on a specific piece of hardware. Two properties are used to evaluate methods. *Safety* describes if the method provides estimates or actual bounds. *Precision* describes how close provided bounds or estimates are to actual values. For hard real-time and safety-critical systems we are interested in methods that provide safe bounds such that safe schedulability is ensured. Precision is important as the amount of programs deemed not schedulable but in reality would be, increases as an analysis becomes more imprecise. Here the goal is to be as *tight* as possible.

The different methods that address the problem are generally classified as one of the following overall approaches:

End-to-end measurement the tasks of the real-time system are executed either on the actual hardware or using a simulator on one or more inputs. Metrics that are measured can be e.g. CPU cycles or time. Figure 4.1 illustrates the problem and how measurement based approaches only provides estimates that are not safe. These measurements are typically called the *minimal* and *maximal*

observed times

Static analysis technique methods classified as static analysis techniques are based on not executing the code as described in Chapter 3. As illustrated in Figure 4.1, these provide safe upper bounds with some unprecision. This safety makes the approach attractive for hard real-time and safety-critical systems, and is also the reason we are considering these methods for our analysis

Hybrid As a result of challenges in both end-to-end measurement and static analysis, hybrid analysis is also found. In a hybrid based approach, different parts of the program are analysed using different approaches. The results are then combined

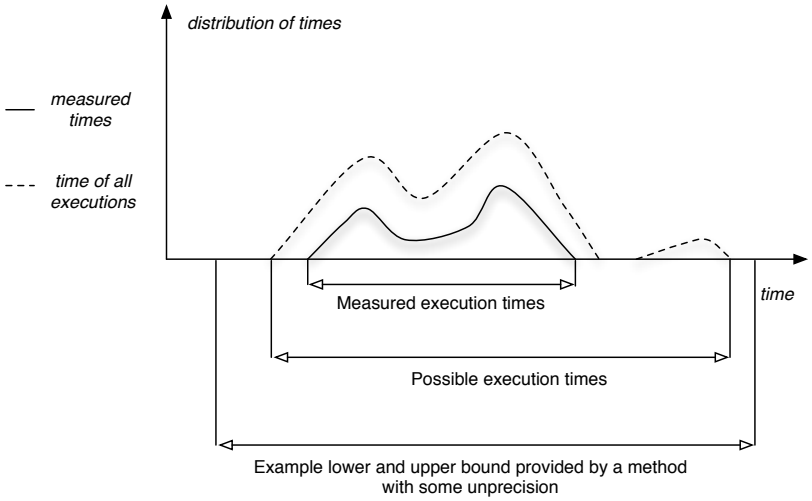


Figure 4.1: The WCET analysis problem and notation - from the previous work[36] that was adapted and modified from the article *The Worst-Case Execution-Time Problem Overview of Methods and Survey of Tools*[39].

With this brief overview, we look at examples of well-known WCET analysis based on the static analysis approach in the following. A method that is particulary interesting is the *Implicit Path Enumeration Technique* (IPET), that has become a popular and much applied method, but we also present a more simple approach. End-to-end measurement and hybrid methods are not considered as these are not viable for our analysis, for reasons previously stated. For additional static methods we refer to [39] that provides an overview.

4.2 Static WCET Analysis Methods

The static analysis methods are generally organised using two types of analysis[18, 10]. *High-level analysis*, or *program path analysis*, determines the execution path (e.g. using a control flow graph) in which the program will exhibit the worst-case execution time and the value of the actual bound. This also includes the control flow analysis to create the control flow graph that represents possible program flows. *Low-level analysis*, or *microarchitecture modelling*, determines the computation time of each atomic unit of flow such as an individual instruction or basic block on a given architecture[11]. The high-level information thus uses the low-level results about execution time of blocks to determine the path in the graph with an upper bound. The low-level analysis is complicated by the complexities of hardware architectures that affects execution time, such as caches. This is very execution time specific and

the focus will be on the high-level analysis of the methods in the following. JOP was created to eliminate (or reduce) these complexities by leaving out many such performance techniques in order to be more predictable and analysable.

A complete WCET analysis is undecidable, and in order to make an approximation, restrictions must be put on the program under analysis. This is done in order to make all parts of a task analysable. The following are some typical program restrictions when using static WCET analysis methods[25, 18]:

- Recursion is not allowed. Recursive algorithms must be replaced by iterative ones
- No function pointers
- Loops must be bounded (which is typically specified explicitly using annotations)
- No dynamic data structures

Looking at these from the perspective of safety-critical systems and SCJ, these are not terrible restrictions on the programming model. E.g. reflection is not available in SCJ, thus no dynamic data structures and thereby class loading is possible. The restriction of function pointers can also be lifted, by the use of a points-to analysis as previously described. With the general requirement of working with bounded and full available programs at compile-time, we continue with examples how to perform the high-level analysis of calculating the execution time bound of a program.

4.2.1 Path-based Calculation

The path-based approach to calculate the bound can be considered the "naive approach". It is based on the use of graph algorithms to explore all possible execution paths of a program in search for the longest path[11]. Paths are therefore visited explicitly. There are different concrete algorithms to perform the calculation, however, in the cited material this works (abstractly) as follows:

Algorithm 1: Path-based calculation

Data:
$CFG \leftarrow$ Control flow graph for a program

```
1 begin
2 |   LongestPath ← null
3 |   repeat
4 |   |   LongestPath ← next longest path in CFG using Dijkstra's algorithm
5 |   until LongestPath is feasible;
6 end
```

Flow facts are used to specify whether a path is feasible or not. Flow facts are considered hints on the program flow[21] (e.g. a fact may express that two specific nodes cannot occur in the path). This can be considered a step to increase precision, as the worst execution path found will be the most expensive that is feasible according to the facts. An approach as this where execution paths are explicitly enumerated to find the worst case path, however, suffers from time complexity increasing exponentially with the depth of conditionals. Listing 4.1 illustrates a Java program with a conditional inside a loop. In this case the loop alone has 2^{1000} unique execution paths. As this is a typical case in Java programs, this approach is not very feasible. Similarly execution frequencies of all nodes must be known in advance, e.g. in case of loops, before explicitly traversing execution paths. It should be noted that variations

and improvements have been made, e.g. by considering only one iteration of a natural loop.

Listing 4.1: Exponential blowup in possible execution paths - adapted and modified example from *Performance Analysis of Embedded Software Using Implicit Path Enumeration*[18]

```
1 private void badMethodToAnalyse() {
2     for(int i = 0; i < 1000; i++) {
3         if (i < 50) {
4             // do something
5         }
6         else {
7             // do something else
8         }
9     }
10 }
```

4.2.2 Implicit Path Enumeration Technique (IPET)

A different approach that has become popular for high-level analysis, is the *Implicit Path Enumeration Technique (IPET)* in which the WCET analysis is made into an *Integer Linear Programming* (ILP) problem[18, 24]. The JOP platform used in this project also uses this approach for its provided WCET tool[29]. As the name implies, it implicitly considers all execution paths when obtaining a solution (the bound on execution time) rather than doing so explicitly like in the previous method. IPET provides the count of visited nodes (which was assumed to be known for the path-based approach), that is, in the worst-case how many times a given node is executed[11].

An integer linear programming problem consists of an objective function, a set of non-negative decision variables restricted to be integers and a set of constraints. The goal is to maximise or minimise the objective function such that the constraints are satisfied.

Let CFG be the control flow graph for a program φ and $BB_i \in CFG$ be a basic block from the control flow graph. The objective function for determining WCET is then defined as the following[29]:

$$WCET = max \sum_{i=1}^{N} c_i e_i \tag{4.1}$$

where N is the total amount of basic blocks, e_i is a positive integer representing the execution frequency of BB_i and c_i is the execution time of BB_i. Intuitively, we can see that by maximising the function, we will get the worst-case execution time. The constraints that should be satisfied, are derived from the control flow graph. These constraints describe the possible program flow, which results in the implicit enumeration of possible execution paths. Two types of constraints are distinguished between, *flow* and *loop* constraints. These are also called *structural* and *functional* constraints respectively.

The flow constraints are derived directly from the CFG and are expressed by defining the execution frequency e_i for BB_i as the following[29]:

$$e_i = \sum_{j \in I_i} f_j = \sum_{k \in O_i} f_k \tag{4.2}$$

where f is a set of decision variables denoting the execution frequency of edges, I_i is the set of incoming edges to BB_i and O_i is the set of outgoing edges from BB_i.
Loop constraints require some knowledge on how loops are modelled in a CFG. Consider the example CFG illustrated in Figure 4.2. A loop header is the entry node

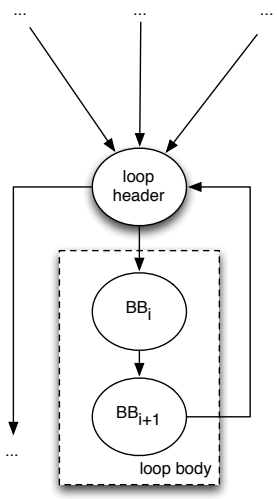


Figure 4.2: A natural loop in a CFG

of a loop. This will always be entered at least once, and is formally the node that dominates every node that is a part of the loop, called the loop body. A natural loop is one in which there is only a single header.

Assume some loop header q. Let F_h be the set of frequencies for each incoming edge to q that enters the loop, and f_l be the frequency of the single outgoing edge from the q to its body. The loop constraint is then defined as (modified for clarification purposes)[29]:

$$f_l \leq n \sum_{f_e \in F_h} fe \tag{4.3}$$

where n is the explicit bound on the loop. Note that the back edge(s) from the loop body to the loop header are not part of the set F_h.

Finally, in order to bootstrap the process, a start node S and a termination node T is added to the graph and assigned special constraints. Let f_s be the single outgoing edge from S to the first basic block and f_t be the incoming edge(s) to T. Then the following constraints are specified:

$$\begin{aligned} f_s &= 1 \\ f_t &= 1 \end{aligned} \tag{4.4}$$

Intuitively, these constraints specify that we start and exit the program a single time. Note that there can easily be more than one incoming edge to the termination node depending on how the CFG is constructed.

To summarise, we have an objective function to maximise, a set of constraints and a set of decision variables that comprise the ILP problem. The final step of finding the optimal solution, can be done using an available LP solver that takes the problem as input and as a result will provide the worst-case execution time as output as well as the values of the decision variables.

Example

We here provide an example to illustrate IPET and how it is used to perform high-level analysis by transforming the problem into an ILP problem. Note that this sample is similar to ones provided in the cited material, but here at the level of Java source code. We assume that loop bounds be explicitly annotated in the source program as

done for JOP and that the execution time cost of basic blocks are provided by some low-level analysis. Consider the example program in Listing 4.2 (the lines are marked with how they could be matched to basic blocks).

Listing 4.2: Sample Java program to analyse

```
1      private void someRealTimeTask() {
2  BB1:    int y = 0;
3          int j = 0;
4
5  BB2:    while(j < 20) { //@WCA loop <= 20
6  BB3:        for(int i = 0; i < 50; i++) { //@WCA loop <= 50
7  BB4:            if (i == 42) {
8  BB5:                y = 42;
9                  }
10                 else {
11 BB6:                someCalculation();
12                 }
13             }
14 BB7:        j = j + 1;
15         }
16     }
```

A possible CFG for this program that is to be used in transforming the analysis into an ILP problem is illustrated in Figure 4.3 with edges labelled with the name of a decision variable. In addition, each basic block is associated with an example of an execution time cost provided by a low-level analysis.

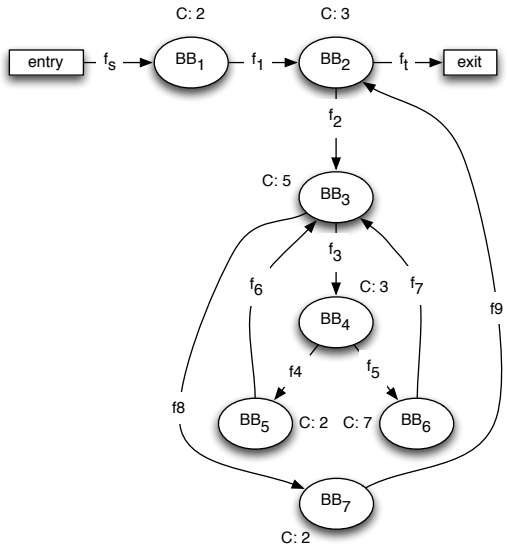


Figure 4.3: A possible CFG for Java program in Listing 4.2 with decision variables on edges and execution time costs for basic blocks

Given the example program and CFG representation, the objective function that should be maximised is defined as:

$$WCET = 2e_1 + 3e_2 + 5e_3 + 3e_4 + 2e_5 + 7e_6 + 2e_7 \tag{4.5}$$

The flow/structural constraints, represented by and extracted from the CFG are as follows:

$$\begin{aligned} f_s &= 1 \\ e_1 &= f_s = f_1 \\ e_2 &= f_1 + f_9 = f_t + f_2 \\ e_3 &= f_2 + f_6 + f_7 = f_3 + f_8 \\ e_4 &= f_3 = f_4 + f_5 \\ e_5 &= f_4 = f_6 \\ e_6 &= f_5 = f_7 \\ e_7 &= f_8 = f_9 \\ f_t &= 1 \end{aligned} \tag{4.6}$$

As can be seen both in the source program, two loops are present. In the CFG, the loop headers are represented by the basic blocks BB_2 for the while-loop and BB_3 for the for-loop. The bounds on these should be represented by the loop, or functional constraints, as these are not directly represented in the CFG, but depends on the functionality of the program. Upper bounds on each loop are provided by the programmer through the use of annotations as seen in Listing 4.2 in the syntax used for applications targeting JOP and using the JOP WCET analysis tool. Using these bounds, the loop constraints are:

$$\begin{aligned} f_2 &\leq 20 f_1 \\ f_3 &\leq 50 f_2 \end{aligned} \tag{4.7}$$

The objective function(4.5) subject to the flow(4.6) and loop constraints(4.7) can then be solved with respect to maximisation of the objective function by using one of many available LP solvers. The result of this will be the maximum value of the objective function (in this case, the computed worst-case execution time) and the values of the decision variables that satisfies the solution. For the provided example, the value of the objective function maximised is:

$$WCET = 15205 \tag{4.8}$$

with the following values for the decision variables that satisfies the constraints (omitting f_s and f_t):

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 20 \\ f_3 &= 1000 \\ f_4 &= 0 \\ f_5 &= 1000 \\ f_6 &= 0 \\ f_7 &= 1000 \\ f_8 &= 20 \\ f_9 &= 20 \end{aligned} \tag{4.9}$$

In the path-based method all execution paths were explicitly considered, but here the result is not a concrete path, but rather a count for each of the nodes of how many times they are executed. This count is the values of the decision variables. The actual

path can, however, be traced. The worst-case execution path can be defined as an extension to Definition 3.3 of a path through a control flow graph provided in Section 3.2. With the execution frequency e_i for each basic block BB_i known, the worst-case execution path is the longest sequence of nodes $(v_1, ..., v_n)$ such that $\forall i \in N : e_i > 0$. In the previous example, we can thus see that the worst-case execution path (with single loop iterations) would be $(BB_1, BB_2, BB_3, BB_4, BB_6, BB_3, BB_7, BB_2)$.

Considerations of IPET

IPET has become a popular approach to the high-level analysis for WCET of real-time systems. A problem is that solving ILP is NP-complete. While the time-complexity of the path-based approach increases exponentially with the depth of conditionals, from a theoretical standpoint, transforming the problem into an ILP problem is not better. Two immediate reasons can be associated with the popularity of the approach. First, several algorithms are available for solving ILP problem that provides optimised techniques in the area. Second, the type of ILP problems for WCET analysis can often have polynomial time solutions and does not experience the exponential blowup[18]. IPET is therefore considered an attractive approach, that can also be used for the analysis required in this project.

CHAPTER

FIVE

THE JVM AND JAVA BYTECODE

The following is based on the JVM specification by Oracle[20]. The *Java Virtual Machine* (JVM) is an abstract computing machine that is generally written in software and running on top of a real hardware platform – hence the word *virtual*. Like an ordinay computing machine, it has its own instruction set and handles different memory areas at run-time, however, it differs as being independent from the underlying hardware and operating system. With these properties, and the fact that it reads a generic *class* file format, the JVM attains portability making it attractive to use in many situations. In this chapter, we briefly describe the JVM and the bytecode it executes. It should be noted that this chapter describes the JVM based on the specification for Standard Edition Java. While notable differences exists between this and a Safety-Critical Java compliant JVM, the basic core concepts appear in both. Likewise, the used SCJ implementation for JOP in this project, is based on an existing JVM that is also compliant with RTSJ. Whenever differences exists, these will be emphasised.

5.1 Overview of the JVM

The steps of compiling and executing JVM compliant applications is illustrated in Figure 5.1. Note that from now on, we will focus on Java applications running on the JVM, however, the same principles are valid for all languages that can be expressed in the class file format such as Scala. In addition, it should be noted that the JVM specification is open as to many of the inner workings, thus actual implementations will vary.

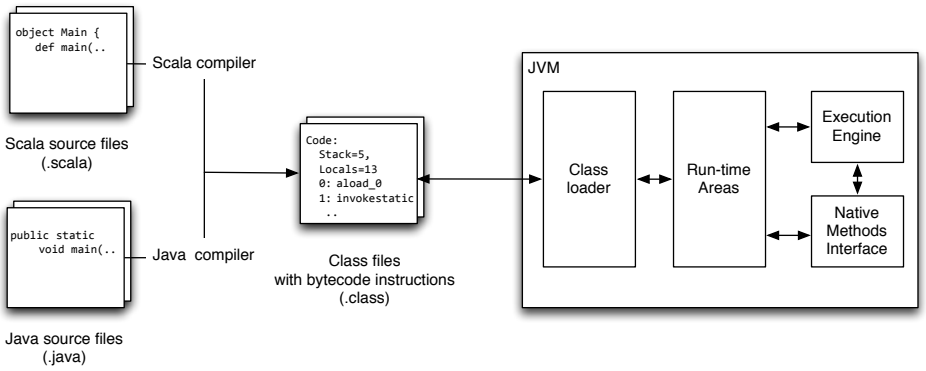


Figure 5.1: The steps of compiling and executing applications on the JVM

Java source files, i.e. those with a *.java* file extension, are first compiled using a java compiler such as Oracles, *javac*. The compiler will output *.class* files for the respective *.java* files, each containing information about a particular class. By default, only line numbers and source file information is included in the class files, however, setting the compile flag, *g*, will include all debugging information[22]. When running an application, the JVM reads and executes the bytecode instructions within the respective class files. Traditionally, when the JVM reads a constructor invocation instruction in bytecode and it is the first access, the JVM dynamically finds the binary class representation matching the type name and then creates a new class. In a linking phase, the class is then combined into the run-time state of the Java Virtual Machine such that it can be executed. Finally, `<cinit>` is invoked, which initialises the class or interface. For the SCJ implementation of the JVM, however, every class is loaded and initialised prior to the instantiation of the `Safelet` class[28]. This requirement favours real-time application development as possible class loading and initialisation does not have to be checked and included in the WCET analysis. Additional differences between a JVM for execution of bytecode e.g. Standard Edition Java and SCJ exists. We refer to the SCJ specification for this information.

5.2 JVM Run-time Memory Areas

During program execution, the JVM manipulates several run-time memory areas. Some of the memory areas are persistent throughout the entire execution whereas others are allocated on-the-fly. This section will outline these areas as well as their respective usage. Some of the concepts in the following sections, will be based on an ongoing code example, provided in Listing 5.1. The program contains a method for calculating the square root of an arbitrary integer value.

Listing 5.1: Sample Java program

```
1  package program.simple;
2  import java.lang.Math;
3
4  class SimpleJavaProgram {
5     public double SquareRoot(int x) {
6        return Math.sqrt(x);
7     }
8     public static void main(String args[]) {
9         SimpleJavaProgram p = new SimpleJavaProgram();
10        double result = p.SquareRoot(16);
11    }
12 }
```

5.2.1 Private Thread Area

Figure 5.2 illustrates the run-time areas with two executing threads. Upon thread creation, a private memory area is allocated, which stores a stack of *frames* (JVM stack) and a *program counter* (pc) for keeping track of the currently executing instruction within a thread. Upon method invocation and termination, a frame is respectively pushed to, and popped from, the stack. A frame contains an array of *local variables*, an *operand stack* and a reference to the *constant pool*. Each single local variable and a single element on the operand stack is a unit of the same size (an integer). The local variables span all variables in the method implementation including parameter arguments[1]. By using the class file disassembler, *javap*, on the class file representing the Java application in Listing 5.1, a formatted and more human readable version of the

[1] For instance methods, a reference to `this` is always present.

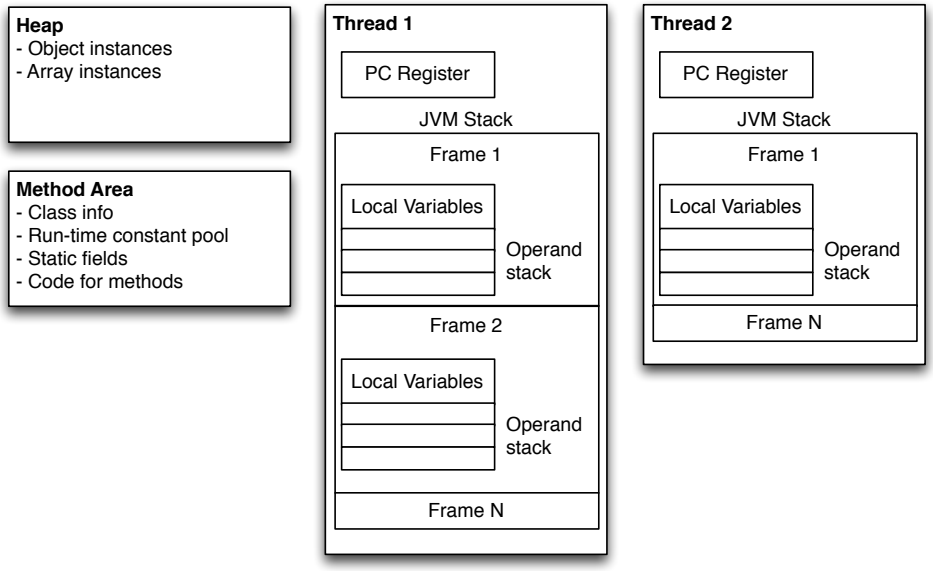


Figure 5.2: The run-time memory areas

bytecode can be inspected. Listing 5.2, shows the `Code` attribute for the `SquareRoot` method, in which the local variable information is kept.

Listing 5.2: Code attribute showing locals and the local variable table

```
1  public double SquareRoot(int);
2      ...
3      Code:
4        stack=2, locals=2, args_size=2
5          ... // bytecode instructions
6        LineNumberTable:
7          line 7: 0
8        LocalVariableTable:
9          Start  Length  Slot  Name   Signature
10           0       6     0    this   Lprogram/simple/SimpleJavaProgram;
11           0       6     1    x      I
```

As seen in Line 8-10 in the Listing, there are two local variables, which are the mandatory `this` for instance methods, and the `integer` (I) variable x. Besides the number of local variables being known at compile-time, the maximum required depth of the operand stack is also available in the class file as seen in Line 4. Here values of `long` and `double` occupies two units in the depth, where each remaining data types uses one unit. The depth of the stack represents the maximum height that is required at some particular time for the JVM to execute the instructions for the method. The `args_size` on the same line is the number of parameter arguments including `this` as the method in question is an instance method. The JVM stack is either of a fixed size or dynamically expandable depending on the JVM implementation.

5.2.2 Heap

The heap is a shared memory region storing object instances and arrays – see Figure 5.2. The memory is allocated upon JVM startup and remains persistent throughout execution. Depending on the JVM implementation, different garbage collection algorithms can be used to reclaim dead objects or arrays. The heap can either be of a fixed size or dynamically expandable depending on the JVM implementation. In

SCJ, the concept of the Heap is gone (or this implicitly becomes the Immortal Memory Area). Instead the collective memory region used for storing object instances and arrays are divided into the individual scoped memory regions as described in Chapter 2.

5.2.3 Method Area

The method area is like the heap, a shared memory region between all participating threads – see Figure 5.2. For each class it, amongst other things, stores class information covering field and method data, a run-time constant pool and code for methods and constructors. The area is created upon JVM startup. Listing 5.3 shows the first part of the constant pool for the decompiled class file for the Java application in Listing 5.1.

Listing 5.3: Constant pool for the Java application in Listing 5.1.

```
1 class program.simple.SimpleJavaProgram
2   SourceFile: "program.java"
3   minor version: 0
4   major version: 51
5   flags: ACC_SUPER
6 Constant pool:
7    #1 = Methodref   #6.#27   //  java/lang/Object."<init>":()V
8    #2 = Methodref   #28.#29  //  java/lang/Math.sqrt:(D)D
9    #3 = Class       #30      //  program/simple/SimpleJavaProgram
10   #4 = Methodref   #3.#27   //  program/simple/SimpleJavaProgram.
11                                  "<init>":()V
12   #5 = Methodref   #3.#31   //  program/simple/SimpleJavaProgram.
13                                  SquareRoot:(I)D
14   #6 = Class       #32      //  java/lang/Object
15   #7 = Utf8        <init>
16   ...
```

5.3 Java Bytecode

The bytecodes constitute the virtual instruction set of the JVM. These instructions are directly executed by the JVM and are the least unit of execution as well as the *object code* of a Java program. Each instruction has the length of a single byte, but may require additional operands, each with a length of a single byte as well. The complete Java bytecode instruction set is particularly interesting. Both instructions, that may be considered simple, found in native hardware (e.g. to manipulate registers), are found in the set, as well as more high level instructions supporting the high level features of the Java language. The instruction `iadd` with two operands, performs a simple addition of two integers and leaves the result as TOS (top of stack) on the operand stack of the executing frame. A similar instruction is typically found in hardware instruction sets. In contrast, the instruction `newarray` allocates a new array of a specified type as the operand, and leaves a reference to the newly created array as the TOS. Examples of other higher level instructions are `new`, for allocating a new object, and `invokevirtual` for method invocation through dynamic dispatching.

The previously introduced class files contains the bytecode instructions produced by the compiler for the individual methods. Listing 5.4 shows an extract of the class file with bytecode instructions for the `main` method of the previous Java sample program from Listing 5.1. Compiling the Java program with a canonical Java compiler produces the same, or a semantically equivalent, sequence of bytecode instructions. In the resulting sequence of bytes, the details of e.g. creating a new object can be seen. A new object instance of the type `SimpleJavaProgram` is created in Line 5 with

the `new` bytecode instruction that also leaves the reference to the newly created object as the TOS. The `#3` indicates the third entry in the constant pool – see Listing 5.3. Following this creation the constructor is invoked in Line 7 using the `invokespecial` instruction. With the new object instance initialised, the remaining instructions in the sequence invokes the method to perform a square root calculation. This approach of having a JVM work at the bytecode level, is the enabling power of letting other programming languages, such as Scala or Clojure, utilise the JVM.

Listing 5.4: Bytecode in class file

```
1 public static void main(java.lang.String[]);
2 ...
3   Code:
4    Stack=2, Locals=4, Args_size=1
5    0:   new #3; //class program/simple/SimpleJavaProgram
6    3:   dup
7    4:   invokespecial   #4; //Method "<init>":()V
8    7:   astore_1
9    8:   aload_1
10   9:   bipush  16
11   11:  invokevirtual   #5; //Method SquareRoot:(I)D
12   14:  dstore_2
13   15:  return
14 ...
15 }
```

CHAPTER

SIX

SPIDEYBC - TOOL FOR STATIC MEMORY ANALYSIS

This chapter will cover the development of *SpideyBC* – a tool that performs static analysis of SCJ applications for worst-case memory usage[1].

We start by presenting the requirements for the tool, which are elicited through a combination of considering the initial problems in Section 1.1 and examining the `StorageParameters` class in SCJ. Next, there will be a choice of framework supporting the static analysis. The following sections will describe the design and implementation of the tool with emphasis on specific challenges and how these are handled.

Note that from this point, we work on the basis of SCJ applications exclusively and at the bytecode level.

6.1 Requirements

We identify three overall requirements for the tool:

- Analysis of dynamic memory
- Analysis of the JVM stack
- Presentation of results

These requirements will be elaborated in this section. Following the requirements, restrictions on the programs under analysis are provided.

6.1.1 Dynamic Memory Allocation of Methods

In Section 2.2.3 we saw how the developer must explicitly state backing store sizes for *Immortal Memory*, *Mission Memory* and for each event handler, its *Private Memory Area* (including additional parameters for any nested scopes). The sizes of these areas are directly dependent on the amount of dynamic memory allocated in the execution of the initialisation methods for the safelet and missions, as well as `handleAsyncEvent` for each handler invocation. In addition, the dynamic memory allocated in constructors and auxiliary methods must also be included when specifying the sizes.

[1] The name of the tool is an abbreviation of *spider* and *Java bytecode* and symbolises a spider webbing the bytecode to create structure.

The following bytecode instructions (omitting operands) result in allocation of dynamic memory:

- *new* - creates a new object
- *newarray* - creates a new array of primitive types
- *anewarray* - creates a new array of references
- *multianewarray* - creates a new multidimensional array

Given this, the tool must be able to take as input, one or more method signatures as entry point(s) and provide a safe upper bound on dynamic memory allocation. As an example, assume the developer wants to know the worst-case dynamic memory allocation for the release of a periodic or aperiodic event handler. A fully qualified method signature to the handler would be provided as an input entry point. As the method may include invocations of other methods, we will obtain a more precise analysis result by working on an interprocedural level – i.e. analysing referenced methods and transferring analysis information between these from callers to callees.

Figure 6.1 illustrates this graphically.

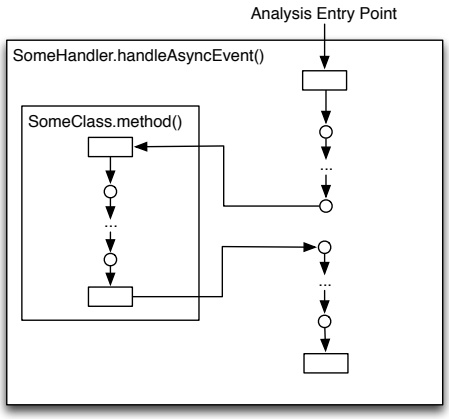


Figure 6.1: Starting analysis from the method, `SomeHandler.handleAsyncEvent`. All dynamic memory allocated in the referenced method, `SomeClass.method`, are included

In relation to specifying the different storage parameters in a SCJ application, the developer can analyse the dynamic memory of several methods after which the results can be combined. The aggregated result will then give an indication of a worst possible memory usage. For example, the value used to specify a missions `MissionMemorySize` can be estimated by analysing the methods of all contained handler classes. Similarly, for the safelets `immortalMemorySize`, the methods of all participating missions must be analysed. The same idea applies to specifying `totalBackingStore` sizes.

6.1.2 JVM Stack Size

In addition to dynamic memory, the developer is also required to parameterise the JVM stacks for handlers. This requires knowledge of the maximum size for the stack of a handler, which is dependent on two parameters – the size of each frame (local variables and the operand stack) and the depth of the stack. Given this, the tool must determine a safe bound of the JVM stack sizes starting from the input method signatures. Figure 6.2 illustrates the concept of specifying a JVM stack size by determining the worst-case size using the sizes of frames (known at compile-time) and knowledge of possible method invocations.

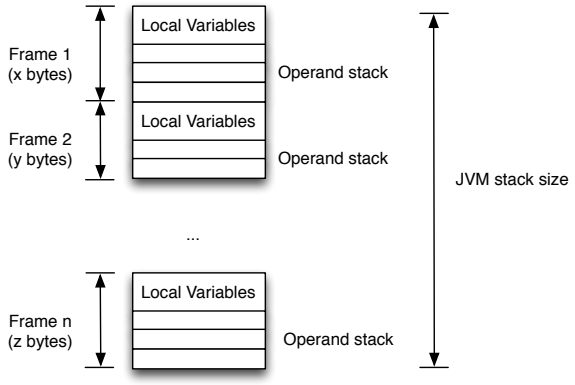


Figure 6.2: The JVM stack size

6.1.3 Presentation of Results

Results of the analysis are only usable to the extent that they are presented. A simple approach would be, in the case of dynamic memory, to output the amount of bytes allocated in the worst-case starting from the invocation of a method. However, we argue that it is necessary to provide additional details such that the developer is able to gain knowledge on what leads to the result. The tool must provide information on the path taken in the program in the worst-case. The challenge is that the programmer works at the source level of e.g. Java, while we work at the bytecode level. Traceability is therefore important. Being able to trace the instructions involved in the worst-case path back to the original source code, will give the end user useful feedback on where allocations happen and on the overall worst-case execution path.

6.1.4 Restrictions for Programs

In Section 4.2, typical restrictions imposed on programs for static analysis were listed. Similarly, in order to meet the requirements for this tool, a few restrictions in relations on programs under analysis are necessary. We require the following in order to work with finite programs:

Explicit loop bound annotations All loops must be explicitly bounded using annotations.

No recursion Recursion is not allowed, and must thus be substituted with iteration.

No reflection The entire program must be available at compile-time – this is trivially satisfied as reflection is not allowed in SCJ.

Furthermore, we require that class files contain the extended debug information – see Section 5.1 for javac. This is required to include local variable information. As source file information and line numbers are included by default, the requirement of traceability is trivially satisfied. Also, having line number information available both enables annotation retrieval and prevents additional verification effort according to DO-178B's requirement stating that additional verification must be performed if the object code is not traceable to the original source code[12].

6.2 Analysis Approach and Supporting Framework

Before describing the design and implementation of the tool, this section presents the approach used for analysis of the program flow. Furthermore we base the tool on an

existing framework that provides many of the foundations required. We provide an overview of such appropriate and available frameworks and state the one used for this tool.

6.2.1 High-Level Analysis Technique

As a part of our approach to this project, which was described in Section 1.2.1, we argued that looking for methods used in static WCET analysis would be useful for memory analysis. In Section 4.2, two methods were described. The IPET method was found to be a well known approach for high-level analysis of program flow for hard real-time systems. By substituting the cost of each node as execution time with that of a dynamic memory allocation cost, this technique is applicable for the tool.

It was stated that the use of IPET, based on solving ILP problems, often can be done in polynomial time. There may also be cases of applications in which this is not possible, thus solutions are found in exponential time. For this tool this is not considered a major issue as (1) SCJ applications will typically be relatively small and (2) the tool is not used on a live system (it is static analysis), thus analysis time will not impact a deployed live system.

We therefore base our high-level analysis for dynamic memory allocation on the use of IPET as this is a "tried and tested" technique.

6.2.2 Framework

In order to analyse the memory usage of an application, the concepts and theories of static program analysis (Chapter 3) will have to be applied. Based on the semantics of the Java bytecode, it must be possible to construct a (I)CFG in which underlying instructions can be traced back to the source code as traceability was a requirement. In addition, we should be able to perform points-to analysis when handling dynamic dispatching to produce more precise and tight analysis results compared to simple analysis of the class hierarchy.

We will use a framework that is capable of supporting CFG (or ICFG) generation as well as supporting points-to analysis. The reason for choosing an existing framework is that these generally include performance optimised algorithms and implementations. These therefore provide increased performance and precision. Several Java bytecode operating frameworks are available[33, 23, 15, 3]:

- **BCEL:** The *Byte Code Engineering Library* is a toolkit for performing static analysis and dynamic creation or transformation of Java class files. BCEL can, for example, be used to substitute the default class loader with a custom class loader, that intercepts requested class files at run-time in order to perform transformations before these are loaded into the JVM and executed. The static analysis part of BCEL mainly revolves around manually inserting analysis code into the class files upon class loading. The WCA tool for JOP is based on the BCEL framework
- **ASM:** ASM contains many of the same features as in BCEL including class file creation and transformation. It is designed with simplicity and performance in mind, which makes it an attractive framework to use in many situations
- **SOOT:** Soot provides both analysis of bytecode and manipulation. It is mainly used to produce optimised bytecode, but was originally made for the purpose of comparing different analyses. It provide several kinds of internal representations and points-to analysis algorithms
- **WALA:** The *T.J. Watson Libraries for Analysis*, or simply *WALA*, is a collection of analysis libraries for analysing Java bytecode and other related lan-

guages such as JavaScript. Some of its most significant features include analysis of class hierarchies, interprocedural data-flow analysis, pointer analysis, (I)CFG construction and call graph construction

Each of the above libraries could in theory be used as the supporting framework for performing bytecode analysis. In this project we will use WALA. First of all, WALA natively supports many useful data structures and concepts out of the box such as pointer analysis and I(CFG) construction. A second reason for choosing WALA, is that it has been used in the creation of a "points-to analysis tool" for finding illegal references in SCJ applications – i.e. detecting references made to objects in scopes with shorter lifetimes, which can lead to dangling references[9]. By using WALA as opposed to the other libraries, an analysis suite for SCJ applications could be a potential idea, in which all tools uses the same underlying framework.

6.3 Design

In this section we describe elements of the important design decisions of SpideyBC. We emphasise that many design decisions are not described here due to the size and magnitude of the tool. Focus is therefore on overall elements and in particular how it is used.

6.3.1 Overall Components

To separate overall concerns, SpideyBC is divided into three components, each having a clear responsibility. These components correspond to a partitioning of the whole analysis into three distinct phases in a linear sequential order. Figure 6.3 illustrates the components and how they correspond to phases of the analysis.

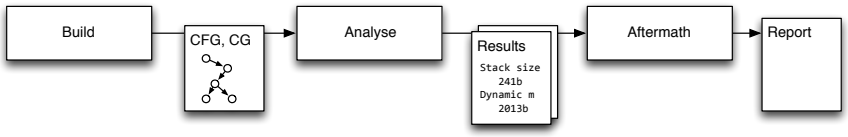


Figure 6.3: Overall components for separation of concerns in SpideyBC

Briefly, each component is responsible for the following:

Build Handles initial setup and initialisation of the analysis based on input (such as the program to be analysed). The result is the data structures (such as the means of getting necessary CFGs and the overall CG) required for the actual analysis

Analyse Takes the data structures constructed and may perform additional manipulations on these if necessary. Performs the actual worst-case dynamic memory analysis using the IPET technique using the data structures and specified entry points, as well as worst-case JVM stack size analysis. The result is data structures containing the results of the whole analysis

Aftermath Takes the raw analysis results and processes these to generate a user friendly result for the developer. This phase and the associated component is directly responsible for the requirements related to presentation. The output is a report that encapsulates and displays all relevant aspects of the analysis in a presentable way

6.3.2 WALA Types and CFG Representation

In this section we briefly cover some of the most interesting data types in WALA. These types are constructed and used in the build and analysis phases and will frequently be referred to during the implementation. The types are described in Table 6.1.

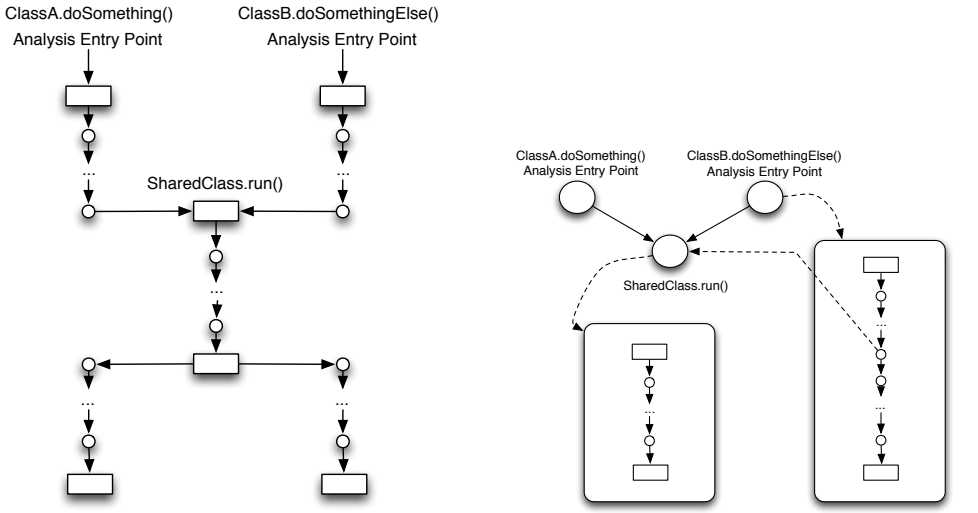
Call graph items	
`IMethod`	Interface representing a Java method. An object of this type will, amongst other things, have methods for obtaining different parts of the method signature and for retrieving the maximum number of local variables and maximum stack height
`CGNode`	Interface representing a Java method (`IMethod`) appearing in a given context. Some of the interesting instance methods include getting its context (`Context`), CFG (`ControlFlowGraph`) and for obtaining information regarding its call sites (if any) along with the associated target(s)
`CallGraph`	Interface representing a call graph consisting of interconnected call graph nodes (`CGNode`). The call graph can be context-sensitive, which means that there can be multiple call graph nodes (`CGNode`) for a single method (`IMethod`)
CFG items	
`IInstruction`	Interface representing a Java bytecode instruction
`IBasicBlock<T>`	Interface representing a basic block of an instruction type `T`. The basic block contains one or more Java bytecode instructions. (`IInstruction`). An implementing object will have instance methods for iterating through its instruction(s) and for figuring out whether or not the block is an entry or exit block
`ControlFlowGraph<I, T extends IBasicBlock<I>>`	Interface representing a control flow graph. Must be parametrised with `I` and `T`, where `I` is a type that implements `IInstruction`. With an object of this type, basic graph functionality is available such as getting the entry node or getting the predecessors/successors of a node. The graph can be traversed by iteration. Each node (`T`) is uniquely identifiable in the graph

`InterproceduralCFG`	Class representing an interprocedural control flow graph where its basic blocks (and their instructions) are on SSA form (`ISSABasicBlock` and `SSAInstruction`). Besides providing basic graph operations, it is possible to retrieve constituting CFGs, however, on SSA form (`ControlFlowGraph<SSAInstruction, ISSABasicBlock`)
Other items	
`SlowSparseNumberedLabeledGraph <T, U>`	Class representing an extended graph having labelled edges. The type can for example be parametrised with a subtype of `IBasicBlock` as `T` and `String` as `U`

Table 6.1: Commonly used WALA types and their descriptions

6.3.3 CFG or ICFG

Transforming the problem of worst-case memory allocation into an ILP problem can be done in two overall approaches. One can use a strategy of solving smaller individual problems using intraprocedual CFGs or to solve one big problem using an interprocedural CFG. The two ideas are illustrated in Figure 6.4.



(a) Solving one large ILP problem per analysis entry point

(b) Solving ILP problems for each referenced call graph node and combining the results in a bottom-up approach

Figure 6.4: The ICFG and CFG approach

Figure 6.4(a) shows the first approach of conducting an analysis on an application with two entry points; `ClassA.doSomething` and `ClassB.doSomethingElse`. Both entry points contains invoke instructions that resolves to the `SharedClass.run` method. Starting from one of these entry points, the basic blocks will be visited and linear constraints generated. Upon encountering an invoke instruction, the basic blocks of the possible target(s), will be included in the ILP problem formulation.

Figure 6.4(b) shows the other approach on the same application. Starting from a call graph entry node, the CFG of the node is retrieved and an ILP problem constructed for the node only. Upon encountering an invoke instruction while traversing the basic blocks of the CFG, the resolved callee node(s) are handled as isolated cases in terms of constructing ILP problems. In relation to the figure, the computed ILP solution of the `SharedClass.run` is used as the cost for the basic block containing the invoke instruction in each of `ClassA.doSomething` and `ClassB.doSomething`.

For this project, we will use the latter approach of solving separate ILP problems and combining the results in a bottom-up approach. What makes this approach superior compared to the other, is that ILP results for call graph nodes can be saved for later usage. Thus, when analysing, say `ClassB.doSomethingElse`, after already having analysed `ClassA.doSomething`, the result for `SharedClass.run` can be fetched instead of being recalculated – This can not be done with the other approach, whereas instead one large problem for the whole ICFG would have to be computed for each entry point.

6.3.4 Low-level Concerns

The low-level part of SpideyBC is concerned with the size of types in memory. For a basic block containing one of the previously listed bytecode instructions that results in dynamic memory allocation, the cost must be the size of the type that an object is instantiated from.

The size of a type is dependant on the underlying platform. Despite Java primitive types have defined sizes that determines the range of values they can hold, a particular platform may have constraints in terms of data alignment. This means that any address reference to memory must be divisible by some integer, and as a result padding is used[4]. Similarly, a type with two integer fields may be cleverly optimised by the compiler such that some bits are shared. As a result, a class with a number of fields of integer types may take up less memory than one with less fields of integer types if only the first type is optimised.

One approach to handle this would be to target SpideyBC specifically for e.g. JOP and use the way objects are stored for this platform and not let the developer be concerned with this. However, this is not very scalable.

Instead, SpideyBC is designed to take a model of the underlying platform as input, together with the application to be analysed. This model specifies the actual sizes of how reference types are stored in memory for the target platform. Listing 6.1 provides an example of the format for such a model specified in a JSON file.

Listing 6.1: Example model for an application to be analysed that specifies size of types in bytes

```
1 {
2     ReferenceSize: 4,
3     PrimordialTypeSizes: [
4         {"java.lang.Object": 1},
5         {"java.lang.Exception": 10}
6     ],
7     ApplicationTypeSizes: [
8         {"com.example.MyObject": 5},
9         {"[I": 8},
10        {"[[I": 12}
11    ]
12 }
```

In the provided example, all sizes are in bytes. Line 2 specifies the size of references and is required. The size for actual types are divided into two arrays for primordial and application types. As an example, Line 8 specifies that the type `MyObject` takes

up five bytes in memory, Line 9 specifies that an integer array of length one takes up eight bytes and Line 10 specifies that a two-dimensional integer array takes up length 12 for the length 1x1. Note that for arrays, the size entered must be for that of a single element.

The reason for this design approach is as follows:

- Providing the low-level details as input to the analysis allows for reuse on other platforms
- Embedded programmers in general can be expected to have good knowledge on their target platform and how data objects are stored in memory. It is therefore not considered a major issue to provide these details
- SCJ applications are not expected to be very large, and not expected to perform many dynamic allocations in the execution phase, but rather set up necessary objects during the initialisation. In reality, the number of types the developer need to specify in the model will therefore not be expected to be many

6.3.5 Input Parameters

The model that parametrises the low-level parts of the analysis and a JAR archive of the application are the two main inputs to SpideyBC. In addition, other input parameters can be specified such as the signatures of the methods to be analysed. SpideyBC takes the following command-line arguments (and possible values):

-jvm_model – Path to JSON file containing the model

-application – Path to JAR archive containing the application under analysis

-jar_includes_std_libraries – *True* if the application JAR includes everything required to run (e.g. compiling SCJ applications to JOP includes all standard libraries available for the platform as a part of the archive). *False* if the application JAR includes only the target application under analysis. In this case, the SE Java library available on the executing machine will be included and used in the analysis, which may severely increase analysis time[2]

-source_files_root_dir – Path to the root directory of the source files for the application provided in the JAR. Files containing types are assumed to be organised according to their package hierarchy. This is used to provide results in context to the original source files as well as for extracting loop and array annotations

-output_dir – Path to a directory where SpideyBC can store analysis results

-main_class – The fully qualified type containing the `main` method

-entry_points – Full method signature name(s) for entry points to the analysis (separated by comma)

Before continuing with the implementation details, the design and overall program flow is summarised in Figure 6.5

6.4 Implementation

This section will describe different parts of the SpideyBC implementation. We focus on parts that present challenges and how these have been solved as well as those elements of the tool that present the most important functionality. The full source code is available on GitHub[2].

[2]In fact, it is recommended to use WALA with SE Java versions below 1.5.

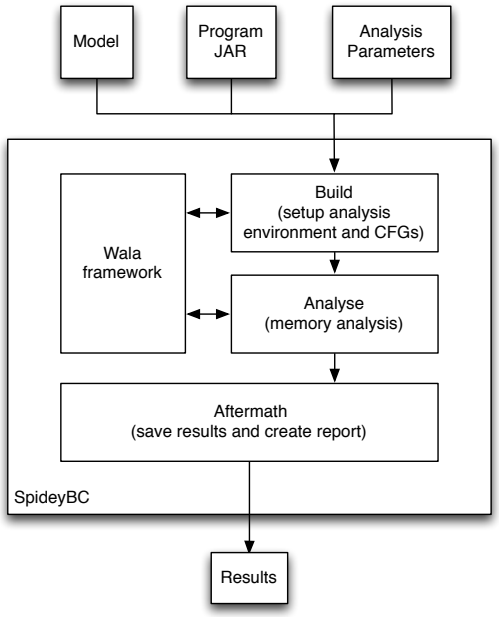


Figure 6.5: Overview of the SpideyBC tool

6.4.1 Construction of CG and CFGs

The main objective of the build component is to construct the necessary data structures for the subsequent analysis. This involves the use of WALA to produce the structure of the class hierarchy, call graph and control flow graphs. The build component is implemented in the `sw10.spideybc.build` package. The implementation performs four overall steps:

1. Create analysis scope that determines which types are to be analysed, and divide these into *primordial* and *application* scope. This includes creating a mapping between type names and source files, which can be done by scanning the user supplied source files root directory for files. All types in application scope are expected to have a corresponding source file
2. Create class hierarchy from the analysis scope using WALA. The class hierarchy can be used to provide details on types in the analysis
3. Perform analysis that constructs the overall CG from the main method entry point. This includes pointer analysis. Each node of type `CGNode` in the constructed call graph provides an interface to get the corresponding CFG for that node.
4. The final step sets up a singleton of the type `AnalysisEnvironment` that the analysis component may use to get all the constructed data structures. This type contains fields for accessing e.g. the class hierarchy and CG

Listing 6.2 provides an extract of the `AnalysisEnvironmentBuilder` type that implements the listed steps. The provided method, `buildZeroXCFAAnalysis`, performs the third step of constructing the CG. WALA provides different algorithms for constructing a CG. In SpideyBC, the CG and underlying CFGs are constructed using a variant of Andersens algorithm for pointer analysis. This is seen in Line 10. This constructs a CG with a context-sensitivity of depth of three. Reasons for enabling context-sensitivity will be elaborated in the next section as this is related to one of the applications used in the evaluation.

Listing 6.2: Creating the call graph with context-sensitivity

```
1 private CallGraph buildZeroXCFAAnalysis(AnalysisScope analysisScope,
2                                         ClassHierarchy classHierarchy) {
3
4     AnalysisOptions options = buildEntryPoint(analysisScope,
          classHierarchy);
5
6     Util.addDefaultSelectors(options, classHierarchy);
7     Util.addDefaultBypassLogic(options, analysisScope, Util.class.
          getClassLoader(), classHierarchy);
8
9     AnalysisCache cache = new AnalysisCache();
10    nCFABuilder builder = new nCFABuilder(3, classHierarchy, options,
          cache, null, null);
11
12    return builder.makeCallGraph(options);
13 }
```

WALA provides a very customisable interface of how a pointer (*PointerKey*) and a heap location (*InstanceKey*) are abstracted. These abstractions are also used in conjunction with the pointer analysis. As an example, targets can be dinstinguished between individual allocation sites compared to only a concrete type. We here use the default value of letting abstractions be based on simply the types in the class hierarchy. Similarly, one can provide special abstractions of a context for the context-sensitivity, by implementing a context selector and types that abstracts different contexts. This allows for different variations of context-sensitivity rather than method invocations. In [9], the authors model SCJ memory scopes as contexts and utilise the pointer analysis for checking invalid assignment. WALA provides default context and selector for method calls. This means that in the analysis of the call graph, one can get a context consisting of the method invocations that leads to the node in question. In SpideyBC, we use this provided context selector, `nCFAContextSelector` (this is implicitly used when the relevant argument for `nCFABuilder` is null). This provides the described context based on call-strings, such that a context for a given `CGNode` instance in the call graph is the call-string of the last n invocations (in this case three). Each context is represented by an instance of `CallStringContext`. If no context-sensitivity is used, the context `Everywhere` is used to represent a single context for each method.

Context-Sensitivity Requirement

The callstring consists of the three latest invocations up to and including a particular node. This was added as the solution to a limitation in the tools ability to handle generic types when analysing the CSP based Watchdog use case during the evaluation in Chapter 7.

Listing 6.3 shows an extract of the CSP queue data structure that was implemented that takes a type argument for the type of the elements that are stored in a queue. The problem here is found in Line 14, where the `dispose` method is invoked on a particular element in the queue. In the CSP implementation, a connection may have a queue of packets, and invoking `dispose` on a connection will invoke `reset` on its packet queue. Furthermore we often have a queue of connections, in which each connection in the queue has a queue of packets.

Listing 6.3: Generic queue data structure used in the CSP implementation

```
1 public class Queue<T extends IDispose> {
2     ...
3     protected final class Element {
4         public T value;
5         public Element next;
6     }
```

```
7
8       public synchronized void reset() {
9           Element element = null;
10          for(byte i = 0; i < count; i++) {
11              ...
12              if (element.value != null) {
13                  element.value.dispose();
14                  element.value = null;
15              }
16              element = (element.next == null ? start : element.next);
17          }
18          ...
19      }
20      ...
21  }
```

The concrete issue is that a connection implements the `IDispose` interface. As a result, the pointer analysis states that the invocation of `dispose` in Line 13 may be on a connection object. Because this then invokes reset on its queue of packets, the static analysis will go into a loop. I reality we know the never-ending loop will never occur, because a connection does not have a queue of connections. We collect calling context about the latest three invocations in the analysis. When evaluating possible targets of a invoke instruction, if either possible target is found in the callstring of the `CGNode` being evaluated, that path is not considered.

6.4.2 Traversal & ILP Constraints Generation

The analysis component, implemented in the `sw10.spideybc.analysis` package, uses the data structures available from the `AnalysisEnvironment` instance. The worst-case dynamic memory allocation analysis takes entry in the `Analyzer` class. This iterates over each analysis entry point and traverses the call graph and the control flow graph starting from each of these. As stated in Section 6.3.3, this works by creating an ILP problem for each `CGNode` in the call graph, and use the results from depending nodes in a bottom-up manner. In the following we provide the general algorithms for the implementation followed by more details on selected parts of this first analysis.

Algorithms

Memory analysis starts for each entry point provided as input to the analysis in the call graph. This is illustrated in Algorithm 2.

Algorithm 2: Starting dynamic memory analysis from each of the entry nodes in the call graph

Input: *entries* ← Entry point nodes in the call graph

```
1 analysisResults ← ∅
2 foreach node n in entries do
3 |   result ← analyzeNode(n)
4 |   analysisResults.add(result)
5 end
```

As can be seen the *analyzeNode* operation provides the actual analysis details of constructing an ILP problem for a specific `CGNode` and solving this. This operation is outlined abstractly in Algorithm 3.

Algorithm 3: The `analyzeNode` operation - traversing a CGNode for generating linear constraints and solving the ILP problem

```
   Input: n ← node in CG
 1 analysisResults ← Analysis results
 2 cfg ← NIL
 3 loopheaders ← NIL
 4 problem ← new empty ILP problem
 5 if analysisResults contains n then
 6 |  return results for n
 7 end
 8 cfg ← makeNumberedLabeledGraph(n.cfg)
 9 loopheaders ← getLoops(cfg)
10 foreach basicblock b in cfg.bfsOrdering() do
11 |  problem.addVariableToObjectiveFunction(b)
12 |  if b.isEntry then
13 |  |  problem.addEntryConstraint(b)
14 |  else if b.isExit then
15 |  |  problem.addExitConstraint(b)
16 |  else
17 |  |  if b.isInvocation then
18 |  |  |  costForBlock ← max(getCostForPossibleTargets(b))
19 |  |  else
20 |  |  |  costForBlock ← getCostForBlock(b)
21 |  |  end
22 |  |  problem.addFlowConstraints(b)
23 |  |  problem.addCostConstraints(b)
24 |  |  if loopheaders contains b then
25 |  |  |  problem.addLoopConstraintFromHeader(b)
26 |  |  end
27 |  end
28 end
29 nodeResult ← solveProblem(problem)
30 analysisResults.add(nodeResult)
31 return nodeResult
```

The essence of the *analyzeNode* operation is that it recursively calls itself on possible target nodes when results are not available (Line 18) through *getCostForPossibleTargets*. As can be seen, all results for individual call graph nodes are stored in a global results pool that allows the reuse of results should a method be invoked from different places. The CFG for the particular node being analysed is extracted from WALA and preprocessed, seen in Line 8, before the traversal. The `makeNumberedLabeledGraph` operation converts the standard `ControlFlowGraph` to a `SlowSparseNumberedLabeledGraph`. We do this to label each edge with a unique identifier that is used as the decision variable in the ILP problem. Line 11 registers the particular basic block as a part of the objective function to be maximised. Line 13 and 15 adds the trivial constraints of setting the execution frequency for the start and end node to one. The remaining constraints added in Lines 22, 23 and 25 adds the constraints as described in Section 4.2.2. We refer to the source code for full implementation details. In the following we briefly describe how the cost of a single basic block that is not an invocation (Line 20) is determined using the input model and the structure of the final results for each `CGNode`. In Section 6.4.3, we elaborate on the *getLoops* operation (Line 9).

Computing Cost of Basic Blocks

The cost for a single basic block is for the dynamic memory analysis extracted from the provided input model. If the basic block contains one of the dynamic memory allocating instructions, the type is examined and the cost is looked up from the model. If the block does not contain any of these instructions (or is an invoke) the cost is zero. The implementation, however, is made such that it would be possible to count other types of resources. The interface `ICostComputer<T extends ICostResult>` contains the method `getCostForInstructionInBlock` that can be used to implement some cost-determining policy. Note that additional methods must be implemented as a part of the interface. In our case, the actual implementation performs a look-up in the model based on the type found in the particular instruction provided as an argument (if the type is not found in the model a warning is provided). An extract of this implementation is seen in Listing 6.4. The actual traversal and generation of constraints are therefore not directly coupled to memory analysis, but simply uses an implementation of the `ICostComputer` interface to determine cost of instructions that comprise a basic block.

Listing 6.4: Providing cost for an instruction by implementing `ICostComputer`

```
1  @Override
2  public CostResultMemory getCostForInstructionInBlock(
3                                              SSAInstruction instruction,
4                                              ISSABasicBlock block,
5                                              CGNode node) {
6
7      /* CostResultMemory implements ICostResult */
8      CostResultMemory cost = new CostResultMemory();
9
10     TypeName typeName = ((SSANewInstruction) instruction).getNewSite().
           getDeclaredType().getName();
11     String typeNameStr = typeName.toString();
12
13     if (typeNameStr.startsWith("[")) {
14         setCostForNewArrayObject(cost, typeName, typeNameStr, block);
15     } else {
16         setCostForNewObject(cost, typeName, typeNameStr, block);
17     }
18
19     return cost;
20 }
```

The interface `ICostResult` must be implemented to store cost results. The most important method on this interface is `getCostScalar` that is used to get a scalar cost for the instruction that can be used in the ILP problem. Similarly when results are aggregated for e.g. a `CGNode`, the cost scalar returns the aggregated cost. For our analysis, the implementation simply keeps an integer field with the allocation size in bytes that are returned directly as seen in Listing 6.5. However, in the case that the resource would be a non-integer value, this would have to be converted into an integer value in the implementation.

Listing 6.5: The cost scalar for our memory analysis is simply the allocated byte size

```
1  @Override
2  public long getCostScalar() {
3      return allocationCost;
4  }
```

Computing Cost of CGNode

After the complete ILP problem is constructed for a `CGNode` the result is computed (the objective function is maximised). We use the open-source LP solver *lpsolve*[1]. The result is stored in an instance that implements the `ICostResult` interface, where `getCostScalar` returns the result for the whole `CGNode` (in this case the amount of bytes allocated in the worst case). Results for each `CGNode` are stored in a singleton of the type `AnalysisResults`, that contains a dictionary mapping a `CGNode` to its `ICostResult` if this has been computed. Furthermore, the analysis results also contains context about the results (such as the trace for basic blocks in the `CGNode` that resulted in the worst case cost).

An example of how the ILP problem is formulated by the tool for an example Java application in the format for lpsolve is provided in Section 6.5 after the implementation.

6.4.3 Handling Loops

In order to generate the loop constraints for a `CGNode`, the CFG is analysed for all loops. In short, for each loop we require to know (1) the loop header, (2) the nodes that constitutes the loop body and (3) the bound on the loop. The functionality for this is implemented in the class `CFGLoopAnalyzer`, which is located in the package `sw10.animus.analysis.loopanalysis`.

Identifying loops is a well-studied problem as programs spend the majority of time executing in loops[4]. Depending on the programming language, several types of loop constructs are often available. From an analysis point of view, how these constructs are mapped to bytecode, is not important. We need to find the loop headers and the natural loop that constitute the body. In [4], an approach is described that can be said to work in three overall steps and is based on two algorithms (step one and three):

1. Create a depth-first spanning tree (DFST) and a depth-first ordering
2. Using the depth-first ordering, identify all nodes that has an incoming edge classified as a retreating/back edge[3] – these are the loop headers. An edge $t \rightarrow s$ is a back edge if s dominates t, that is, s must always be visited before t
3. Construct the natural loop of the back edge which constitutes the loop body

The depth-first ordering is the reverse of a postorder traversal, in that, it visits a node and then visits each child in a right-to-left manner. We thus perform a depth-first search that constructs the DFST and numbers the nodes to give the ordering. In the following we describe the steps in detecting loop headers and basic blocks in the loop body. Finally we describe the format supported in the implementation for specifying loop annotations.

[3]Note that it is not theoretically always the case that back and retreating edges are the same set for a graph. This is, however, the case for almost any program - we refer to the cited material for further information on this topic.

DFST, Depth-First Ordering and Detecting Loop Headers

The algorithm begins by initialising all nodes in the CFG to unvisited and then calls the *search* operation on the entry node, n_0. This is seen in Algorithm 4.

Algorithm 4: Initialisation and starting the *search* operation[4]

```
   Input: cfg ← input CFG
1  T ← ∅
2  foreach node n in cfg do
3  |  mark n "unvisited"
4  end
5  c ← number of nodes in cfg
6  search(n_0)
```

The significant operations happens in the recursive $search(n)$ procedure, provided in Algorithm 5. Note that $dfn[n]$ denotes the depth-first ordering number for node n, T is the DFST for the input CFG and c is a counter for the ordering.

Algorithm 5: $search(n)$ procedure for creating the DFST and depth-first ordering[4]

```
   Input: n ← node in CFG
1  mark n "visited"
2  foreach successor s of n do
3  |  if s is "unvisited" then
4  |  |  add edge n → s to T
5  |  |  search(s)
6  |  end
7  end
8  dfn[n] = c
9  c ← c − 1
```

Figure 6.6 illustrates a simple CFG and how each node would be numbered. The nodes in the CFG will be visited in the order $A - C - D - E - B$. The depth-first ordering will be the reversed visitation order as shown in the figure.

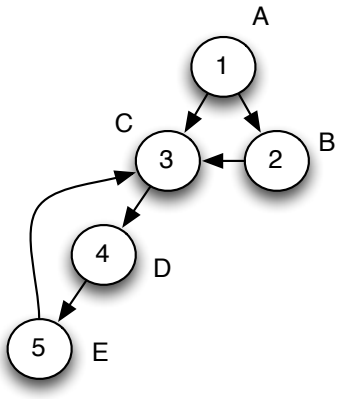


Figure 6.6: Numbering the nodes in depth-first ordering

For the second step, the depth-first ordering on each node can then be used to check for retreating edges that are also back edges. An edge going from node $s \to t$ in the CFG, is a back edge in the case that $dfn[s] \geq dfn[t]$. In such a case, note t is a loop

header. Such an example can be seen in Figure 6.6 for the edge $E \rightarrow C$. In the implementation we perform this check as a part of the depth-first search traversal. More precisely, this is done as a part of the iteration of successor nodes in Lines 2-7 from Algorithm 5. In practice, it should also be noted that it is not necessary to construct the actual DFST for this purpose, only the ordering is needed.

Finding the Loop Body

In the final step, the body of a loop is found. Having found a back edge $s \rightarrow t$, the *natural loop* of the back edge is t and all nodes that can reach s without going through t[4]. Finding the natural loop of a back edge can be done as follows in Algorithm 6. The resulting set, *loop*, contains all nodes that are part of the loop.

Algorithm 6: Constructing the natural loop of a back edge[4]

Input: $cfg \leftarrow$ input CFG, $backEdge \leftarrow$ detected back edge $(s \rightarrow t)$
1 $loop \leftarrow$ {s, t}
2 mark t as "visited"
3 Perform a reverse DFS starting from s until t is visited
4 Add each node visited to the *loop* set

Annotating Loop Bounds in Source Programs

Bounds on the found loops are the final information needed in order to specify the loop constraints. The implementation supports both our own defined annotation as well as annotations supported by the WCA tool in the JOP repository. This enables us to use our tool on existing applications targeting JOP without modifying the bounds. Listing 6.6 shows an example Java method that includes different loop constructs including valid annotations. Annotations are extracted during the build phase in which the source files are scanned when types for the analysis scope are associated with their corresponding source file. The implementation for extracting annotations is located in the package `sw10.animus.util.annotationextractor`.

Listing 6.6: Annotating loops in source files

```
1  private void annotations() {
2      Object obj = null;
3      for(int i = 0; i < 100; i++) { //@ loopbound = 100
4          obj = new Object();
5      }
6
7      int j = 0;
8      while(j < 50) { //@WCA loop <= 50
9          int z = 0;
10         do { //@loopbound = 100
11             obj = new Object();
12             z = z + 1;
13         } while (z < 100);
14         j = j + 1;
15     }
16 }
```

6.4.4 Handling Arrays

Arrays must be handled in a special way as these present several challenges:

1. **Alignment:** Arrays are structures that store elements of a particular type and in a continuous sequence with padding in between to enforce data alignment on

the target platform. This raises the problem of specifying the size while taking alignment into consideration

2. **Multidimensionality:** Arrays can be multidimensional and must thus be treated differently. For example, a one-dimensional and a two-dimensional array of integers are two different types

3. **Array length:** Arrays must be instantiated with a predefined length (set as a constant or variable)

In the following, we will explain how each of these challenges are solved.

The alignment issue is handled by making the assumption that we can multiply the length of the array by the size of the array element type appearing in the supplied model. We believe that this is a fair assumption to make, as the elements of an array are stored in a continuous sequence in memory – the padding between the elements must thus be the same no matter how many elements it contains.

Multidimensional arrays can be treated much like one dimensional arrays. The compiler-made types of a one dimensional array of integers compared to a two dimensional array of integers are `[I` and `[[I` respectively. The developer is therefore required to specify in the model how much an element within each array type occupies in bytes, as was also seen in Listing 6.1 from Section 6.3.4. For calculating the total size of an array, the length(s) must be multiplied with the size in the model. For example in case of a 2x4 2d array of element size 4, its total size is $2 \cdot 4 \cdot 4 = 32$ *bytes*.

As mentioned in Section 6.1.1, there are three bytecode instructions for creating arrays, namely *newarray*, *anewarray* and *mulianewarray*. When any of these instructions are encountered by the `getCostForInstructionInBlock` method (see Listing 6.4) within the memory cost computer, the array cost must be deduced. As seen in the Listing, if the instruction in question contains a left square bracket, it must be one of the available array allocation instructions. As a consequence, the method `setCostForNewArrayObject` is called for determining its cost. Listing 6.7 shows the implementation of this method.

Listing 6.7: Getting the array length from the bytecode or source code before calculating the array cost

```
1  private void setCostForNewArrayObject(CostResultMemory cost,
2                                        TypeName typeName,
3                                        String typeNameStr,
4                                        ISSABasicBlock block) {
5
6   IBytecodeMethod method = (IBytecodeMethod)block.getMethod();
7   int lineNumber = method.getLineNumber(block.getFirstInstructionIndex());
8
9   Integer arrayLength = tryGetArrayLength(block);
10
11  if(arrayLength == null) {
12    Map<Integer, Annotation> annotationsForMethod;
13    annotationsForMethod = extractor.getAnnotations(method);
14
15    if (annotationsForMethod.containsKey(lineNumber)) {
16      Annotation arrayAnno = annotationsForMethod.get(lineNumber);
17      arrayLength = Integer.parseInt(arrayAnno.getAnnotationValue());
18    } else {
19       /*
20        * Warning: allocates array without
21        * specified memory length annotation
22        */
23       ErrorPrinter.printAnnotationError(AnnotationType.AnnotationArray,
              method, lineNumber);
24    }
25  }
```

```
26  try {
27    int allocCost = arrayLength * model.getSizeForQualifiedType(typeName);
28    cost.allocationCost = allocCost;
29    cost.typeNameByNodeId.put(block.getGraphNodeId(), typeName);
30  } catch(NoSuchElementException e) {
31    /* Warning: model does not contain array type */
32    ErrorPrinter.printModelError(ModelType.ModelEntry, method, lineNumber,
          typeName);
33  }
34 }
```

For determining the cost, the array length must be known. The `tryGetArrayLength` method in Line 6 tries to extract the previous bytecode instruction as this will hold the length if the array is defined with a constant length in source code. If the method returns `null`, the array is required to have a source code annotation that specifies its length. This will happen if the size is determined at run-time. Listing 6.8 shows different types of array allocations as well as the required annotation format in case the length is a variable.

Listing 6.8: Array allocations

```
1 int[] arrayOfInts = new int[10];
2
3 float[][] arrayOfFloats = new float[10][5];
4
5 MyObject[] arrayOfMyObjects = new MyObject[20];
6
7 double[] arrayOfDoubles = new double[len]; //@ length = 10
```

After obtaining the length, through bytecode or annotation, the cost can be computed as seen in Listing 6.7 in Line 25-27.

6.4.5 Worst-Case Stack Analysis

In order to find the worst-case JVM stack size from a given call graph node, the most expensive path starting from that node must be found. As stated in Section 6.1.2, the longest path is not necessarily the most expensive as stack frames often vary in size. After having run the dynamic memory analysis, the analysis results (`AnalysisResults`) will contain all call graph nodes (of type `CGNode`) together with their respective dynamic memory analysis results on instances of `CostResultMemory`. The following additional fields on the `CostResultMemory` type are used for the stack analysis:

maxLocals: The maximum number of local variables

maxStackHeight: The maximum operand stack height

stackCost: The sum of `maxLocals` and `maxStackHeight`

accumStackCost: The sum of the most expensive (worst-case) path including the node itself. Will initially be 0 for all nodes

During the traversal of the call graph in the dynamic memory analysis, the `maxLocals` and `maxStackHeight` fields are set for a specific `CGNode` upon finishing the analysis for this. Thus, the necessary information required for determining the worst-case stack size is available after the first analysis. Note that we are able to consider the total cost of a `CGNode` as the sum of the `maxLocals` and `maxStackHeight` field, stored in the `stackCost` field. This can be done as each local variable and a single stack element is the same unit in size as described in the JVM details in Section 5.2.1.

Algorithm 7 shows the `Analyze` operation that initiates the process of calculating the most expensive path and its accumulated stack cost for each entry node.

Algorithm 7: `Analyze` procedure for initiating the JVM stack size analysis for each entry point

```
   Input: entries ← Entry point nodes in the call graph
1  foreach node n in entries do
2  |  dist(n)
3  end
```

Algorithm 8 shows the recursive `dist` operation that accepts a node (`CGNode`) as its only parameter.

Algorithm 8: Pseudo code of the `dist` procedure

```
   Input: n ← node in CG
1  succs ← successor nodes of n
2  max ← −1
3  cost ← −1
4  maxSucc ← NIL
5  if succs not empty then
6  |  repeat
7  |  |  s ← succs.next()
8  |  |  cost ← dist(s) + n.stackCost
9  |  |  if cost > max then
10 |  |  |  maxSucc ← s
11 |  |  |  max ← cost
12 |  |  end
13 |  until succs is empty;
14 |  storeTrace(n, maxSucc)
15 |  n.accumStackCost ← max
16 |  return max
17 else
18 |  n.accumStackCost ← n.stackCost
19 |  return n.stackCost
20 end
```

The algorithm will be explained through the simple call graph shown in Figure 6.7.

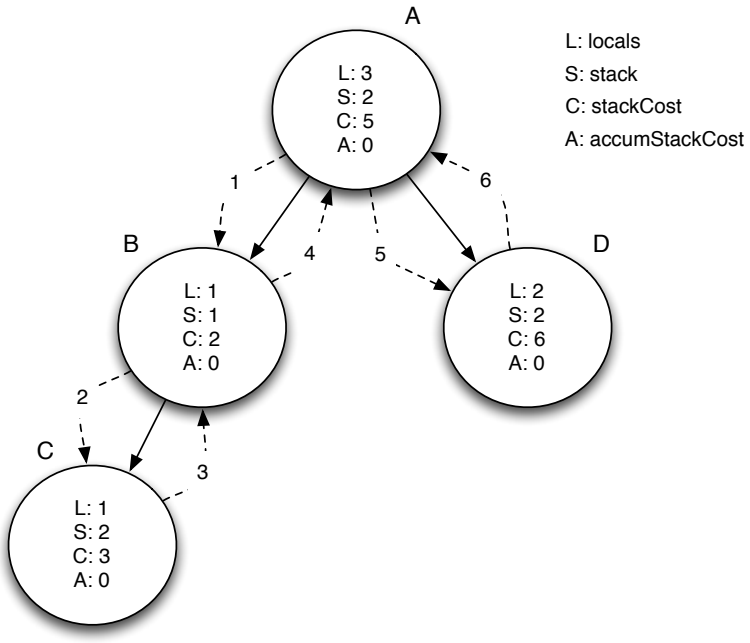


Figure 6.7: Simple call graph. The solid directed arrows represent edges whereas the dashed numbered arrows represent the execution flow of the algorithm. The values on each node represents the initial values before running the algorithm

The numbers in the dashed lines in the figure corresponds with the numbers below and will explain what happens in each step. Assume that there is a single entry point A so that `dist` will be called a single time with A when running Algorithm 7.

1. As node A has successors (B and C), `dist(B)` is called in Line 8
2. As node B has successors (C), `dist(C)` is called in Line 8
3. As node C does not have any successors, C's accumulated stack cost is set to its own stack cost (A: 3) before returning the value. This happens in Line 18-19
4. Execution returns to the call site in Line 8 with dist(C) having evaluated to 3. This makes $dist(C) + n.stackCost$ evaluate to $3 + 2$, which gives 5. In Line 9, *cost* is greater than *max* ($5 > -1$) and the maximum successor is set to C along with *max* updated to 5. As B does not have any other successors, the loop breaks and the path from B to C is saved. Next B's accumulated stack cost is set to *max* (5). Finally *max* is returned. This happens in Line 14-16.
5. Execution returns to the call site in Line 8 with $dist(B)$ having evaluated to 5. This makes $dist(B) + n.stackCost$ evaluate to $5 + 5$, which gives 10. In Line 9, *cost* is greater than *max* ($10 > -1$) and the maximum successor is set to B along with *max* updated to 10. As A still has a successor (D), `dist(D)` is called in Line 8.
6. As node D does not have any successors, D's accumulated stack cost is set to its own stack cost (A: 6) before returning the value. This happens in Line 18-19.
7. Execution returns to the call site in Line 8 with $dist(D)$ having evaluated to 6. This makes $dist(D) + n.stackCost$ evaluate to $6 + 5$, which gives 11. In Line 9, *cost* is greater than *max* ($11 > 10$) and the maximum successor is set to D along with *max* updated to 11. As A does not have any other successors, the

loop breaks and the path from A to D is saved. Next A's accumulated stack cost is set to *max* (11). Finally *max* is returned. This happens in Line 14-16.

In the last step, we see that the path from A to D constitutes the worst-case path as it evaluates to 11 compared to the path from A to B to C, which only evaluates to 10 even though this is a longer path.

6.5 ILP Constraints Generation Example

To round off the implementation, this section presents a brief example of a small Java program, its representation in bytecode and the corresponding linear constraints that SpideyBC generates for the open-source *lpsolve* framework. The example takes its starting point in the small Java program provided in Listing 6.9.

Listing 6.9: Sample Java program

```
1 public static void main(String[] args) {
2     Object obj = null;
3     for(int i = 0; i < 20; i++) { //@ loopbound = 20
4         obj = new Object();
5     }
6     int[] newArray = new int[10]; //@ length = 10
7 }
```

Using the standard javac compiler, the bytecode in Listing 6.10 is provided.

Listing 6.10: Bytecode for the method in Listing 6.9

```
1  public static void main(java.lang.String[]);
2    Code:
3     Stack=2, Locals=3, Args_size=1
4     0:   aconst_null
5     1:   astore_1
6     2:   iconst_0
7     3:   istore_2
8     4:   goto    18
9     7:   new #3; //class java/lang/Object
10    10:  dup
11    11:  invokespecial   #15; //Method java/lang/Object."<init>":()V
12    14:  astore_1
13    15:  iinc    2, 1
14    18:  iload_2
15    19:  bipush  20
16    21:  if_icmplt   7
17    24:  bipush  10
18    26:  newarray int
19    28:  astore_2
20    29:  return
21 }
```

As can be seen, two different types are instantiated at run-time – a plain `Object` and an integer array of length 10. The model in Listing 6.11 specifies examples of low-level details necessary to perform the analysis (note that these values are not targeted a specific platform, but simply for illustration purpose).

Listing 6.11: Input model for the sample program

```
{
    ReferenceSize: 4,
    PrimordialTypeSizes: [
        {"java.lang.Object": 1}
    ],
    ApplicationTypeSizes: [
        {"[I": 5},
    ]
}
```

Providing the application and the model as input to the tool results in the CFG in Figure 6.8 being constructed, for the `CGNode` that corresponds to the main method.

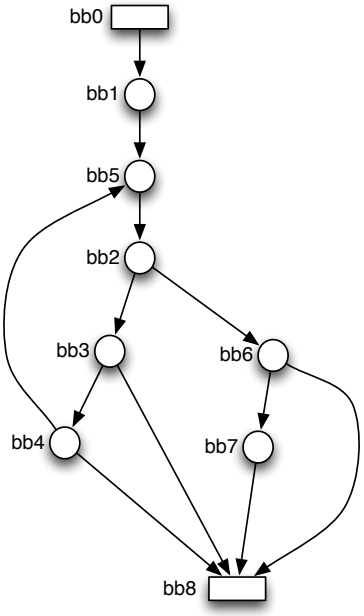


Figure 6.8: CFG constructed for the sequence of bytecode instructions in Listing 6.10

Using the model, annotations in the source files and the constructed CFG, the ILP problem listed in Listing 6.12 is constructed using the open-source *lpsolve* project.

Listing 6.12: Constraints generated for the CFG illustrated in Figure 6.8

```
LP PROBLEM: Maximize
 1*bb0 + 1*bb1 + 1*bb2 + 1*bb3 + 1*bb4 + 1*bb5 + 1*bb6 + 1*bb7 + 1*bb8
Subject To
 1*f0 = 1
 0*f0 + -1*bb0 = 0
 1*f0 + -1*f1 = 0
 0*f0 + -1*bb1 = 0
 1*f5 + -1*f2 + -1*ft0 = 0
 1*f5 + -1*bb2 = 0
 1*f2 + -1*f3 + -1*ft1 = 0
 0*f2 + -1*bb3 = 0
 1*f3 + -1*f4 = 0
 0*f3 + -1*bb4 = 0
 -1*f5 + 20*f1 = 0
 1*f1 + 1*f4 + -1*f5 + -1*f6 = 0
 0*f1 + 0*f4 + -1*bb5 = 0
 1*f6 + -1*f7 + -1*ft2 = 0
 50*f6 + -1*bb6 = 0
```

```
19  1*f7 + -1*ft3 = 0
20  0*f7 + -1*bb7 = 0
21  -1*bb8 + 0*ft0 + 0*ft1 + 0*ft2 + 0*ft3 = 0
22  1*ft0 + 1*ft1 + 1*ft2 + 1*ft3 = 1
```

The results for the constructed problem are seen in Listing 6.13. Here it can be seen that the resulting upper bound on memory allocation is 70 bytes. Results for basic block variables describe the total cost for the block. Results for the edge variables (f prefixed) describes the execution frequency in the solution. Thus in the provided results, we can see that basic block six will in the worst-case allocate 50 bytes, and the program will go through the edge labelled $f3$ 20 times.

Listing 6.13: Results for the ILP problem in Listing 6.12

```
1 Objective: 70
2   {f6=1,  f7=0,
3    bb8=0, bb6=50,
4    bb7=0, bb4=0,
5    bb5=0, bb2=20,
6    bb3=0, bb0=0,
7    bb1=0, ft1=0,
8    ft0=0, ft3=0,
9    ft2=1, f1=1,
10   f0=1,  f3=20,
11   f2=20, f5=20,
12   f4=20}
```

6.6 The Final Tool

Having described the overall design and selected implementation details in the previous sections, this section briefly presents the final result and how SpideyBC looks like in action for programmers.

6.6.1 Front End

A front end provides the entry point to the application compared to running the tool through the command line. This is seen in Figure 6.9. Here the programmer can select and enter relevant inputs, run the analysis and see status of the analysis.

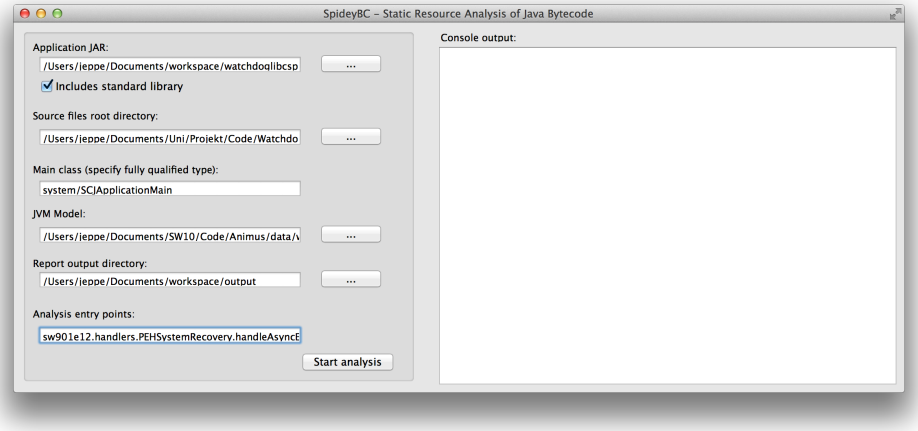


Figure 6.9: SpideyBC front end for running analysis on an application

6.6.2 Analysis Report

The final report generated by the tool is web-based and is created in a local folder provided as an input parameter. This provides the programmer with both the memory sizes of the dynamic memory and JVM stack sizes on each entry point as well as the additional details and traces. The significant elements of reports will be highlighted in the following sections. Refer to Appendix C for viewing additional screenshots of the report than those presented here.

Worst-Case Dynamic Memory Allocation

Results for the worst-case dynamic memory allocation for the provided entry point method signatures are available using the *Allocations* menu. Figure 6.10 shows how the overall memory cost in bytes are displayed for each of the entry points the programmer provided.

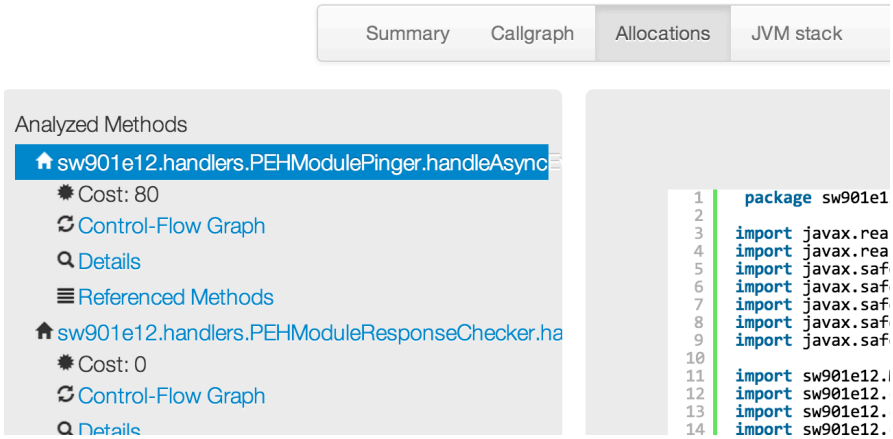


Figure 6.10: Worst-case dynamic memory allocation results per entry point method signature

Furthermore, the programmer can see a highlight of the trace for the path taken in the selected entry point method, as seen in Figure 6.11.

```
36
37      @Override
38      @SCJAllowed(Level.SUPPORT)
39      public void handleAsyncEvent() {
40          if (Config.DEBUG) {
41              console.println("PEHModulePinger");
42          }
43
44          for (Module slave : slaves) { //@WCA loop = 10
45              connection = manager.createConnection(slave.getCSPAddress(), slave.getCSPPort(), CSPManager.TIMEOUT_SING
46              if(connection != null) {
47                  packet = manager.createPacket();
48                  packet.setContent(42);
49                  connection.send(packet);
50                  Packet response = connection.read(10);
51                  slave.setResponse(response != null ? true : false);
52                  connection.close();
53              }
54          }
55      }
56  }
```

Figure 6.11: Trace for the execution path in the selected entry point method that leads to the worst-case memory cost

Finally, aside from seeing the control flow graph for the entry method, the programmer can choose to see details for the selected entry point method. This provides additional information on which types, and how many instances, contribute to the estimated worst-case cost. This is seen in Figure 6.12.

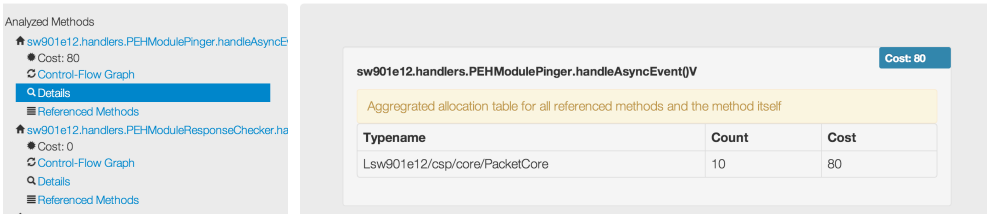


Figure 6.12: Details on which types and allocation count that result in the worst-case cost

Call Graph Exploration

A subset of the call graph is visually presented in a way such that it can be explored by the developer in order to follow the worst-case path for the dynamic memory analysis. Expanding the fake root node reveals all analysed entry methods, which can be further expanded to reveal their referenced methods as long as these participate in the worst-case allocation path. For each node belonging to the application scope (coloured green), the developer can view the source code with background highlighting of the affected lines. Nodes belonging to the primordial scope (coloured red) cannot be expanded nor can these be source code inspected. Nodes marked with a "C" are either directly or indirectly responsible for dynamic memory allocation – directly if they themselves allocate memory and indirectly if any of their children do. The call graph can be seen in Figure 6.13.

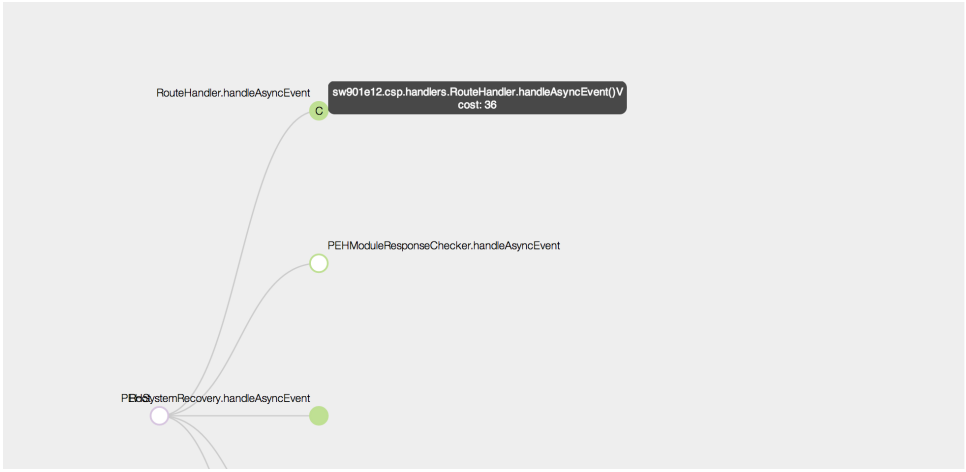


Figure 6.13: Exploring the call graph. Clicking on a node will reveal all possible callees. Double clicking on an application node will reveal its source file with possible highlights. Hovering on a node marked with a "C" will show its allocation cost

Worst-Case Stack Size

The *JVM stack* menu shows the worst-case JVM stacks for each provided entry method. Besides getting the stack sizes in bytes, the developer can visually view the participating frames each containing information about the method signature, the maximum number of local variables, the maximum stack height and an accumulated size for the frame itself and all frames below. This is seen in Figure 6.14.

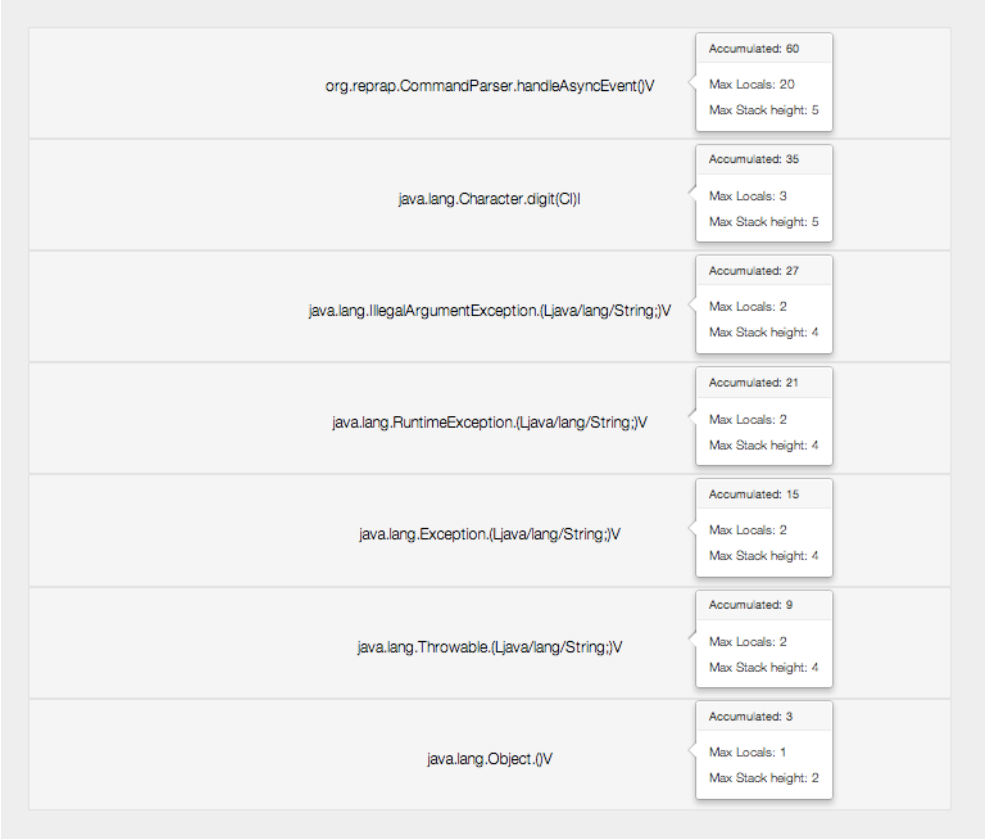


Figure 6.14: The worst-case stack size is shown in details by the frames that would be on the JVM stack as well as their respective sizes

CHAPTER

SEVEN

EVALUATION

In the previous chapter, the worst-case memory consumption tool SpideyBC was introduced. This chapter will present an evaluation of the two primary features, dynamic memory and JVM stack analysis. The evaluation is done by using the tool on two SCJ use cases. The goal is to evaluate the precision of bounds provided by the tool in these cases. In order to compare the precision, the traces in the output reports are used to manually inspect the parts of the programs under analysis where allocations occur. We thus compare the resulting bounds from specific parts of the programs with behaviour that would be expected at run-time.

7.1 Approach

The tool is evaluated on two SCJ applications. The first is an SCJ application for a 3D RepRap printer[32]. The use case was presented at the JTRES workshop in 2012 and the source code is available as a part of the JOP GitHub repository. The second use case is a watchdog application developed by us based on the implementation of CSP for SCJ in our previous work[37]. The Watchdog serves to monitor the availability of different modules in a distributed setting. Appendix B provides a description of the implemented Watchdog for reference. The source code for the Watchdog use case is available at our GitHub repository[2].

We identify the following steps for the evaluation on the two applications:

1. Identify the entry points that should be analysed – we here focus on the handlers of the applications as analysing other parts of the application often involves infrastructural code that increases the complexity – a topic that will be discussed in the reflection in Chapter 8.1
2. Specify the input model for the required types
3. Run the tool on the input parameters
4. Examine the results by inspecting the traces to compare the results with what would happen at run-time

Our approach is based on analysing these applications as-is. The only modifications will be to fill in required annotations. Step 2 will be based specifically on the JOP architecture, the platform used in this project as well as the platform originally targeted in the two applications. Step 4 describes how we will do the actual evaluation of the provided results. We consider an inspection of the specific locations in the source code where allocations occur the best way to compare against the values provided by the tool. As no tool is available for performing the analysis on SCJ applications,

this currently requires manual inspection and estimation of how many instances are allocated (this was the starting problem that the master's thesis is based on!)[1]. We do not consider this an issue, however. Because the applications were originally implemented for the scoped memory model not many instances are allocated in handlers, thus the complexity of allocation patterns are expected to be (relatively) lower than in a traditional Java program.

7.2 Setup

In the following we describe the second step of setting up input models. We note that the run configurations that were created to run the analysis within the Eclipse environment as well as the input model for each application are available on our GitHub repository[2]. These provide everything that encapsulates what is described in this section and what produces the results to be presented in the following section. For easily reproducing the results, an evaluation folder has been created in the repository, which contains the models, application jars and configuration files for the visual front end of the tool. Note that some of the properties within the configuration files will need to be changed as these point to local hard drive locations.

For the input model in each analysis, the input model must be specified with correct type sizes for the target platform. This requires some knowledge of the underlying hardware. We briefly describe details of JOP that are necessary for specifying the correct type sizes.

7.2.1 Memory Access and Layout on JOP

JOP is a 32-bit architecture, with a word size of 32 bits[28]. Furthermore, each field of a primitive type is 32-bit aligned (except for long and double, that are 64-bit aligned). This means that a single field in a class of e.g. the byte type will take up 32 bits, or 4 bytes, when allocated. References are similarly 32 bits in size and aligned.

JOP uses a special memory layout in which all object references goes through an indirection called a *handle*. This is used in relation to the garbage collector for non-SCJ applications. On startup, a fixed region of the memory is reserved for handles, and each handle always takes up the same size. Figure 7.1 illustrates the use of handles. Aside from a reference to the object instance, the handle contains e.g. garbage collector related information and reference to the run-time structure with class fields, method table and so on. In the case the object is an array, the handle contains the length.

[1] The SCJ specification mentions a `SizeEstimator` class which can give a conservative upper bound on the amount of memory required to store objects. However, as this class is not implemented for the target platform (JOP), this can not be utilised.

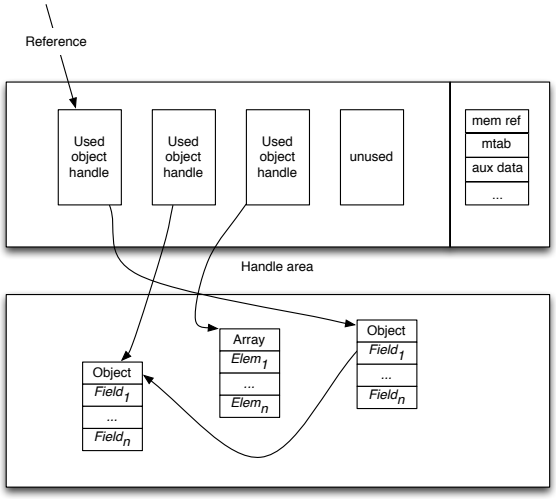


Figure 7.1: Handle indirection used in JOP

This also means that because all classes are initialised at start in SCJ, all class related data is stored in immortal memory. When a new object is instantiated, a handle is reserved and only the instance fields are allocated in memory. As an example, consider a class with three static fields and two instance fields. The static fields will reside in immortal memory, and when objects are instantiated of the type, only memory for the two instance fields will be allocated.

An example is the `PacketCore` type used for CSP packets that are used in the Watchdog application. This class contains ten static fields and two integer instance fields. Each of these instance fields takes up 4 bytes in memory, and as a result, allocating a new object of this instance will have a cost of 8 bytes.

7.3 Results

Table 7.1 shows the analysis results for the event handlers in the two use case applications. The first column represents the respective class names in which the `handleAsyncEvent` methods are defined.

CSP based Watchdog		
Handler class	*Dynamic memory (bytes)*	*Stack size (bytes)*
`PEHModulePinger`	80	53
`PEHModuleResponseChecker`	0	14
`PEHSystemRecovery`	0	11
`RouteHandler`	0	32
`ISRHandler`	0	27
RepRap		
Handler class	*Dynamic memory (bytes)*	*Stack size (bytes)*
`CommandParser`	288	60
`CommandController`	264	45
`HostController`	0	23
`RepRapController`	0	15

Table 7.1: Worst-case memory allocations and maximum JVM stack sizes for the event handlers. The values are extracted from the reports after having run the analysis

Before running SpideyBC on the Watchdog, source code annotations used for debugging purposes are removed in order to ensure that all missing loop bounds and array lengths are reported such that they can be specified and set in accordance to the application under analysis. This is not necessary for the RepRap as the source already has valid loop annotations (supplied by the JOP development team for the WCA tool). The input models for both the Watchdog and the RepRap are left blank. By doing this, SpideyBC will report, in the initial run, the absence of every type that (may) be allocated in the model. The source code of these types can then be inspected in order to calculate the respective sizes using the approach described in Section 7.2.1. Having obtained this information, the types and their sizes can be entered into the model before re-running the tool.

7.3.1 Watchdog Results

For the Watchdog application, only the handler in class `PEHModulePinger`, is responsible for dynamic memory allocation. By inspecting the source code and bounding the number of observable modules to ten through a loop annotation, ten instances of type `PacketCore` will be allocated – refer to Appendix B to view the loop in the `handleAsyncEvent` method. The `PacketCore` instances will be allocated in the referenced `read` method of the `Connection` instance. An instance of `PacketCore` occupies 8 bytes as it contains two integer fields. This yields a total of $10 \cdot 8 = 80$ bytes and matches the result produced by SpideyBC.

7.3.2 RepRap Results

Compared to the Watchdog, the RepRap is a more comprehensive use case and is thus also more interesting from an analytical perspective. Upon running the tool on the RepRap, three types were reported missing from the model:

Primordial type: `java.lang.IllegalArgumentException`

Application type: `org.reprap.Parameter`

Application type: `[C`

In addition, an array length annotation was missing. By inspecting the array declaration and the surrounding code, a worst-case annotation for the length could be inserted.

From Table 7.1, it can be seen that dynamic allocations occur from the handler methods of the `CommandParser` (288 bytes) and the `CommandController` (264 bytes). What is common for both of these methods is that they eventually invoke the array allocating method, whose length was just specified with an annotation. Besides allocating this array, this method also allocates another array of the same size and type[2](`char`). With this information, the two arrays occupy 264 bytes. According to the SpideyBC reports, the `CommandParser` allocates another 24 bytes, which originates from the previously missing type, `org.reprap.Parameter`.

Again, the SpideyBC output values matches what you would expect in terms of dynamic memory allocation. In regards to the JVM stack sizes, by disassembling the java class files and viewing the respective frame sizes, the computed worst-case stack sizes also seems reasonable.

[2]As this other array is declared with a constant length, SpideyBC is able to figure out its length automatically.

7.3.3 Using the Results in the Watchdog

After having obtained the worst-case memory sizes for the Watchdog, the parameters for the `StorageParameters` instances can be set for the different handlers. This can be seen in Listing 7.1.

Listing 7.1: Setting the parameters for the `StorageParameters` instances in the Watchdog

```
1  /* PEHModulePinger */
2  storage = new StorageParameters(80, new long[] { 53 }, 0, 0);
3  ...
4  /* PEHModuleResponseChecker */
5  storage = new StorageParameters(0, new long[] { 14 }, 0, 0);
6  ...
7  /* PEHSystemRecovery */
8  storage = new StorageParameters(0, new long[] { 11 }, 0, 0);
9  ...
10 /* RouteHandler */
11 storage = new StorageParameters(0, new long[] { 32 }, 0, 0);
12 ...
13 /* ISRHandler */
14 storage = new StorageParameters(0, new long[] { 27 }, 0, 0);
```

CHAPTER

EIGHT

REFLECTION & FUTURE WORK

In the previous chapters we introduced the worst-case memory consumption analysis tool SpideyBC, and performed an evaluation of this on two use cases. In this chapter we reflect on selected parts of the process, restrictions in the approach taken and how the analysis could be done differently. Furthermore we present ideas for future work primarily in relation to the implemented tool but also for analysing SCJ programs in general.

8.1 Reflection

This section will reflect on some of the most significant choices and the tool itself.

8.1.1 Model Checking

Our approach to our initial problem was based on looking for techniques for determining WCET as these could be adapted to memory usage instead of execution time. As IPET is a dominant approach in this area, we utilised IPET in terms of calculating the worst-case dynamic memory consumption of a set of entry methods. However, the possibilities in applying model checking would have been an interesting approach as well. In terms of working specifically towards garbage collection in SCJ, model checking could have been an approach where a model of a system with a garbage collector could be used to verify the safety of an application. We still believe that IPET is a suitable solution for SpideyBC, however, note that it could be extended to use model checking.

8.1.2 Evaluation

For evaluating SpideyBC, we ran the tool on two SCJ use case applications – our Watchdog and the RepRap in the JOP repository. The main focus in the evaluation was on dynamic memory analysis and revolved around analysing the application handler methods in order to be able to specify the total backing store sizes of each handler. To verify the produced dynamic memory result of a handler, we investigated the source code in conjunction with considering the generated report information. Furthermore on this basis, we were able to justify the results. We are aware that this is an informal approach that should entail some degree of scepticism, however, as no other formal memory analysis tool is available to our knowledge for SCJ applications, this was considered a suitable approach, also considering the relatively small scale programs. Note that there has been work towards integrating analysis functionality in the WCA tool for determining worst-case heap allocations (WCHA) in RTSJ appli-

cations, however, as we were unable to find this functionality by inspecting the WCA source code, this will not be considered.

The JVM stack analysis results were not elaborated in great detail – this is a result of the difficulty of verifying that the produced sizes actually are the worst-case JVM stack sizes. The naive approach of simply following method invocations in the source code whilst adding the frame sizes for all program paths is a brute force approach in which it becomes necessary to consider all paths individually. For non-trivial programs this is not a viable technique.

By reflecting on the evaluation, we would have liked to have spent more effort on validating the results and the results produced by SpideyBC in general. Ideally, it is desirable to show that the analysis is in fact sound.

8.1.3 Analysing the SCJ Infrastructure

With SpideyBC, we are able to provide a safe upper bound on the worst-case memory consumption of an event handler. As a consequence, we are able to specify the storage parameters (`StorageParameters`) instance that is necessary in the constructor call of a user defined implementation of an event handler (periodic or aperiodic). Besides being able to specify the total backing store of a handler scope, we can parameterise the JVM stack size of the underlying thread in which the handler is bound. For the tool to be fully applicable for enabling developers to specify all remaining memory areas, namely `missionMemorySize`, `immortalMemorySize` and the `StorageParameters` instance for the mission sequencer in the `Safelet` implementation's `getSequencer` method, other methods than just the handlers must be analysed as well.

As an example, for the `missionMemorySize`, a missions `initialize` method must be analysed as the objects allocated in this scope are allocated in Mission memory. In this method, all participating event handler objects are instantiated, which involves calling the constructors. These constructors must immediately call their base class constructors with the supplied arguments, which are defined within the infrastructure. Also, each handler is registered to the mission with a call to `register`, which is a base class method on either `PeriodicEventHandler` or `AperiodicEventHandler` depending on the handler type. To be able to analyse the `initialize` method of a mission, the methods (and their referenced methods) in the native SCJ classes must be analysed as well. This entails adding infrastructural allocated types (primordial) to the model and creating source code annotations for loops and arrays. In case the SCJ types are linked to the application in a binary format, this would not be possible. If the source code is available and editable, there are still many challenges to overcome. First, there are numerous SLOC to consider as SCJ is rather large as it also uses RTSJ related classes. The biggest challenge is to be able to specify the required annotations as these often depend on the input originating from the user application. In other words, as the infrastructural annotations are application dependent, analysing another SCJ application requires changing many of the annotations to match the application under analysis.

8.1.4 Analysing the Standard Library

During the implementation part, we encountered an issue concerning the `String` object within the standard library. Creating a `String` variable, creates a constant in the constant pool and is thus treated like any other primitive type. Concatenating two strings, however, translates the compiled Java code into creating a `StringBuilder` instance that is initialised with the first `String` and on which `append` is called with the second `String`. In the source code of the `append` method, a new underlying array is created with twice the size of the previously held `String`, hereby requiring an annotation. Upon copying the previous `String` content into the new array, several loops are

involved which must also be annotated. Given that the source code is available such that it can be annotated, if the SCJ application manipulates several `String` objects, the library annotations must be set according to the worst-case `String` with respect to its length. This will in turn, always result in a pessimistic allocation cost.

The `String` concatenation example is an example of an implicit use of the standard library that must be taken care of. Similarly, explicit usage of the standard library must also be handled.

8.2 Future Work

In this section we present proposals for future work on the tool and a particular topic related to general program analysis of SCJ applications.

8.2.1 Increase Precision in Loops

As the tool provides a safe upper bound, the next step is to increase the precision by tightening the bound. The majority of execution time is spent in loops[4], thus it makes sense to look at how to optimise this. Consider the program in Listing 8.1 with a single loop.

Listing 8.1: Example loop with a "costly" path that will only execute in one iteration

```
1 public void loopingMethod() {
2     for(int i = 0; i < 20; i++) {
3         if (i == 5) {
4             createManyObjects();
5         } else {
6             createSingleObject();
7         }
8     }
9 }
```

Assume that the branch in Line 4 results in a higher cost than that of Line 6. However, the first branch will only be executed one time during the loop. With IPET using the flow and loop constraints, the most expensive path in a loop will be evaluated as if it was executed during each iteration. Concretely this means that in the example program, the resulting cost of the loop will be 20 (the loop bound) times the cost of the `createManyObjects` method invocation.

Figure 8.1 illustrates a possible CFG for the program in Listing 8.1, where BB_3 and BB_4 corresponds to the `createManyObjects` and `createSingleObject` invocations respectively.

In addition to the set of constraints that would currently be derived, we are interested in adding two more flow constraints as follows:

$$\begin{aligned} f_2 &= f_4 \leq 1 \\ f_3 &= f_5 \leq 19 \end{aligned} \tag{8.1}$$

Here we tighten the bound of the analysis by providing more context about the program. Because such additional constraints can improve the precision significantly, it should be examined how this can be done and incorporated into the tool. Ideally this should be derived by the tool given the program as input, e.g. by the use of data-flow analysis, however, the programmer could also provide hints for the analysis.

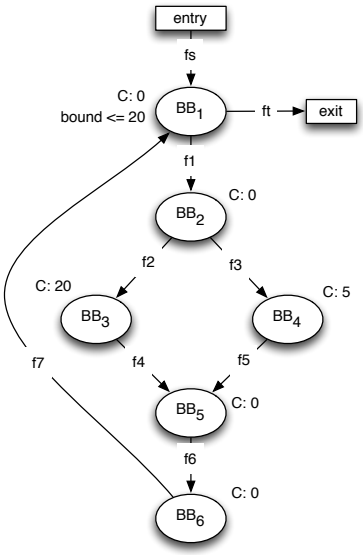


Figure 8.1: CFG for the method in Listing 8.1

8.2.2 Increase Precision at Branches

Similar to the loops, branching is a significant part the programs and it is desirable to tighten the bound by looking at these. During the execution, different paths may be mutually exclusive or always be executed together, despite not being part of the same branching block. Consider the program in Listing 8.2 that stores sensor readings or reports an error if a sensor is deemed to have failed if the value is above a certain threshold.

Listing 8.2: Example program storing sensor readings

```
1  public void processSensorMeasurements(double[] measurements) {
2      bool sensorFailed = false;
3      for(double value : measurements) {
4          if (value > 100.0) {
5              sensorFailed = true;
6          }
7          else {
8              storeResult(value);
9              /* Do something more */
10         }
11
12         if (sensorFailed) {
13             generateError();
14             break;
15         }
16     }
17 }
```

In the program we can see that Line 5 is always executed with lines 13 and 14. Similarly, Line 8 is mutually exclusive with lines 13 and 14. Such cases have previously been described and examined[18].

Let e_i and e_j denote the execution frequency of BB_i and BB_j respectively. In the case that BB_i and BB_j are always executed together, the authors suggest the following constraint be added to the set[18]:

$$e_i \quad = \quad e_j \tag{8.2}$$

Mutual exclusion between blocks are more difficult but also provides the real increase in precision. Again, let e_i and e_j denote the execution frequency of BB_i and BB_j. In the case that BB_i and BB_j are mutually exclusive, the constraint are as follows[18]:

$$(e_i = 0 \;\&\; e_j = 1) \quad | \quad (e_i = 1 \;\&\; e_j = 0) \tag{8.3}$$

The trade off for this approach is that we now have two actual sets, because the constraint in 8.3 is not a linear constraint. Thus we must transform the set of constraints into two sets that splits up the disjunction. As more of these disjunctive constraints are added, the amount of constraint sets that must be solved increases exponentially.

Similar to the improvements in loops, it would be ideal if the tool could derive these constraints whenever possible. With the trade off of having to now solve more sets, it could also be desirable to support turning such features on and off.

8.2.3 Generalising the Architecture Input Model

Currently the input model must contain memory sizes for each type that is allocated in some path starting from the entry points. We showed how these were derived for the JOP, however, this can be considered cumbersome for the programmer.

Alternatively, it would be desirable to look into other ways of handling the low-level details. As an example, the SCJ specification defines the `SizeEstimator` class[34] that can provide upper memory bounds on class instantiations on a concrete platform and its SCJ implementation. This is, however, currently not implemented on JOP, but in a complete SCJ version this can be considered a way to automate this process. Additionally, from the specification, this is emphasised to only be an estimate, thus resulting in a further imprecise analysis. Another approach could be to look into how a more general model of the underlying architecture could be specified with enough details to calculate type sizes algorithmically.

8.2.4 Synthesis of Analysis Results

Currently the analysis results are provided for the programmer through the analysis report that is the final output of the tool. We previously described how the programmer could specify safelet and mission initialisation methods as well as handlers as entry points and aggregate the results for setting all storage parameters. In the evaluation we saw how the results were used to specify storage parameters for the handlers of the Watchdog application.

Instead, the tool could perform synthesis by combining the analysis results and the input program and produce Java code where the storage parameters are filled in. In a more extreme approach, the tool could perform bytecode manipulation and produce the direct object code that would be equivalent to having compiled the SCJ application with the storage parameters specified.

8.2.5 Support for Recursion

In Section 6.1.4 the restrictions for programs under analysis were listed in which we required that there be no recursion. This requirement could be lifted, however, by the use of a same approach as we saw with bounding loops. The tool could be extended with the support for bounded recursion, e.g. by annotating the recursive invocation with the maximum depth. While recursive algorithms can be transformed to use iteration (and vice versa), it can be desirable to have recursion for a more declarative code.

8.2.6 Analysis of Standard Libraries and Infrastructure

As we described, analysis of standard libraries (in particular related to strings) and the infrastructure is troublesome. Not only for this analysis, but we also experienced this when performing WCET analysis of the Watchdog application. One particular issue is the need for specifying loop bounds, and in a standard library this must be set to the highest possible value of all parts of the application that use the particular piece of code. As a result, precision is lost. Current attempts towards analysing code dependent on standard libraries and underlying infrastructure can be considered "defensive", as in the approach is usually by trying to work around these, exclude them completely or make very pessimistic assumptions.

We argue that any work done towards enabling analysis of standard libraries and infrastructure is beneficial as it will both assist in the analysis we have worked with but also for e.g. WCET analysis. One such approach to facilitate the analysis can be suggested as follows that uses existing Java features. Consider the example implementation of some library method in Listing 8.3. The loop bound is parametrised rather than provided directly at the iteration site. The actual value will be provided at each call site, such that one bound does not need to cover all possible call sites.

Listing 8.3: Suggestion for parametrisation of loop bounds in libraries

```
1 class StdApi {
2     public static void someStdLibMethod(string[] values) {
3         for(string value : values) { //@ loop bound = <arg_length_bound>
4             // Do something with value
5         }
6     }
7 }
```

Listing 8.4 illustrates examples of two clients of the method, where the bound is provided using Java annotations.

Listing 8.4: Suggestion for providing arguments to loop bound parameters using Java annotations

```
1 public void someUserOfStdLib(string[] values) {
2     String[] names = new String[20];
3     ...
4     @Bounds(arg_length_bound=20)
5     StdApi.someStdLibMethod(names);
6 }
7
8 public void someOtherUserOfStdLib(string[] values) {
9     String[] fruits = new String[10];
10    ...
11    @Bounds(arg_length_bound=10)
12    StdApi.someStdLibMethod(fruits)
13 }
```

CHAPTER

NINE

CONCLUSION

During this master's thesis we worked on the following problem:

> *How can the principles of static program analysis be applied for resource analysis of Java applications?*

The problem was delimited to the focus on worst-case memory consumption based on problems that were highlighted in our previous work. In SCJ the developer must manually specify sizes of scoped memory regions and the JVM stack for each handler. This requires that the programmer manually inspect the source code to provide these sizes which is error-prone. Additionally, a new version of the DO-178B standard that has been a target standard to certify applications against in the design of SCJ, has opened up for the use of garbage collection. This also requires knowledge on the memory consumption of the application.

The problem was approached by looking at static program analysis in general and more concretely, how it is used to perform worst-case execution time analysis. As a result, we were able to implement the tool *SpideyBC* that performs the analysis on SCJ applications at the bytecode level. The problem of determining an exact static analysis of memory consumption is undecidable. Because we are interested in a safety property, the tool provides an over-approximation that is a safe upper bound. Furthermore, in order to work with finite programs, restrictions were imposed on programs that can be analysed.

In general static program analysis, we discovered that by using control flow graphs, it is possible to construct a representation of the possible execution flows of a program. In order to handle dynamic dispatching a pointer analysis can be used to analyse possible targets of a reference.

For determining the worst-case memory consumption in terms of dynamic memory allocation for the memory regions, we were able to use the widely popular IPET approach from WCET analysis. With this approach, the problem of determining a safe upper bound on dynamic memory allocation was transformed into an integer linear programming problem using the program representation. This could then be solved using any available implementation of an algorithm for solving linear programming problems.

For determining a safe upper bound on the JVM stack size, the analysis results of the pointer analysis could be used. By using a call graph that represents method invocation relationships and the information on stack requirements for a method, that is available at compile-time, the upper bound was determined by finding the most expensive path through the call graph.

The tool was evaluated using two SCJ use cases. We found that the tool is able to use the implemented techniques to perform the desired analysis.

BIBLIOGRAPHY

[1] LPsolve, 2013. Binaries and source code available at http://sourceforge.net/projects/lpsolve/.

[2] SpideyBC – Tool for Static Memory Analysis of Java Bytecode, 2013. Git repository available at https://github.com/jlandersen/sw10/tree/master/Code.

[3] WALA – T.J Watson Libraries For Analysis, 2013. The official WALA Wiki documentation available at http://wala.sourceforge.net/wiki/index.php/Main_Page/.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] F. E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[6] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. Technical report, 1994.

[7] A. Burns. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach. In *Advances in Real-Time Systems, chapter 10*, pages 225–248. Prentice Hall, 1994.

[8] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*.

[9] A. E. Dalsgaard, R. R. Hansen, and M. Schoeberl. Private Memory Allocation Analysis for Safety-Critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 9–17, New York, NY, USA, 2012. ACM.

[10] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustavsson, and H. Hansson. Towards Industry Strength Worst-Case Execution Time Analysis, 1999.

[11] J. Engblom, A. Ermedahl, and F. Stappert. Comparing Different Worst-Case Execution Time Analyis methods. In *Work-in-Progress session of the 21st IEEE Real-Time Systems Symposium*, RTSS 2000, Orlando, Florida, USA, 2000.

[12] Federal Aviation Administration. Guidelines for Approving Source Code to Object Code Traceability, 2002. This is an electronic document. Date of publication: December 1, 2002. Date retrieved: November 28, 2012. Document available at http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-12.pdf.

[13] Federal Aviation Administration. Software Approval Guidelines, 2003. This is an electronic document. Date of publication: June 3, 2003. Date retrieved: November 28, 2012. Document available at http://www.airweb.faa.gov/Regulatory_and_Guidance_Library/rgOrders.nsf/0/640711b7b75dd3d486256d3c006f034f/$FILE/Order8110.49.pdf.

[14] C. Frost, C. S. Jensen, B. Thomsen, and K. S. Luckow. WCET Analysis of Java Bytecode Featuring Common Execution Environments. Master's thesis, Aalborg Universitet, 2011.

[15] L. Hendren, P. Lam, J. Lhoták, O. Lhoták, and F. Qian. Soot, A Tool for Analyzing and Transforming Java Bytecode, 2003. Date retrieved: May, 2012. Slides available at http://www.sable.mcgill.ca/soot/tutorial/pldi03/tutorial.pdf.

[16] V. Hilderman and T. Baghi. *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)*. Avionics Communications, Incorporated, 2007.

[17] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[18] Y. S. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[19] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the Impact of Context-Sensitivity on Andersen's Algorithm for Java Programs. *SIGSOFT Softw. Eng. Notes*, 31(1):6–12, Sept. 2005.

[20] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification - Java SE 7 Edition, 2012.

[21] P. Lokuciejewski and P. Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer, 1 edition, 2011.

[22] Oracle. javac – Java Programming Language Compiler, 2013. Documentation available at http://docs.oracle.com/javase/6/docs/technotes/tools/windows/javac.html.

[23] OW2 Consortium. ASM, 2013. Binaries and source code available at http://asm.ow2.org/.

[24] P. Puschner and A. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *Journal of Real-Time Systems*, 13:67–91, 1997.

[25] P. P. Puschner and C. Koza. Calculating the Maximum, Execution Time of Real-Time Programs. *Real-Time Syst.*, 1(2):159–176, Sept. 1989.

[26] D. Rayside. Points-To Analysis. University lecture notes., 2005. Date retrieved: April, 2012. Document available at http://www.cs.utexas.edu/~pingali/CS395T/2012sp/lectures/points-to.pdf.

[27] M. Schoeberl. JOP: A Tiny Java Processor Core for FPGA, 2008. Date retrieved: December 10, 2012. Date last modified: February 24, 2008. Web site available at http://www.jopdesign.com.

[28] M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, 2009. This is an electronic document. Document available at http://www.jopdesign.com/doc/handbook.pdf.

[29] M. Schoeberl and R. Pedersen. WCET Analysis for a Java Processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '06, pages 202–211, New York, NY, USA, 2006. ACM.

[30] M. Schoeberl and J. Vitek. Garbage Collection for Safety-Critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '07, pages 85–93, New York, NY, USA, 2007. ACM.

[31] M. I. Schwartzbach. Lecture Notes on Static Analysis. 2009.

[32] T. B. Strøm and M. Schoeberl. A Desktop 3D Printer in Safety-Critical Java. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 72–79, New York, NY, USA, 2012. ACM.

[33] The Apache Software Foundation. Apache Commons BCEL, 2013. Binaries and source code available at http://commons.apache.org/proper/commons-bcel/.

[34] The Open Group. Safety-Critical Java Technology Specification, 2012. Internal document dated December 2012.

[35] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, 2004.

[36] M. Todberg and J. L. Andersen. A Study of Safety-Critical Java and its Specification Applied, 2012.

[37] M. Todberg and J. L. Andersen. Cubesat Space Protocol and Watchdog Implementations in Safety-Critical Java, 2012. Git repository available at https://github.com/Todberg/SW9/tree/master/Code.

[38] B. Wichmann, A. Canning, D. Clutterbuck, L. A. W. , N. J. Ward, and D. W. R. Marsh. Industrial Perspective on Static Analysis. *Software Engineering Journal*, 10(2):69–75, 1995.

[39] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[40] W. Wögerer. A Survey of Static Program Analysis Techniques. 2005.

APPENDIX

A

ANDERSENS ALGORITHM FOR C PROGRAMS

Let $malloc{-}i$ denote a pointer target for an allocation site i. The result is the function $pt(p)$ that for a pointer variable p returns the set of possible pointer targets to which p may point to. For each variable id in the program, $[\![id]\!]$ denotes the set of possible targets to which id may point. The analysis assumes that all pointer manipulations in the program are one of the following six kinds (other types of pointer manipulation can be normalised to these)[31]:

1. $id = malloc$
2. $id_1 = \&id_2$
3. $id_1 = id_2$
4. $id_1 = *id_2$
5. $*id_1 = id_2$
6. $id = null$

The following constraints are specified for each of these pointer manipulations, that is used to create the points-to graph[31]:

$$\begin{aligned} id = malloc: &\quad \{malloc{-}i\} \subseteq [\![id]\!] \\ id_1 = \&id_2: &\quad \{\&id_2\} \subseteq [\![id_1]\!] \\ id_1 = id_2: &\quad [\![id_2]\!] \subseteq [\![id_1]\!] \\ id_1 = *id_2: &\quad \&id \in [\![id_2]\!] \Rightarrow [\![id]\!] \subseteq [\![id_1]\!] \\ *id_1 = id_2: &\quad \&id \in [\![id_1]\!] \Rightarrow [\![id_2]\!] \subseteq [\![id]\!] \end{aligned}$$

The resulting points-to function is then defined as the following:

$$pt(p) \quad = \quad [\![p]\!]$$

The general intuition behind these constraints, is that in the case that a pointer variable v_1 points to another pointer variable v_2, all possible targets of v_2 are possible targets of v_1. We refer to [31] that provides details on *The Cubic Algorithm* that can be used for creating the points-to graph based on these constraints. The worst-case time complexity of creating and solving these constraints is $O(n^3)$.

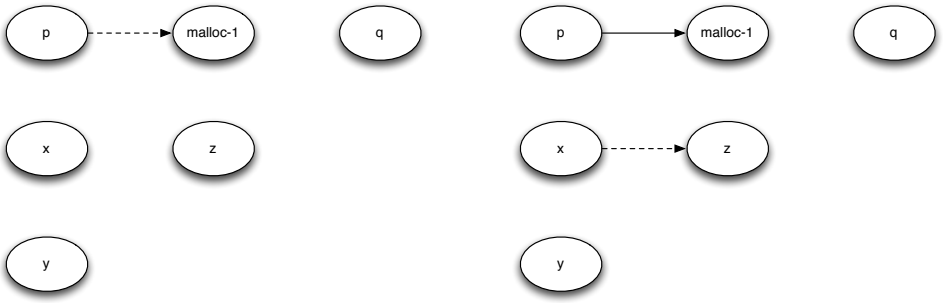
Consider the following sequence of statements with pointer manipulation:

```
var p, q, x, y, z;
p = malloc
x = &z
y = x
q = p
p = *y
```

Assume we are interested in knowing possible targets of q. That is, we wish to know the set returned by the function $pt(q)$. The following constraints are produced for the program:

$$malloc{-}1 \subseteq [\![p]\!]$$
$$\{\&z\} \subseteq [\![x]\!]$$
$$[\![x]\!] \subseteq [\![y]\!]$$
$$[\![p]\!] \subseteq [\![q]\!]$$
$$\&x \in [\![y]\!] \Rightarrow [\![x]\!] \subseteq [\![p]\!]$$

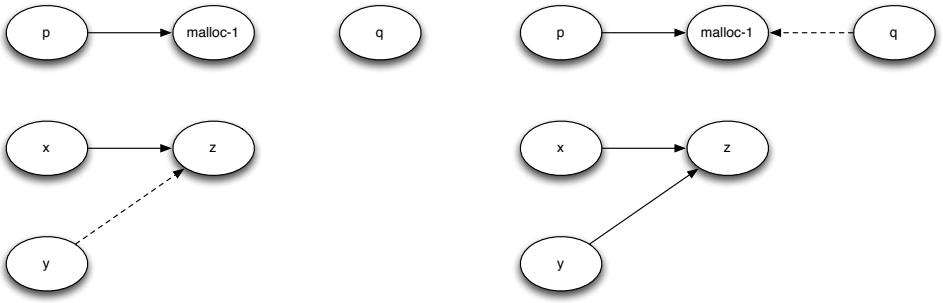
Let us consider how the resulting points-to graph is constructed, under these constraints. Figure A.1(a) illustrates the trivial inclusion of the newly allocated target as a possible target for the variable p (dashed edges indicates the new relationship between pointers). This is a result of the first constraint. From the second constraint, Figure A.1(b) illustrates the effect in the points-to graph. As x is assigned the address of z, z becomes an element in the set of possible targets for x.



(a) From constraint $malloc{-}1 \subseteq [\![p]\!]$ (b) From constraint $\{\&z\} \subseteq [\![x]\!]$

Figure A.1

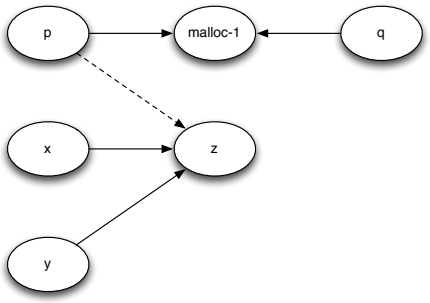
The third constraint makes z a possible target of y as illustrated in Figure A.2(a). The fourth constraint is similar, with $malloc{-}1$ becoming a possible target of q.



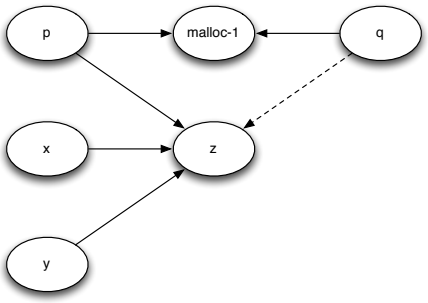
(a) From constraint $[\![x]\!] \subseteq [\![y]\!]$ (b) From constraint $[\![p]\!] \subseteq [\![q]\!]$

Figure A.2

The fifth constraint is the most interesting one. Because we make an assignment of the value produced by dereferencing y to p, this requires all possible targets of y to become possible targets of p. This is illustrated in Figure A.3(a). However, from the fourth constraint, this also makes these new targets of p possible targets of q as illustrated in Figure A.3(b), which is also the final points-to graph.



(a) From constraint $\&x \in [\![y]\!] \Rightarrow [\![x]\!] \subseteq [\![p]\!]$



(b) (Continued) From constraint $\&x \in [\![y]\!] \Rightarrow [\![x]\!] \subseteq [\![p]\!]$

Figure A.3

As a result, we can here see the solution to what possible targets of q are:

$$pt(q) = [\![q]\!] = \{malloc{-}1, z\}$$

APPENDIX

B

LIBCSP BASED WATCHDOG IN SCJ

This chapter briefly describes the CSP based Watchdog from our previous work, which was used in the tool evaluation described in Chapter 7. In total, three watchdog variants have been created that can be found on GitHub[37]:

Watchdog (Level 0) Implementation conforming to compliance level 0 under a cyclic executive scheduler.

Watchdog (Level 1) Implementation conforming to compliance level 1 under a fixed-priority scheduler.

Watchdog (Level 1) CSP Same as the above implementation, but instead of embedding I^2C communication directly into the software, the developed CSP library is utilised, enabling the possibility of using different communication interfaces depending on the hardware configuration.

The level 1 CSP based version that will be described in this chapter is based on modifying the non CSP based level 1 version.

B.1 Introduction

The Watchdog is based on the following problem setting. In a distributed system multiple modules communicate to accomplish a task as seen in Figure B.1. In case of failure in one or more modules, the system should take appropriate actions as other modules may be dependent on the failed module(s). The Watchdog has the single goal of detecting such failures of any module and take appropriate action.

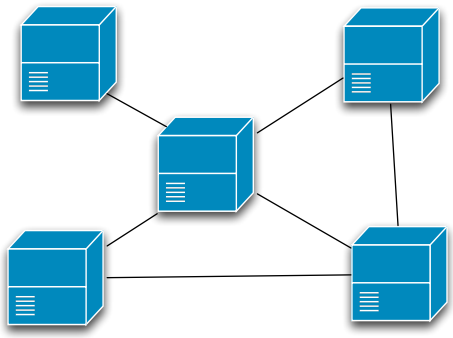


Figure B.1: Five separate modules working on a common task in a distributed setting.

B.2 Tasks and Temporal Requirements

The tasks required for the Watchdog depends on the communication flow. The Watchdog can be designed to be the master such that it initiates and monitors each module, or each module can act as master that regularly contacts and resets a timer in the Watchdog. A multi-master approach in this distributed setting with modules contacting the Watchdog, could lead to starvation of one or more modules due to repeated bus congestion, leading (incorrectly) to the Watchdog assuming a module failure.

Instead the original Watchdog was designed to act as the sole master on the bus, in which failure of each module on the bus is checked one by one. Therefore the Watchdog consists of three periodic event handlers performing the following duties:

Pinger This task periodically pings a number of registered modules and stores the response (if any) in a shared data structure.

Checker This task checks all module responses in the shared data structure and sets a recovery flag in case a module has failed to reply.

Recovery This task executes a recovery routine taking some user-defined appropriate action if the recovery flag it set.

Table B.1 shows the tasks along with related scheduling assignments. In addition to the application specific tasks, the CSP implementation extends the set with a routing task and one for incoming packets on the I^2C bus.

Task	Type	Period	Deadline	Priority
Pinger	Periodic	500	200	5
Checker	Periodic	500	100	10
Recovery	Periodic	500	50	15
Routing (CSP)	Periodic	5	3	20
I^2C (CSP)	Aperiodic	-	-	25

Table B.1: Task Set

The periods of the tasks are assigned such that the Watchdog begins its monitor cycle every half second. We assign priorities according to deadline monotonic priority ordering. We argue that despite deadline enforcement and detection not being available in SCJ, it is suitable to consider deadlines due to the precedence relationship between the tasks - not only must both the *Pinger* and *Checker* tasks be done executing when either is to be released at a new period, the *Pinger* task must run before the *Checker* which we in turn want to run before the *Recovery* handler. The deadline of the *Pinger* is set on the pessimistic assumption that each module communication times out 10 ms after the Watchdog initiates communication with it (in which case it is considered failed). In the worst case where the maximum of 10 modules times out, 100 ms will be spent on this alone. In order to correct the priorities and take the precedence relationship into account, we need the *Checker* and *Recovery* handlers to enforce these to run in a serial manner, despite all being assigned the same period. As in Burns [7] we do this by stretching the deadline of the *Checker* and *Recovery* handler tasks and relate their deadline to the start of the transaction with the *Pinger*, and not their own period and create a new transformed task set as listed in Table B.2. An alternative approach could be to provide the *Checker* and *Recovery* handlers with an initial offset, which would avoid creating a critical instant. Note that this transformed task set also includes initial estimated computation times, that the implementation must not exceed (based on a pessimistic assumption for each task).

Task	Type	Period	Computation Time	Deadline	Priority
Pinger	Periodic	500	150	200	15
Checker	Periodic	500	20	300	10
Recovery	Periodic	500	40	350	5
Routing (CSP)	Periodic	5	4	5	20
I^2C (CSP)	Aperiodic	-	2	3	25

Table B.2: Transformed Task Set

B.3 Modifying the Watchdog to use CSP

The non CSP based version was implemented first, which handles I^2C communication directly as a part of the application. This section assumes that the reader is aware of implementation specific details in the previous version. We refer to the listed GitHub repository for this.

To incorporate the CSP library in the Watchdog, all communication classes dealing with I^2C and their respective packages are removed - i.e. `sw901e12.comm` and `sw901e12.comm.modules`. The field, `receivedResponseOnLastPing`, originally located on the now deleted `ModulePinger` is transferred to the `Module` class itself. This class is also extended with fields for `CSPAddress`, `CSPPort` and `MACAddress`. In the mission, an array of modules are created with different properties depending on the execution context (board or simulator). The registration process of a module for the simulator and the board can be seen in Listing B.1.

Listing B.1: Module for the simulator and the board.

```
1  /* Simulator Module */
2  int MACAddress = Config.MAC_ADDRESS;
3  int CSPAddress = Config.CSP_ADDRESS;
4  int CSPPort = 0x01;
5  slaves[0] = Module.create("Module 1", MACAddress, CSPAddress, CSPPort);
6  IMACProtocol loopbackInterface = InterfaceLoopback.getInterface();
7  loopbackInterface.initialize(MACAddress);
8  manager.routeSet(CSPAddress, loopbackInterface, MACAddress);
9
10 /* Board Module */
11 int MACAddress = 0x01;
12 int CSPAddress = 0x01;
13 int CSPPort = 0x01;
14 slaves[0] = Module.create("Module 1", MACAddress, CSPAddress, CSPPort);
15 IMACProtocol I2CInterface = InterfaceI2C.getInterface();
16 I2CInterface.initialize(MACAddress);
17 manager.routeSet(CSPAddress, I2CInterface, 0xFF);
```

When run in the simulator, the `MACAddress` and `CSPAddress` values for each module remains the same and are fetched from a `Config` class (its source and destination addresses are always the node is itself). The `CSPPort` value, however, must vary in order to distinguish between connections. Next the module is created with the mentioned parameters and added to the array of slave modules. The interface is set to Loopback with the same `MACAddress` and the node registered in the routing table. When run on the board, the `MACAddress` and `CSPAddress` must have different addresses than that of the Watchdog, as the modules no longer reside within the same application. Now it is also necessary to supply a proper next hop MAC address to the requested interface based on the network topology (in the example, 0xFF). The implementation of the `handleAsyncEvent` is changed to use the CSP API. The changed *Pinger* can be seen in Listing B.2.

Listing B.2: The Pinger handler.

```
@Override
@SCJAllowed(Level.SUPPORT)
public void handleAsyncEvent() {
  if (Config.DEBUG) {
    console.println("PEHModulePinger");
  }
  for (Module slave : slaves) {
    conn = manager.createConnection(slave.getCSPAddress(), slave.
        getCSPPort(), CSPManager.TIMEOUT_SINGLE_ATTEMPT, null);

    if(conn != null) {
      packet = manager.createPacket();
      packet.setContent(42);
      conn.send(packet);
      Packet response = conn.read(10);
      slave.setResponse(response != null ? true : false);
      conn.close();
    }
  }
}
```

In the old implementation of the *Pinger* handler, the `ModulePinger` object for each slave was fetched and its `ping` method invoked as this would know the correct procedure of how to ping the particular slave device. This is no longer necessary when using CSP, as the protocol handles this internally using various implementations of MAC-layer protocols. The updated *Checker* handler can be seen in Listing B.3.

Listing B.3: The Checker handler.

```
@Override
@SCJAllowed(Level.SUPPORT)
public void handleAsyncEvent() {
  if(Config.DEBUG) {
    console.println("PEHModuleResponseChecker");
  }
  for (Module slave : slaves) { // @WCA loop<=10
    if(slave.getResponse() == true) {
      slave.resetResponse();
    } else {
      mission.executeRecovery = true;
      break;
    }
  }
}
```

The *Checker*, has not changed much apart from going directly to the slave module in order to check the response flag. Finally the *Recovery* handler can be seen in Listing B.4.

Listing B.4: The Recovery handler.

```
@Override
public void handleAsyncEvent() {
  if(Config.DEBUG) {
    console.println("PEHSystemRecovery");
  }

  if(mission.executeRecovery) {
    if(Config.DEBUG) {
      console.println("Initiating system recovery...");
    }
    recovery.executeRecovery();
  }
```

```
13 }
```

This handler did not require any changes and could be used as is. Finally, it should be noted that in addition to the original three periodic event handlers, the routing handler is now also present. This has a period of 5 ms, and runs with the highest priority (20). Because the routing handler is not a part of the precedence relationship between the original tasks, the deadline of the routing handler is the same as its period.

APPENDIX

C

ANALYSIS REPORT

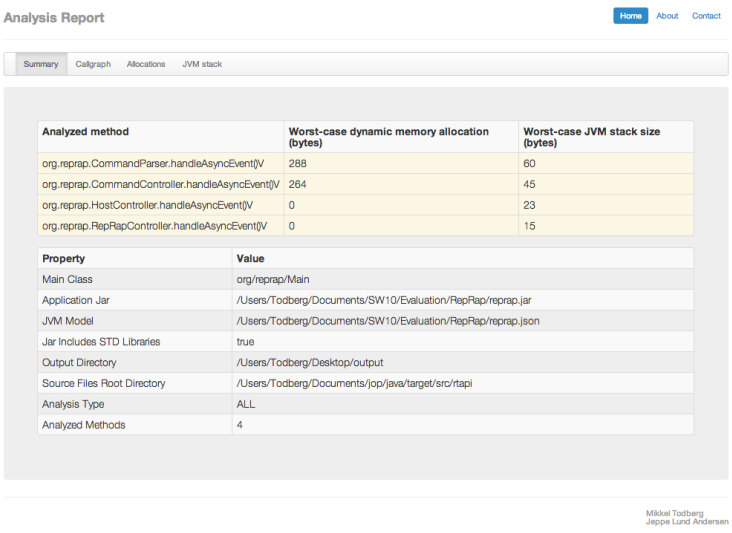


Figure C.1: Summary page with results and tool arguments

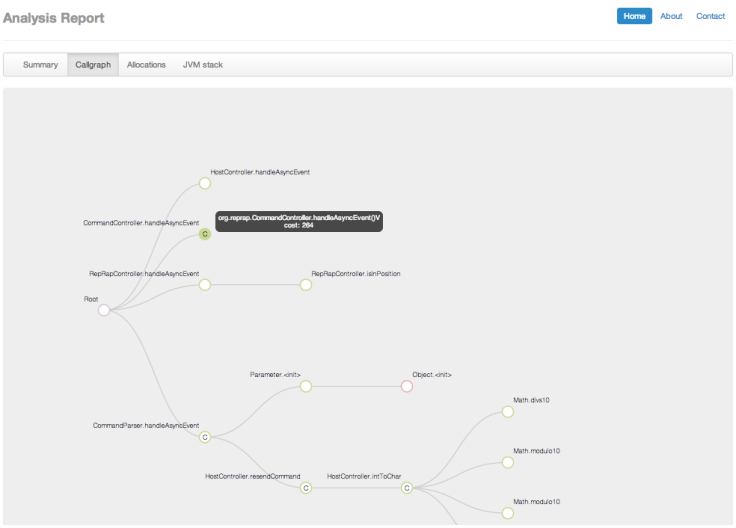


Figure C.2: Callgraph page with direct source code access

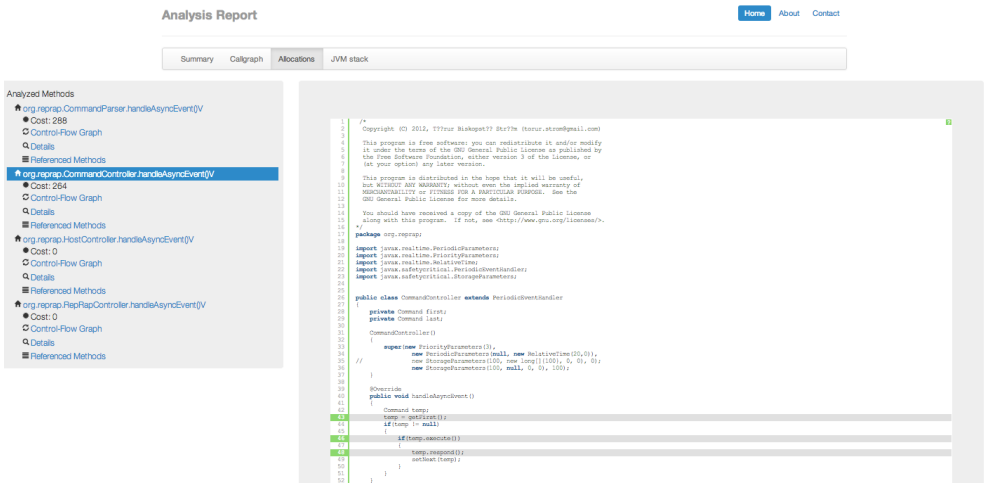


Figure C.3: Allocations page showing affected source code lines

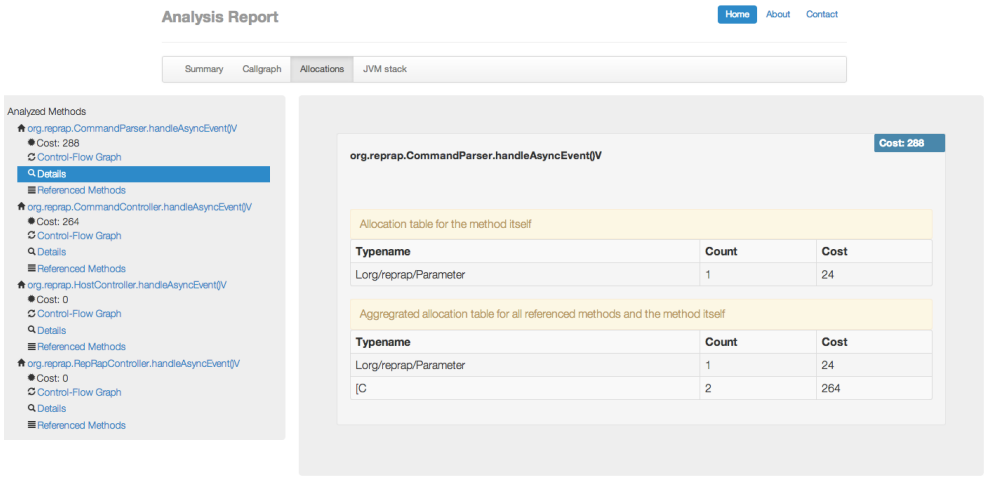


Figure C.4: Allocations page showing the actual allocations and origin

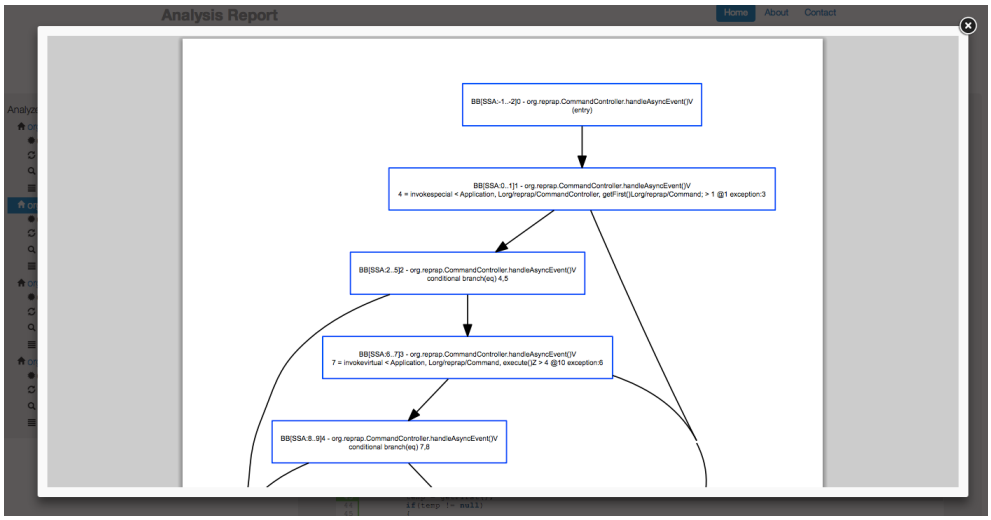


Figure C.5: Visual control flow graph of an analysed method

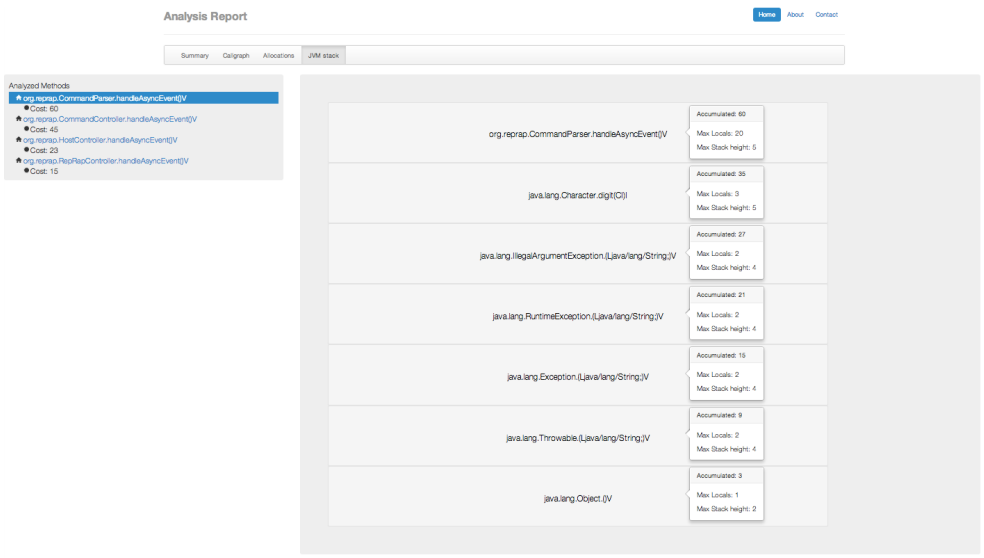


Figure C.6: Worst-case stack page (with details) for the analysed methods

APPENDIX

D

SUMMARY

This project describes the development of the tool *SpideyBC* – a worst-case memory consumption (WCMC) analysis tool for Java bytecode applications. The goal of the project is to show how an established static analysis technique for worst-case execution time (WCET) analysis can be used for memory analysis.

Java is a popular programming language, and in the recent years, attempts have been made to make use of Java technology in the area of real-time and safety-critical systems development. Safety-Critical Java (SCJ) is a restricted version of Java that aims to support the development of applications in Java that can be certified against rigid standards. A key difference in the SCJ specification is the replacement of automatic memory management through a garbage collected heap with a scoped memory model. This provides predictable and analysable memory allocation and deallocation behaviour, but requires the developer to reason about memory usage. Sizes of memory regions and JVM stacks must be explicitly stated in the programs for the underlying infrastructure.

To prevent the developer from having to intertwine analysis code with application code for WCMC analysis, we use static analysis techniques as the foundation. By using static program analysis techniques, the applications being analysed are never actually executed. Two types of static analysis are performed on the input application – dynamic memory and JVM stack sizes. As part of the input, the developer provides a set of methods that acts as the analysis entry points. The two types of analysis are then performed for each of the entry points.

The fundamental technique for the analysis is to provide a safe upper bound on allocation of dynamic memory, which is done using the well known *Implicit Path Enumeration Technique* (IPET) that is also applied in the area of WCET analysis. IPET is fundamentally based on transforming the WCET problem into an integer linear problem that considers all possible paths through a program implicitly. By changing the cost of different points in the program representation from execution time to memory allocation sizes, the tool is able to provide safe upper bounds starting from each of the entry points. The developer can use these bounds to state the required sizes of memory regions in the SCJ application, as opposite to doing this through manual inspection of the program. For the analysis of the maximum sizes of JVM stacks, we use the representation of a programs possible method invocations to determine the most expensive stack size.

The result of the project is a tool that performs these analysis. The tool is evaluated by using it on two SCJ use cases. The first use case is a RepRap 3D printer that was presented at the JTRES workshop in 2012. The second use case is constructed and presented as a part of this project. The use case is a watchdog that monitors modules in a distributed system. The Watchdog incorporates the Cubesat

Space Protocol implementation from our previous project. The results show that the tool provides safe upper bounds on the provided input. Using the tool, we have been able to identify the memory parameters required to be specified in the applications. Furthermore, we present directions for future work on the tool namely to tighten the bound provided such that the precision is increased. As a final remark, we would like to emphasise that while the scope of the project has been SCJ applications, working at the bytecode levels presents the possibility of applying it on other types of Java programs.