The University Of Sheffield.

# Verification of hardware dependent software

Submitted for the candidature of the degree of Doctor of Philosophy
Department of Computer Science

*Author:* Ramsay G. Taylor
*Supervisor:* Professor John Derrick

17th January 2012

## Abstract

Many good processes exist for ensuring the integrity of software systems. Some are analysis processes that seek to confirm that certain properties hold for the system, and these rely on the ability to infer a correct model of the behaviour of the software. To ensure that such inference is possible many high-integrity systems are written in "safe" language subsets that restrict the program to constructs whose behaviour is sufficiently abstract and well defined that it can be determined independent of the execution environment. This necessarily prevents any assumptions about the system hardware, but consequently makes it impossible to use these techniques on software that must interact with the hardware, such as device drivers.

This thesis addresses this shortcoming by taking the opposite approach: if the analyst accepts absolute hardware dependence — that the analysis will only be valid for a particular target system: the hardware that the driver is intended to control — then the specification of the system can be used to infer the behaviour of the software that interacts with it. An analysis process is developed that operates on disassembled executable files and formal system specifications to produce CSP-OZ formal models of the software's behaviour. This analysis process is implemented in a prototype called Spurinna, that is then used in conjunction with the verification tools Z2SAL, the SAL suite, and IsabelleHOL, to demonstrate the verification of properties of the software.

## Declaration

I declare that this thesis was composed entirely by myself and that the work contained herein is my own except where explicitly stated otherwise. This work has not been submitted for any degree or professional qualification other than as specified.

**Ramsay Taylor**

# Contents


**1 Introduction** **1**
1.1 Motivation . . . 1
1.2 Objective of this work . . . 2
1.3 Thesis Outline . . . 4

**2 Background** **5**
2.1 Introduction . . . 5
2.2 Verification . . . 5
2.3 Code analysis . . . 6
2.3.1 Overview . . . 6
2.3.2 MISRA C . . . 7
2.3.3 SPARK Ada . . . 8
2.3.4 Hardware verification . . . 9
2.3.5 Summary . . . 9
2.4 Modeling . . . 10
2.4.1 Overview . . . 10
2.4.2 The Z notation . . . 11
2.4.3 B . . . 13
2.4.4 CSP . . . 13
2.4.5 Object Z . . . 15
2.4.6 CSP-OZ . . . 17
2.4.7 Summary . . . 17
2.5 Property verification . . . 18
2.5.1 Overview . . . 18
2.5.2 Hoare logic . . . 19
2.5.3 Separation logic . . . 21
2.5.4 IsabelleHOL . . . 22
2.5.5 SAL . . . 23
2.5.6 Z2SAL . . . 24
2.5.7 Summary . . . 25

**3 The proposed solution** **27**
3.1 Introduction . . . 27
3.2 The analysis process . . . 27
3.3 Formal model . . . 28
3.3.1 System state specification . . . 29
3.3.2 Sequential instruction specifications . . . 30
3.3.3 Branch instruction set specification . . . 34
3.4 Automated inference . . . 38

**4 Separation in Z** **41**
4.1 Overview . . . 41
4.2 Introduction . . . 41
4.3 Basic case . . . 42
4.4 A relaxation of the basic case . . . 44
4.5 A simple function case . . . 46
4.6 A simple relational case . . . 51
4.7 Conclusion . . . 52

**5 Analysis** **53**
5.1 Disassembly . . . 53
5.1.1 Overview . . . 53
5.1.2 Executable files, object code, and disassembly . . . 53
5.1.3 Parsing . . . 57
5.1.4 Summary . . . 59
5.2 Branch Identification . . . 59
5.2.1 Overview . . . 59
5.2.2 Branch instructions . . . 59
5.2.3 Branch template format . . . 60
5.2.4 Example . . . 61
5.2.5 Summary . . . 64
5.3 Formal Instantiation . . . 64
5.3.1 Overview . . . 64
5.3.2 Template specifications . . . 64
5.3.3 Instantiation . . . 67
5.3.4 Summary . . . 67
5.4 Simplification . . . 68
5.4.1 Overview . . . 68
5.4.2 Sequential block compression . . . 68
5.4.3 Algorithm . . . 69
5.4.4 Summary . . . 69
5.5 Program Encapsulation . . . 70
5.5.1 Overview . . . 70
5.5.2 Internal branch instructions . . . 70
5.5.3 Function calls . . . 71
5.5.4 Class definition . . . 72
5.5.5 Summary . . . 72

**6 The Spurinna implementation** **73**
6.1 Overview . . . 73
6.2 Input files . . . 73
6.3 Analysis stages . . . 75
6.3.1 Architecture . . . 75
6.3.2 Branch identification . . . 76
6.3.3 Formal instantiation . . . 77
6.3.4 Simplification . . . 78
6.3.5 Program encapsulation . . . 78
6.4 Output formats . . . 79
6.5 Summary . . . 81

**7 Model checking for verification** **83**
7.1 Overview . . . 83
7.2 System, requirements, and analysis . . . 83
7.3 Z2SAL . . . 84
7.4 Model checking Linear Temporal Logic properties . . . 87
7.5 Model checking for fault detection . . . 89
7.6 Summary . . . 92

**8 Verification by symbolic proof** **95**
8.1 Overview . . . 95
8.2 Symbolic proof in IsabelleHOL . . . 95
8.3 Converting Z to an Isabelle model . . . 97
8.4 Proofs of properties . . . 101
8.5 Proof assistant counterexamples for fault detection and tracing . 102
8.6 Summary . . . 104

**9 Verifying a hardware usage requirement** **105**
9.1 Overview . . . 105
9.2 A hardware interaction example . . . 105
9.3 SAL verification . . . 107
9.4 Summary . . . 110

**10 Conclusions** **111**
10.1 Software analysis . . . 111
10.2 Future work . . . 112

**Bibliography** **119**

**A Flattening CSP-OZ to pure Z** **121**

**B Derived MaxInt CSP-OZ model** **125**

**C MaxInt for import to Z2SAL** **133**

**D MaxInt SAL file** **137**

**E SAL suite counter example for brokenmaxint** **145**

**F Isabelle Z theory file** **153**

**G Max example Isabelle theory file** **155**

**H Exdev SAL file** **163**

# Chapter 1

# Introduction

## 1.1 Motivation

This work was prompted by the author's experiences with safety critical software projects in industry. Many good processes exist for ensuring the integrity of software. Some are verification techniques that seek to confirm that the software satisfies necessary requirements, others are development processes that are designed to minimise the creation of faults by ensuring that the programmers always have a clear idea of their objectives, and always have a clear understanding of the source code that they and other team members are writing. All of these techniques are valuable to software development, but many of the processes in use have limitations that prevent or complicate their application to software that interacts directly with hardware.

Many projects make use of static analysis to give a measure of assurance for the safe functioning of the code. Some of these analysis techniques measure a general software quality metric (such as conformance to a coding standard) to give an engineering estimate of its quality. This is often demonstrably very accurate and able to give a very high level assurance, but it lacks the mathematical certainty that is (at least philosophically) possible through the application of formal methods. Other techniques have a formal underpinning, particularly those that are based on "safe" language subsets. These use restricted versions of common programming languages to make the code completely statically determinable and so amenable to static analysis. The details of these languages are discussed further in §2.3 but the significant issue is that by restricting the language they necessarily make themselves unusable for applications that rely on the features that have been removed. One area that depends heavily on features that are often restricted or removed is hardware control. This will be the focus for the work presented here.

Hardware control and interaction is an area that is central to many safety-critical systems as they tend to be embedded systems or systems with a device control aspect. Many restricted languages remove any feature that interacts

with the hardware, since these features reduce portability and prevent deterministic reasoning about the code without making assumptions about the behaviour of the hardware. However, in the case of device drivers it makes perfect sense to make assumptions about the hardware — a specification of the hardware's behaviour is always necessary if software is to be written to control it.

## 1.2 Objective of this work

The principle difficulties that current techniques face when verifying hardware-dependent software are:

- They have no way to statically determine the behaviour of code that interacts directly with the hardware.
- Current techniques are necessarily detached from the hardware (for instance language subsets, discussed in §2.3) or treat hardware in an abstract, general way (such as the heap model in Separation Logic §2.5.3).
- Verification must fit into an industrial workflow and not be overly dependent on expert skills, and must be applicable to large scale systems in reasonable time.

This work aims to produce techniques that use knowledge of the behaviour of specific hardware in order to allow the verification of its control software. More detailed background on verification in general, and the use of restricted language subsets is presented in §2. This work attempts to avoid the difficulties of analysing high level language code by taking the opposite approach — analysing code at the executable file level. While restricted languages attempt to make code sufficiently abstract that hardware details are irrelevant, the objective here is to make use of known hardware details to make high level language concerns irrelevant. The intention is to disassemble executables using a formal specification of their target architecture as a guide. This should produce an interpretation of the software based on the environment in which it will run and should provide a better basis for understanding its interaction with the hardware.

The principle contribution of this work is a technique for inferring a formal model of the behaviour of a hardware dependent system that has the following properties:

- The ability to represent the interaction of software and hardware components in the same model, and allow the verification of properties of hardware usage;
- a fully automatic implementation with no human input required after the submission of the hardware specification and the software for analysis;
- produces models of a size and complexity that can be understood by humans and is practical for the application of formal verification techniques;

- maintains traceability from the produced model back to the source program to support fault localisation and repair.

Along with solving the technical problem of analysing executable code using formal methods, this work will strive to produce techniques that can be automated wherever possible. Two common criticisms of working at the executable code level are that the sheer number of individual instructions make analysis difficult, and that the code is in a machine readable form rather than one intended for human understanding. By allowing automation of the labourious parts of the analysis it is possible to avoid both concerns, since computer systems are intended to handle large volumes of repetitive work, and no criticism can be made of using a machine to interpret machine readable data.

The potential for exploring the analysis of specific, complex assembly language and CPU detail was present at many stages of this work. Areas such as virtual memory addressing, multi-processor architectures, timing, and interrupts could each have been explored individually for the entire duration of the work. This exploration was eschewed in favour of developing a complete workflow that converts an executable to a formal representation with a focus on being applicable to industrial verification processes. This required certain assumptions to be made about the software and the execution environment. It is assumed that the target of this verification will be a high integrity embedded system in which the hardware and software design emphasises reliability over "rich" features, and all components of both the hardware and software system are under the control of the implementing organisation. This may not be direct design control, but it is assumed that the hardware and operating system components will be known, expected to be relatively static for the life of the system, and available for analysis.

Specifically, the analysis assumes that the virtual memory and execution environment of the program is correctly maintained. It assumes that the virtual address space is free of aliasing, and is predictable — so the value written to a location is the value that will be read subsequently unless it is explicitly altered by this program. Also, interrupts are ignored since it is assumed that the CPU will resume the program with the same environment as when it was interrupted. Where the interrupt mechanism alters hardware registers or something else that is observed by the program this should be represented in the hardware specification or in the verification properties.

These assumptions are not unreasonable, and can be considered in the framework of a layered process of system verification. The assumptions should be based on the verification of the CPU hardware, and of the operating system components that drive it — such as the interrupt handlers and the code that maintains the Memory Management Unit (MMU) control registers. Both of these are instances of highly hardware-dependent software they might well be the first elements of a system to be analysed and verified using the techniques presented here. Since it is usual for interrupt handlers to be executed without the possibility of being further interrupted it would be quite reasonable to use these assumptions when performing the verification of their behaviour. They

could be verified against the property that they return the CPU to the interrupted program without altering its state, and so analysis of the remaining software could proceed with this as a well-founded assumption. Similarly, the operation of the MMU could be conducted on a system specification that models the interactions between the real address space and the virtual address space. The correct maintenance of the properties of the virtual address space could be verified and used as justification for the assumptions used analysing higher layer software.

Another assumption is that the structure of the software is static for the active life of the system. Self-modifying code is an interesting and active research area[23], but preventing the application of such ideas is an acceptable loss if an extremely high-integrity system is required. Similar arguments can be made about other complex system structures.

## 1.3 Thesis Outline

This document consists of ten chapters and eight appendices. These form five parts:

- §2 gives an overview of the background to hardware-dependent software verification and a review of relevant prior work.
- §3,§4, and §5 describe the analysis process that was developed by this work, including some theoretical work developed to support the analysis, and detail of the phases of the analysis workflow that converts an input executable into a formal model.
- §6 introduces the Spurinna package that implements this analysis process, §7, §8, and §9 describe some case studies that analyse small code samples and verify defined properties.
- §10 contains conclusions and discusses potential future work.
- The appendices contain materials that may be useful for reference when considering the analysis process. The contain the full text of the example formal models and the SAL and Isabelle files used in the verification examples.

# Chapter 2

# Background

## 2.1 Introduction

This section presents a literature review of the areas that are relevant to this work. After a general outline of the verification problem, the literature is presented in three areas, corresponding to the three elements needed for a verification process. Each section contains a brief summary of the use made of the material in this thesis. The selection and application of particular methods is described in detail in §3.

## 2.2 Verification

The aim of any verification process is to ensure that a system exhibits certain properties. What these properties are is one of the inputs to the process, and the choice of suitable property definitions to ensure general reliability is a large and complex subject that is outside the scope of this work. Once the properties have been selected the verification of these properties in a software system requires three phases of analysis:

- **Code analysis** A technique to determine how the software behaves.
- **Modeling** A language is needed to represent the behaviour at a level of abstraction appropriate to analyse the properties of interest.
- **Property verification** A process to check that the behaviour satisfies the required properties, and identify violations with information that can be used to correct the problem.

Different verification techniques will have a greater or smaller explicit emphasis on some of these phases, but they will all exist in some fashion. Even in manual code reviews the behavioral model will be encoded in the reviewer's notes and mental model, and the property verification technique will be the reviewer's approach to fault finding.

The remainder of this chapter will give an overview of the work that already exists in each of these three areas, with the intention of identifying suitable building blocks for an analysis technique that can be applied to hardware dependent software.

## 2.3 Code analysis

### 2.3.1 Overview

The first phase of analysis — determining how the software behaves — is complicated by many factors. Sheer size and complexity of software can make it difficult to fully determine the behaviour of the system, but size and complexity can be countered with automation. A more critical difficulty in understanding the behaviour of a software system is the lack of clear semantic definitions for many of the most common high level languages.

C.A.R. Hoare wrote [35] that:

> Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of pure deductive reasoning.

Unfortunately, this is only true if the meaning of every statement in the text of the program has a well defined operation. Whilst most programmers operate on the assumption that they know what any given language element will do, their understanding often rests on a number of implicit assumptions. As an example, the C language has been in use for 40 years and multiple documents have existed as *de facto* and *de jure* standards. The book published by the language authors [41] was used as a standard in many areas; the American National Standards Institute released a standard in [11]; the International Standards Organization accepted the 1990 revision of ANSI C as a standard, and have since published an updated standard [38] that includes many features — such as inline functions — that were available in compilers for several years. Although ISO C99 is largely accepted as the standard there is a very large variation both in the standard that a compiler implements, and the extent to which a compiler correctly implements the standard.

This multiplicity of standards and compiler implementations produces a situation where the exact behaviour of a program written in C cannot be determined with certainty, even if the entire source code is available. Different compilers may produce different executable code, or they may produce semantically different code for different target platforms. In many cases this is due to the flexible and platform-independent nature of the C language. Many features are deliberately and explicitly standardised as "implementation dependent" where the compiler authors are expected to chose the behaviour that will accomplish the rough specification of the behaviour in the way that is most efficient on the target platform. Compiler verification is also an open research area that has

had some successes [21] but is still not widely adopted, so even if the standard was well understood there would be no certainty that the compiler produced a correct implementation.

The implementation dependent choices in the standards highlight the platform independence of the C language as its advantage for general programming, but a significant problem faced where high integrity verification is required. The language is designed to be written with the user unaware of the exact nature of the system on which the program will run. This can be handled elegantly by language constructs that allow the programmer to specify intent and have the compiler select the optimal implementation. However, this can go wrong if the programmer makes too many assumptions based on the system he is using for testing, rather than the target platform, or if the compiler writer has assumed a different interpretation of the programmer's intent and optimises for that instead.

These assumptions about intent and implementation can have particularly drastic repercussions when the limits of the system are exceeded in some way. Attempting to use more memory than exists, assuming a particular layout of the content of structures in memory that is not replicated on a different platform, expecting certain devices to be "memory mapped" at particular locations which actually vary across architectures, and many other errors can occur as the result of assumptions made at different levels.

### 2.3.2 MISRA C

In an effort to reduce the uncertainty resulting from the C standards' implementation dependent aspects, the Motor Industry Software Reliability Association (MISRA) released a "restricted subset" of the C language. The MISRA-C [46] guidelines remove most of the features whose behaviour is implementation dependent or incompletely defined in the major standards. The guidelines also include various coding style guidelines that seek to eliminate code patterns that have seemingly obvious but practically indeterminate meanings.

The incrementation operators are a particularly good example. The C statement:

```
 x = y++;
```

can be expected to assign to variable $x$ the current value of $y$ and then increment the value contained in $y$. However, the statement:

```
x = y++ / y++;
```

is legal C but will have a different behaviour depending on the compiler's evaluation choices. It could have the same semantics as:

```
x = y / (y+1);
y = y + 2;
```

or as:

```
x = y/y;
y = y + 2;
```

depending on exactly when the post-increment operator is evaluated. The MISRA-C guidelines ban the use of the increment operator inside compound statements, much to the chagrin of some of C programmers. Many other similar patterns are banned from MISRA-C compliant code.

The MISRA-C rules are usually implemented in the form of static analysis tools that check that code conforms to a coding standard. Tools such as PC-Lint[8] and LDRA testbed[7] can check conformity to the MISRA-C ruleset, as well as other coding standards, including user-supplied ones.

The dissatisfaction of some C programmers — notably those who aim for maximum semantic density in their code — highlights a key area of resistance to the uptake of language restrictions as a partial solution to software defects. By limiting the language features available to the programmer it can make programming more time consuming for the same semantics. However, this extra time is a result of requiring more explicit reasoning about the operations being performed. In some cases simply forcing programmers to think and explicitly write the operations they wish to perform can result in the identification of logical errors.

### 2.3.3 SPARK Ada

These problems are not exclusive to C. SPARK Ada begins with a similar approach to MISRA-C. The SPARK Ada language is a restricted subset of the Ada language that removes features that allow for indeterminate behaviour. SPARK Ada goes beyond simply eliminating things that are inadequately specified and also seeks to prevent errors that are caused by runtime dynamism. By requiring all collections to be finite and statically defined it is possible for static analysis tools to determine the total resource requirements of the program. This sort of information is vital for embedded systems with limited system resources.

The SPARK restrictions are similar to the MISRA-C rules in that they increase the burden on programmers, but in doing so force the programmers to think explicitly about their systems. The introduction to the SPARK Ada book [17] describes the approach:

> Those familiar with the evolution of Ada 83 into Ada 95 will note that many of the facilities added in Ada 95 are not available in SPARK. This is almost inevitable because most of the new facilities in Ada 95 were added in order to increase dynamic flexibility – that is to give more flexibility at run time. But this is precisely what SPARK is not about; in order to prove that a program is correct, it is necessary that the dynamic flexibility be kept to a minimum.

Having restricted the language to allow meaningful static analysis, the SPARK system includes a suite of static analysis tools maintained by Altran-Praxis Ltd. [9]. To facilitate analysis SPARK Ada code is annotated with Ada comments

that have a defined format that is read by the analysis tools. These annotations contain assertions about the attached Ada code that should be statically determined. The statements include explicit lists of imports and exports, and statements of derivations that show how the author perceives the posterior values of variables exported to be derived from anterior values of imports. The static analysis tools can check that only explicitly imported variables are used and only explicitly exported variables are written, so identifying any unintended side-effects that are possible in pure Ada. The dependency analysis checks the derivation statements to ensure that the values of exported variables cannot be influenced by unexpected sources.

In addition to this data flow analysis, the SPARK toolset can perform control flow analysis. Annotations describing pre- and postconditions of the code segments can be checked by analysis of the control flow graph of the program.

### 2.3.4 Hardware verification

The phrase "hardware verification" is used here to refer to the verification of properties of digital electronic systems, particularly those that compose to form the computer systems on which the software under analysis will run.

Verification has been applied to many classes of digital system, but CPUs are the most relevant to this work. Two examples of formally verified CPUs are VIPER and AMULET. The VIPER processor was developed in the 1980s with the intention of being an entirely formally verified processor to be used in high integrity systems. The design was formalised in Z [40] and later in HOL [49]. The development was performed at the Royal Signals and Radar Establishment, in Malvern and the processor was used in a number of Ministry of Defence applications. The AMULET project developed a verified ARM-based processor in 1993 [33]. The latest AMULET is AMULET3 [34]. The AMULET research group at the University of Manchester continues to develop formally verified hardware projects [53, 57].

In both of these cases, and others [31, 32, 19] the digital system is modelled and verified in a formal modeling language such as CSP, Pi Calculus, Z, or HOL. Understanding the working of the system on which software runs is important to understanding the operation of the software itself, but the practical scope of work possible requires this thesis to take a layered approach. Since the verification of hardware implementations of formal models is a mature subject this work will use a formal model of the computer system and its architecture as an input and assume that the verification of the real implementation is performed independently.

### 2.3.5 Summary

Although MISRA-C, SPARK Ada, and related approaches make it possible to write high level programs that can still be properly analysed, they do so by making concrete the separation between the software and the system on which it runs. This approach is ideal for the vast majority of applications, where

portability with semantic certainty is the key requirement. However, such an approach makes the direct hardware control applications that are the focus of this work impossible. Consequently, it will be necessary to avoid high level languages and all of the uncertainties that come with compilation. The desire to be independent of specific hardware is not present in the development of hardware control software, so the abstraction of implementation details to simplify understanding and portability across platforms is not relevant. Given that this objective is reversed, it is logical to reverse the approach taken by the language abstractions. Instead of trying to remove any requirement for knowledge of the target platform, this work will require a specification of the platform. This specification will be used to gain an understanding of the behaviour of the software on that platform.

Hardware verification is beyond the scope of this research, and is a well-understood field, so it will be assumed that an abstract representation of the system hardware is adequate. The nature of the representation will be discussed in §3 but it will operate at the level usually found in the instruction set reference documentation of the target processor. System features such as registers and memory locations will be modeled explicitly, but electrical and digital logic details will be abstracted.

## 2.4 Modeling

### 2.4.1 Overview

Once the behaviour of the software has been inferred its semantics must be documented in a notation that will best support the verification or refutation of the properties of interest. There are many different notations and languages for representing particular aspects of software behaviour, whilst deliberately abstracting others. This abstraction and simplification is necessary to allow the analysis to be computable in reasonable time, but also to allow the conclusions of the analysis to be comprehensible.

The selection of a formal modeling language largely depends on the nature of the properties that will be analysed. All formal languages emphasise some type of problem and their syntax is designed to write such problems clearly, whilst deliberately hiding other details behind abstraction. The correct choice of emphasis and abstraction is vital if the final model is to be readable, either by humans or by analysis tools. The criteria for readability are slightly different depending on whether human comprehensibility or machine processing are the objectives, but in both cases there is a need for succinctness: presenting the semantically critical elements in enough detail to allow meaningful understanding, but abstracting other elements behind the symbology so that the model elements are small enough to be handled easily.

Available tool support is another important criteria. The techniques for proving or falsifying properties are discussed in more detail in §2.5 but the availability of automated support for analysis will influence the choice of lan-

guage.

### 2.4.2 The Z notation

The Z notation is described by Woodcock and Davies [61] as:

> [...] based upon set theory and mathematical logic. The set theory used includes standard set operators, set comprehensions, Cartesian products, and power sets. The mathematical logic is a first-order predicate calculus. Together, they make up a mathematical language that is easy to learn and to apply. [...]

Z was originally proposed by Abrial et al. [14] and developed at the Programming Research Group at Oxford University. The ISO standard for Z was completed in 2002 [39].

Z allows simple axiomatic definitions of entities that are irrelevant to the specification being written, such as:

$[Man]$
$[Woman]$

It then allows simple definitions in BNF and set theory:

$Person ::= Man \mid Woman$
$Class == \mathbb{P}\ Person$

The mathematical parts of Z are mainly contained in *schemas*. A schema contains two parts: variable declarations and predicates. *State schemas* define pieces of the system state, usually — but not necessarily — the state of discrete components.

*AdvancedAlgebra*

$students : Class$

---

$\#students \leq 100$

This defines that the *AdvancedAlgebra* course contains *students*, which is of type *Class* — so is a set of *Person* items. Below the horizontal line is a predicate that limits the class size to 100 students. Schemas can be written on one line in a simpler form, so *AdvancedAlgebra* could be written as:

$[students : Class \mid \#students \leq 100]$

Another state schema might define the marks the students have obtained using a function:

*AlgebraMarks*

$students : Class$

$$
\begin{array}{|l}
marks : Person \pfun \mathbb{N} \\
\hline
\forall s : Person \mid s \in students \bullet \\
\quad s \in \operatorname{dom} marks \\
\hline
\end{array}
$$

The predicate here requires that there is a mark for every student. We can now perform *schema conjunction* on these two:

$$MarkedAlgebra == AdvancedAlgebra \land AlgebraMarks$$

This produces a new schema — here called *MarkedAlgebra* — that uses all the variables from both the conjunct schemas, and has a predicate that is the logical conjunction of the two predicates. Where variables in different schemas have the same name and type they are assumed to refer to the same thing. Other logical connectives, such as disjunction, are also possible. The conjoined schema in this case is:

$$
\begin{array}{|l}
\multicolumn{1}{l}{MarkedAlgebra} \\
\hline
students : Class \\
marks : Person \pfun \mathbb{N} \\
\hline
\#students \leq 100 \\
\operatorname{dom} marks = students \\
\hline
\end{array}
$$

The other type of schema available in Z are *Operation schemas*. These specify how operations in the system change the state. They can define variables but more usually import state schemas for the parts of the system they alter or refer to. They are written in a pre- and post-condition form, with undecorated variable names referring to the state before the operation, and decorated variable names (e.g. $x'$) referring to the state after the operation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{AddMark} \\
\hline
\Delta\ MarkedAlgebra \\
who? : Person \\
mark? : \mathbb{N} \\
\hline
who? \in students \\
students' = students \\
marks' = marks \oplus \{(who? \mapsto mark?)\} \\
\hline
\end{array}
$$

A number of features are illustrated here. Question marks on variables are a convention to indicate inputs. Exclamation marks are the convention for outputs. The Greek symbol $\Delta$ is shorthand for bringing into scope all the contents of the state schema, as both before and after states — so every variable in *MarkedAlgebra* is brought into scope in both undecorated and decorated form. The predicate declaring that $students' = students$ is important in Z, since unspecified post-conditions are not automatically considered unchanged

— they are considered unspecified, so an implementation is free to set them to any value. The symbol $\Xi$ is shorthand for not only bringing into scope both decorated and undecorated forms of all variables in a state schema, but also including the predicate that their values are unchanged.

Finally, relations of all types are modeled as sets of pairs, where the first element is from the domain of the relation and the second is from the range. The $\oplus$ operator is *overriding*, so the *marks* function is left unchanged except where the domain matches the domain of the second argument. Here the second argument is an anonymous function (i.e. set of pairs) with only one element: $(who?, mark?)$, so only the value of $marks(who?)$ is altered.

Operation schemas are usually instantiated by using renaming in the standard $[x/y]$ form, where $x$ is used in place of all occurrences of $y$. Operation schema can be composed in a number of ways, including sequential composition ($\fatsemi$), conjunction ($\wedge$), and disjunction ($\vee$).

$$AddMark[Bob, 75/who?, mark?] \fatsemi AddMark[Jane, 92/who?, mark?]$$

This specifies performing the *AddMark* operation to give Bob 75 points, followed by performing the same operation to give Jane 92 points. The conjunction of the two operations would not be significantly different in this case as it would resolve to adding both marks simultaneously, but the disjunction would result in *either* Bob's *or* Jane's mark being entered. This is particularly powerful where the predicates are mutually exclusive, so the decision can be made by the conditions. For example $[x, x' : \mathbb{N} \mid x < 30 \wedge x' = x + 1] \vee [x, x' : \mathbb{N} \mid x \geq 30 \wedge x' = 30]$ will increment $x$ until it reaches 30, after which it will leave it unchanged.

### 2.4.3 B

The B formal method [12] is a state based specification language, with many similarities to Z (see §2.4.2). A critical difference to Z is that B is based around an explicit abstract machine model. The emphasis in the design of B was for a direct relationship to the implementation. This implementation based machine model simplifies the creation of tools for simulation and code generation.

Several extensions to B appeared around the time that this work was beginning. CSP||B [58] composes B and CSP in a similar way to CSP-OZ (see §2.4.6). Event-B [13] is an evolution of B to have a simpler syntax and a more heavily process driven style. Event-B has strong tool support from the Rodin suite [15].

### 2.4.4 CSP

Communicating Sequential Processes (CSP)[36] is a process calculus — that is, a language for specifying the behaviour of a system as a collection of processes that evolve separately but interact. In CSP the interaction is modelled through communication along defined *channels*. The evolution of a process can be guided

by its communication with other processes. More critically, the evolution of two processes can be *synchronised* by their communication. This synchronisation is the key element of CSP. It allows a simple and elegant abstraction to reason about many of the most fault-causing problems of distributed systems.

The CSP model contains *processes* and *events*. The most basic process evolution is *prefix*, where an event occurs and then the process evolves to a different process:

$$x \rightarrow P$$

This statement defines a process, so can be the target of a prefix evolution, which can itself be the target of an evolution, etc.:

$$a \rightarrow (b \rightarrow (c \rightarrow P))$$

The $STOP$ process represents deadlock, and is used as the terminal process of all finite-trace models:

$$a \rightarrow (b \rightarrow (c \rightarrow STOP))$$

Processes can also be recursive:

$$P = x \rightarrow (y \rightarrow P)$$

The evolution of a process can also contain choices. These choices are defined as either *internal* ($\sqcap$) or *external* ($\Box$) choices. In the case of internal choice the decision is made entirely by the system and the environment cannot influence the choice. With external choice the decision is usually made by the environment precluding one path and allowing the other.

$$P = (x \rightarrow A) \sqcap (y \rightarrow B)$$
$$Q = (x \rightarrow A) \Box (y \rightarrow B)$$

In this example $P$ and $Q$ have different observable behaviours since $Q$ will "allow" either event $x$ or $y$ to synchronise with it, whereas $P$ will only allow one — and which one will be determined by the system at run time.

The notion of communication and synchronisation is central to CSP models. Where two processes are executed concurrently (which is defined using the $||$ symbol) and their alphabets have an intersection they are assumed to require synchronisation on the shared events.

$$\begin{aligned} P &= a \rightarrow (b \rightarrow P) \\ Q &= b \rightarrow Q \\ System &= P \;||\; Q \end{aligned}$$

In the process *System* the sub-process $Q$ will not be able to evolve through the $b$ event until the sub-process $P$ is also ready to evolve with a $b$ event. Since $P$ requires an $a$ event first, this forces the whole system to wait for an $a$ event before either component proceeds past the $b$ event. This simple syntax for representing synchronisation and inter-process control is the core of most CSP models.

The events are then enhanced to represent communications. Processes can input and output on *channels* using the ? and ! operators. The pipe operator >> directs the output from one process to the inputs of another:

$$\begin{aligned} P &= a!v \rightarrow P \\ Q &= a?v \rightarrow Q \\ System &= P >> Q \end{aligned}$$

This will pass an endless stream of values $v$ from $P$ to $Q$ along the $a$ channel.

Verifications on CSP models usually operate on traces in the systems. Verification techniques such as refinement measure possible traces, and refused events at different points in the traces. A process $Q$ is a *traces refinement* of $P$ if all the traces of $Q$ are also traces of $P$. If $P$ specifies the "safe traces" of a system then an implementation $Q$ performing a subset of them seems reasonable. This is incomplete as a requirement system though, so *failures refinement* requires that the *failures* of $Q$ is a subset of the failures of $P$. Failures are defined as a pair $(s, X)$ where $s$ is a trace that ends with an event $X$ being refused. This models the behaviour that an implementation is allowed to refuse, and prevents the $STOP$ process satisfying a requirement, even though its traces are a subset of any processes traces so it is always a traces refinement. Finally, *failures divergence refinement* adds requirements to identify and limit *livelock*, where a system may perform internal events but not produce any externally visible actions.

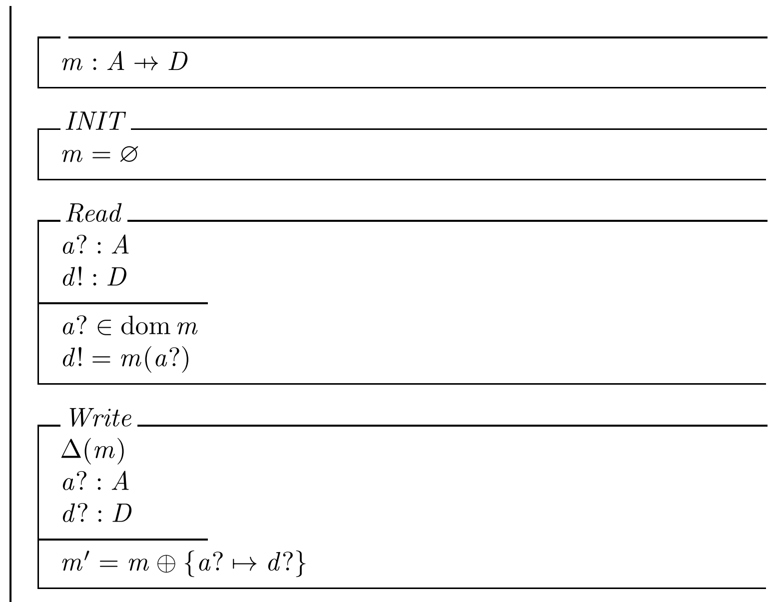
Failure Divergence Refinement (FDR) [44] is a software system designed to analyse CSP models. As its name implies it emphasises *failures divergence refinement*, as discussed in §2.4.4, and can also check the more general traces refinement and failures refinement properties. It is also capable of measuring other system properties, such as identifying potential for deadlock and livelock.

### 2.4.5 Object Z

Object Z[24] is an extension to the Z language (see §2.4.2) to add object oriented ideas such as classes, inheritance, and polymorphism. Object Z creates classes by collecting a Z state schema with a Z operation defining initialisation and some Z operation schemas to define the methods. Visibility notions similar to *public* and *private* in other OO notions are handled with a "visibility list" that identifies public and, implicitly, private methods. An example from [27]:

$Memory[D]$

$\upharpoonright (INIT, Read, Write)$

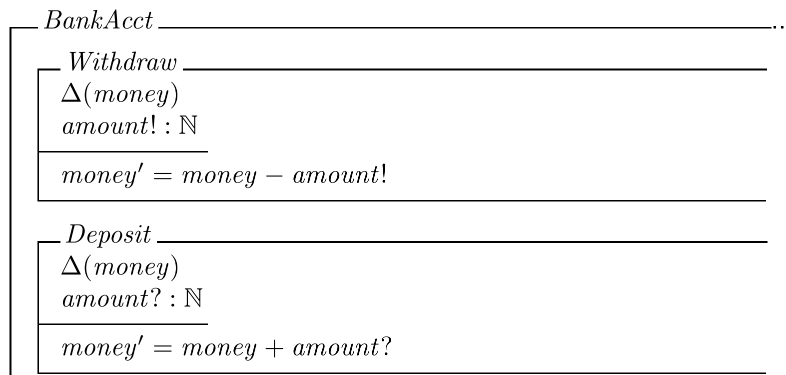



Inheritance is handled by including all the state and transition information but not the visibility list. Multiple inheritance is allowed and is modeled by simply including all of the content from the parents in any order. Schema with the same name are conjoined.

Instantiation of classes simply uses class names as variable types and then uses the dot notation for methods:

$ct : Count$
[...]
$ct.INIT$
[...]

Object Z allows communication between classes using a CSP style || notation, so two methods can be parallel composed and then inputs and outputs with identical base names are matched.



*Transfer*

$from? : BankAcct$
$to? : BankAcct$
$amount? : \mathbb{N}$

$from?.Withdraw(amount?) \mathbin{||} to?.Deposit(amount?)$

### 2.4.6 CSP-OZ

CSP-OZ[30] takes the Object Z extension of Z (see §2.4.5) and extends it further by using the CSP process calculus (see §2.4.4) to add process-based control flow specification for the operations. A CSP-OZ class has a collection of CSP process in which the event alphabet is the Z operation schema names from the body of the class. The `main` process defines the entry point and the evolution of this CSP process specifies the allowed traces of operations that can occur on the class.

*Printer*

$channel\ print : [p : PID]$
$channel\ card, nocard : []$
$main : print?x \rightarrow (card \rightarrow main \sqcap nocard \rightarrow main)$

### 2.4.7 Summary

The analysis process presented later in this work separates the inferred program behaviour into control flow and state change components. Consequently, it is ideal to represent the inferred behaviour in one of the languages that combines a process calculus with a state based language. Within reason, the choice of modeling language between Z and B is arbitrary. The B language has arguably greater tool support and its machine model makes many forms of symbolic execution and model checking easy to implement. However, this machine model is a core component of the language and this constrains all models to use at least this basic structure, even if an entirely different system model might make analysis more efficient or comprehensible. Z has no such constraints, being completely arbitrary system specifications and state change operation specifications. For this reason, and because of the author's prior experience and the Z community available in Sheffield, this work will use the CSP-OZ language for modeling. CSP-OZ allows the control flow components of the analysis to be rendered almost independently of the state change components, which simplifies the analysis and its implementation.

## 2.5 Property verification

### 2.5.1 Overview

Having inferred the behaviour of the system and modeled it in a formal language it is possible to assess whether the required properties hold, and if not it is desirable to produce counter-examples or other information that highlights the ways in which the system violates the properties.

There are a number of ways to establish whether a system's behaviour satisfies a property. The most general method is simply to test the system in a series of situations that have been designed to evaluate the property. Designing a suite of tests that is certain to identify all possible violations of the property is an extremely difficult task. Where the system has been formally modeled, and where the property is specified in a rigorous way it becomes possible to be more precise and complete with the test suite. The ultimate pinnacle of this approach is *model checking*. Where a system can be represented by a state model and a series of transitions it is possible to examine *every possible* trace of transitions from *every possible* state. If this can be accomplished, and if the model is an accurate representation of the system behaviour, then it is certain that the system satisfies the property, or that all failure situations have been identified. For systems of anything above trivial size this approach quickly becomes intractable. Since every possible state must be examined, along with every possible trace, a small increase in the potential state space creates an exponential increase in the search space — a problem known as *state space explosion*. For this reason most model checking is done with a complete transition system by a significantly simplified and abstracted state model. If the state model is chosen well then this does not need to impact the validity of the analysis. It is generally the case that system failures occur at critical values — division by 2 tends to imply that division by 3 will work, whereas division by 0 must be handled more carefully.

An alternative to model checking is a symbolic approach. If division by zero is to be avoided, for example, but all division operations occur within guarded code sections, then it is sufficient to demonstrate that the guards are such that it would be impossible to enter the code section with the divisor set to zero. This approach can produce powerful results since it is not computationally dependent on the size of the state space or of the model. The key drawback of symbolic proof is that anything beyond the more basic proofs requires a level of creative thought that has not been automatable. This forces the proofs to be performed manually, although various tools are available to simplify the proof process, provide feedback and verification of the proofs, and to tackle the simple proofs automatically once the proof condition has been imaginatively rephrased to be handled by machine.

As with formal modeling languages, different proof and analysis techniques are available for different classes of problems. They each have restrictions and abstractions that must be weighed when selecting a technique for a particular set of properties. The choice of CSP-OZ for the formal modeling language was partly influenced by the possibility of easily translating the produced models to

emphasise the components necessary for particular analysis techniques.

### 2.5.2 Hoare logic

What is now known as *Hoare Logic* or *Floyd-Hoare Logic* was presented in the paper titled "An Axiomatic Basis for Computer Programming"[35]. The intention of the work was to present a method for performing deductive reasoning about the properties of a program. To do this it was necessary to introduce the axioms that describe the behaviour of the basic components of computer programs.

The paper begins by describing the rules for computer arithmetic — especially highlighting the differences from conventional mathematics' version of arithmetic, such as value overflow. It then proceeds to present axioms for the operations in a pseudocode language that contains elements common to all imperative programming languages. The axioms are presented in the form of *Hoare triples*, which contain a *precondition*, the pseudocode statement, and a *postcondition*.

$$P\{Q\}R$$

Here $P$ is the precondition predicate, that specifies the conditions that must hold if the defined operation is to occur. All the axioms of Hoare Logic have this form, which is known as an *Hoare triple*; division is only properly defined if the divisor is non-zero, for example. Some axioms are universally applicable and so have the precondition **true**. $Q$ is the program statement in pseudocode. $R$ is the postcondition, which is a predicate defining what can be relied upon to be true after executing this statement, assuming that the precondition is satisfied.

The first axiom is for assignment:

$$\vdash P_0\{x := f\}P$$

where $x$ is a variable identifier;
$f$ is an expression;
$P_0$ is obtained from $P$ by substituting $f$ for all occurrences of $x$.

This pattern of defining preconditions as transformations of the postcondition is repeated for the other axioms and reflects the predicate transformation semantics that underlay the system. It allows useful reasoning about the behaviour of the whole program by eventually representing the operation of the program as a transformation of the expected postcondition into the precondition. Verifying properties then requires proving that the properties are guaranteed by the postcondition predicate, and then assessing the limitations implied by the precondition. Errors in the program are usually manifest as unexpected preconditions ($x > 4$ or something equally strange) that highlight unanticipated dependence on properties of the inputs.

The next axioms are the *Rules of Consequence*:

$$\textit{If} \vdash P\{Q\}R \textit{ and } \vdash R \supset S \textit{ then } \vdash P\{Q\}S$$
$$\textit{If} \vdash P\{Q\}R \textit{ and } \vdash S \supset P \textit{ then } \vdash S\{Q\}R$$

and the *Rule of Composition*:

$$\textit{If } \vdash P\{Q_1\}R_1 \textit{ and } \vdash R_1\{Q_2\}R \textit{ then } \vdash P\{Q_1;\ Q_2\}R$$

These rules allow for a program to be built from a sequential composition of operations, as is the case in almost all imperative programming languages, and allow for the satisfaction of following preconditions to be determined by components of preceding postconditions.

The final rule covers the other major class of imperative programming language construct: *iteration*. The *Rule of Iteration* is presented as:

$$\textit{If } \vdash P \ \wedge B\{S\}P \textit{ then } \vdash P\{\textbf{while } B \textbf{ do } S\}\neg B \wedge P$$

This defines some guard condition $B$ that determines the limit of the iteration, and some *loop invariant* $P$ that is always true during the execution of the loop. Generally, the loop invariant is phrased so that it guarantees some property of the elements that have been iterated over, or the value of some variable derived from the iteration steps.

Although not mentioned in this paper, the axioms of Hoare logic also include a *conditional rule*:

$$\textit{If}$$
$$\vdash P \wedge B\{S\}P \textit{ and } \vdash P \wedge \neg B\{T\}P$$
$$\textit{then}$$
$$\vdash P\{\textbf{if } B \textbf{ then } S \textbf{ else } Y \textbf{ endif}\}Q$$

Subsequent work would eventually reverse the bracket notation to enclose the predicates at either side of the statement: $\{P\}Q\{R\}$ and to add a loop variant to the iteration rule that must decrease to guarantee termination.

Once the axioms have been established it is possible to prove properties of programs by chaining the rules together. The post condition of a preceding element can be used to show that the precondition of a succeeding element is satisfied, and so certify its postcondition to be used to satisfy a following element's precondition, and so on. Although Hoare logic axioms are defined over pseudocode elements, it is generally possible to rephrase any imperative language program into these basic components.

Since its publication, Hoare logic has been applied to a wide range of verification tasks, on languages as disparate as Java[59] and Z [29]. The application of pre- and postconditions and invariants that was pioneered by this work has formed the foundation of verification techniques such as SPARK Ada (discussed in §2.3.3) and Spec#[18]. It has been applied to compiler verification [21] and has been extended to accommodate extra analysis features, such as those presented in Separation Logic discussed in §2.5.3.

### 2.5.3 Separation logic

Separation logic is an extension to Hoare logic [35] that supports reasoning about programs that alter data structures. Reynolds [51] first suggested extending Hoare logic to include an operator to conjoin two propositions that depend on distinct areas of storage. Ishtiaq and O'Hearn [37] then used the Logic of Bunched Implications [47] to extend Reynolds ideas and first began using the $*$ and $-\!\!*$ operators. The ideas were then clarified in [48], which saw many of the definitions and axioms of Separation logic defined. A more detailed semantic analysis followed in [62].

#### Intuitionistic Reasoning about Shared Mutable Data Structure [51]

This paper extends Hoare logic "to deal with programs that perform destructive updates to data structures containing more than one pointer to the same location". The key concept is what the paper calls "independent conjunction". It uses the & operator — which is superseded by $*$ in later Separation Logic papers — to produce syntax such that $P \& Q$ holds only when $P$ and $Q$ hold *and* depend on disjoint areas of storage.

The model defined here has a `value` as *either* an integer, an `Atom`, or a `Location`, with these sets disjoint and countably infinite. The possible stores and heaps are defined as:

$$
\begin{array}{ll}
\mathit{Stores}_V = (V \rightarrow \mathit{Values}) & \text{Where V is a finite set of variables} \\
\mathit{Heaps}_L = (L \rightarrow \mathit{Values}^+) & \text{Where L is a finite set of } \texttt{Locations}
\end{array}
$$

The paper uses a style similar to LISP [45] to represent constructing new variables. It uses $\mathbf{cons}_1$, $\mathbf{cons}_2$, $\mathbf{cons}_3$, etc. to construct sequences of items on the heap. So $x := \mathbf{cons}_3(1, 2, 3)$ will create a new entry in the `heap` function that maps some unique location to the sequence of values $1, 2, 3$ and then updates the store so that $x$ is mapped to this location. Figure 2.1 shows this relationship diagrammatically.

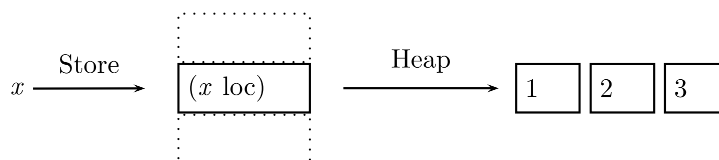



Figure 2.1: An example of the **cons** operation

This differs slightly from the normal programming model of referencing since the sequence $1, 2, 3$ is not, itself, in the same store as the location. In C - and in actual implementations of all languages - reference locations are data values in the program's memory space, just like the values to which they refer. Also, the collections of values that are referred to by the `Heap` function are not subsets of a

contiguous store but are independent sets. Consequently, in this model, there is no way to represent offsets to `Locations`. If $x \mapsto a, b, c$ were in the `Heap` it is not implied in this model that $x + 1 \mapsto b, c$ since $x + 1$ is an entirely separate entry in the domain of the `Heap` function. Figure 2.2 shows this diagrammatically.

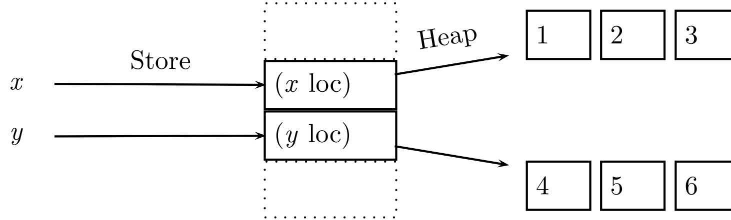



Figure 2.2: The **cons** operation producing disjoint sets

Altering the model to better capture realistic behaviour of offsets was one of the significant additions by O'Hearn et al. [48].

The paper introduces the use of a comma separated list into its system of propositions to represent the condition that holds after the use of **cons** described above. So it is now possible to produce some Hoare logic assertions:

$$\{P\} \quad x := \mathbf{cons}_n(E_1, ..., E_n) \quad \{P\&(x \rightarrow E_1, ..., E_n)\}$$

Later separation logic papers update this both the `Heap` model and the proposition to better represent the notion of sequential locations used in real systems.

This, with a few other straightforward definitions, allows the style of Hoare logic definitions and statements that will be used throughout Separation Logic. The paper ends with a partial correctness specification — that is, the series of Hoare logic assertions between the statements of the program — for a program that deletes zero value elements from a doubly linked list.

### 2.5.4 IsabelleHOL

Higher Order Logic is a branch of mathematical logic, but in this context it refers to the proof assistance systems developed at the Automated Reasoning Group in Cambridge. The original HOL systems were designed for verifying hardware, as discussed in §2.3.4. Isabelle is a related proof assistant developed by the same group for more general logical reasoning [1].

Proof assistants work by allowing the user to present proofs in a particular logical system and the assistant checking that the proof is a correct application of the logic's rules. HOL and Isabelle support a large range of logic systems, both higher order and first order. Systems produced for Isabelle include Hoare Logic, Zermelo-Fraenkel set theory [16], Z [43], and B [25].

A *theory* in Isabelle is a collection of datatypes, functions, and definitions, along with proofs of theorems over these. A theory begins by importing a

suitable logic system. This imports all of the basic rules of the selected system, which can then be used in the definitions.

```
theory Test
imports Datatype
begin

datatype 'a list = Nil ("[]")
  | Cons 'a "'a list" (infixr "" 65)

primrec app ::
    "'a list => 'a list
    => 'a list" (infixr "@" 65)
where
"[] @ ys = ys" |
"(x # xs) @ y = x # (xs @ y)"

primrec rev :: "'a list => 'a list"
where
"rev [] = []" |
"rev (x # xs) = (rev xs) @ (x # [])"
```

Having created definitions it is possible to write *theorems* and *lemmas* that encode properties of the definitions. These can then be proven by referencing the definitions and rules from the imported logic. Isabelle contains a number of automated tactics that will attempt to apply certain common patterns of rules to achieve a proof.

```
lemma app_Nil [simp]: "xs @ [] = xs"
apply(induct_tac xs)
apply(auto)
done
```

Additionally, Isabelle contains a package called *sledgehammer* [52] that will automatically convert the theorem and the definitions into the correct input formats for a range of external automated theorem provers and SMT solvers, which are then run to attempt to find proofs. There are also the *nitpick* [20] and *refute* [60] counter example generators that attempt to find descriptive counterexamples to the theorem if it is false.

IsabelleHOL has been used successfully in the verification of various systems [42], and has been incorporated as the backend to a software analysis system [22].

### 2.5.5 SAL

The Symbolic Analysis Laboratory (SAL [4]) is a suite of tools for performing analysis on state machine models. The tool suite includes a simulator, a

deadlock checker, a symbolic model checker, a bounded model checker, and a counterexample finder. The SAL input language [26] is common to all of the tools and there is a large community creating new tools that all use the common input language.

The SAL input language requires all types to be finite, but then supports finite sets, triples, subranges, arrays, records, total functions, modules, and recursive definitions. The state machines are defined with a collection of state variables and an initialisation, and then a collection of transitions. The syntax for the operations includes conditions on the state that control when the transition can be traced. The effect of the operation on the system state is defined using anterior and posterior variables, with the posterior variables identified with a prime symbol ($'$) in a similar way to the Z notation (see §2.4.2).

The simulator allows a machine to be loaded and transition paths to be explored. The deadlock checker analyses the transition conditions to detect potential states from which no transition will be possible. The SAL language allows for an *ELSE* transition that is always available, which can deliberately remove the possibility of deadlock, but for meaningful deadlock analysis this should be omitted.

The model checkers can check properties written in Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) forms. The symbolic model checker compiles the definitions into optimised binary decision diagrams (BDDs) and simulates models using Buchi automata. The bounded model checker uses the SMT solver *yices*[10], which was also developed at SRI.

### 2.5.6 Z2SAL

The Z2SAL[28] project has developed an automated system for converting Z specifications (see §2.4.2) into the input language of the SAL suite (see §2.5.5). The parser and translation system currently only works for pure Z, and not derivatives such as Object-Z or CSP-OZ. However, within this limitation it is possible to automatically translate entire Z specifications into SAL state machine models.

The process assumes that a Z state schema defines the system state, and a Z initialisation schema provides the details of the state initialisation. Thereafter, all Z operation schema are converted to transitions in the SAL model. The output SAL model is then available immediately for modelchecking against suitable LTL or CTL property specifications. The translation has the distinct advantage of retaining the names of state variables and operations as variable names and transition names respectively in the SAL model. This, coupled with the innate similarity between the anterior/posterior variable conventions between Z and SAL means that users familiar with the Z model can easily read the SAL model. This is of critical importance when errors are detected as they can quickly be traced back to the original Z specification. This makes the use of Z2SAL with the SAL model checkers a very easy and powerful technique for exploring temporal logic properties of a specification.

### 2.5.7 Summary

All of the techniques surveyed here could be applied to executable files that control hardware. In all cases it is possible to translate either CSP-OZ or one of its sub components into the form necessary for the analysis. Different analysis goals will necessitate different approaches, and this work deliberately takes no position on which technique should be applied. If any technique were favoured it would be easy to question the need to render the model to CSP-OZ and then convert, instead of rendering the model directly into the input language of the analysis technique, but this generalism was precisely the objective in the selection of CSP-OZ and in the efforts made in §3.3 to ensure that both the control flow and state transition features are rendered as fully as possible.

Time limitation prevented extensive case studies of verification, but the availability of the Z2SAL tool [28] made this an ideal example, the results of which are presented in §7.

# Chapter 3

# The proposed solution

## 3.1 Introduction

As described in §1, the aim of this thesis is to develop a technique to verify properties of high integrity software systems that interact directly with hardware. This chapter presents the design decisions that were made to produce an analysis process to achieve this goal. The decisions in choice of code analysis technique, modeling language, and verification approach were made with particular emphasis on the three principle objectives of this work:

- Analyse software that depends on hardware behaviour
- Handle large instruction count or large quantities of fine detail
- Produce results that are comprehensible and provide traceability between the results and the original code.

As identified in §1.2, the target for this work is high integrity embedded systems where the engineering team can be expected to have more control over the low level system components than might be the case in a more general software system.

## 3.2 The analysis process

The hardware dependence will be countered with reference to supplied specifications of the hardware behaviour. Given that the target domain for this work is software elements that are designed to provide hardware control or interaction it is reasonable to make the interaction of the software with the hardware the core of the analysis and the produced behaviour model. For the reasons discussed in §1.2 the analysis will be performed on executable files.

Because executable files potentially contain millions of instructions even for relatively small software subsystems it is important that scalability and automation be highly prioritised in the design of the analysis methodology. This

is handled by presenting the specifications in machine-readable form and then creating an entirely automated analysis work-flow that requires no human interaction.

With the absence of human interaction in the analysis process it is important that the eventual results are not totally divorced from the input materials. Identifying property violations is only useful if it is combined with enough traceability information to allow the problem to be localised in the source code and corrected. Additionally, the presentation of the analysis results must be clear and reasonable enough to be understood by a developer of the source program, rather than exclusively by expert users of the analysis technique. These conditions seem competitive since the greatest hurdle for comprehensibility will be the sheer number of instructions, whilst traceability requires that exact information be kept about the source of a particular model element.

Most executable files — especially, but not exclusively, those compiled with debugging information — contain considerable *symbol* information with function names and other readable information derived from the high level language. This information will be retained in the formal model to allow easy and comprehensible linkage to the source. The programmatic structure will also be retained as the model will encapsulate program elements at the function level, which on most architectures is clearly defined and used by compilers to implement functions (or their equivalents) in the high level language. Finally the virtual address of the instructions that are modeled by a particular formal model component will be retained as a subscript to the name. After the simplification stage (described in detail in §5.4) these will be collected into address ranges but they will still identify small and programmaticly self-contained sections of the source executable.

The sequential parts of the executable may contain long chains of instructions that are sequentially composed. To improve readability and to simplify subsequent application of verification techniques each of these sequential chains can be compressed into single operation schema that represents the entire chain's effect in the system. There are likely to be many of these sequences so the efficiency of this compression is of considerable importance. The techniques presented in §4 can be used to identify operations whose behaviour will not be altered by simple, textual concatenation into a single operation schema. This drastically reduces the computational overhead required to compress these chains as no evaluation of the semantics of the Z is required.

## 3.3 Formal model

The system state is easily represented by a Z state schema. This may be impractically large but Z supports compositional definitions so it can be componentised to whatever degree suits its author. It is not necessary to specify the entire working of the system under analysis, only such details as are of interest for the properties being analysed. The analysis process makes no demands at all on the system specification other that it contains a state schema named *System*.

The behaviour of software instructions will be divided into two classes. As far as possible the model will separate the control flow of the program from its interaction with the system state. Those instructions that define the execution paths will be referred to as *branch instructions*, whilst those that alter the system state but not the control flow will be labeled *sequential instructions*. The sequential instructions are easily modeled by Z operation schema. The control flow is more elegantly modeled in CSP. Those branch instructions whose behaviour depends on or alters the system state will have components in both languages.

The CSP-OZ notation allows for the combination of both languages into an organised, object-oriented framework. Other languages such as Circus or B allow for similar combinations of state and control flow but have more tightly entwined semantics, as well as a larger body of syntactically implicit components. For simplicity of automated processing — especially with a view to parallel analysis of large-volume, simple components — it was decided that CSP-OZ offered the best notation.

Although the system state can be modeled as a Z state schema directly, the instructions in the program all need individual operation schemas. The format of CPU instruction sets is as a collection of instructions with identifying mnemonics and parameters. It is reasonable to formally specify the instruction set in the same pattern. §3.3.2 and §3.3.3 introduce notations for *template specifications*. These have identifiable names and parameter signatures that can be matched to each instruction in the program and instantiated by syntactic replacement of identifiers in the template to produce a formal representation of the instructions' effect in the system.

### 3.3.1 System state specification

All the specifications in subsequent sections will describe operations that can be performed on the system state. Whilst the system state itself is not used directly as an input to the analysis process, its structure will influence the specification of the sequential and branch instructions. This section briefly discusses some design considerations for system state specifications and presents some simple examples that will be used for the remainder of this document.

The construction of the system state model will depend on the complexity and structure necessary to adequately represent each of the instructions in the instruction set. This work is deliberately ignorant of the exact system state model chosen. Different verification goals will require more or less detail specifications of certain aspects of a given platform. For any choice of specification it will always be possible to argue that it does not adequately represent some particular feature of the system's behaviour, but the model should be chosen to emphasise those aspects of the behaviour over which some property will be analysed.

The example system models in this work are chosen with one objective: clarity and simplicity of the examples. As such they will be deliberately crude models of the systems in questions. They will use very basic models of system

memory access as a simple function from address to stored value, and they will encode processor registers as first class elements of the system state.

Figure 3.1 shows a simple representation of the address space and six 32bit registers of an Intel x86 system. The invariants encode the relationship between the extended registers $eax$, $ebx$, $ecx$, and $edx$, and their smaller, historic components $ax$, $ah$, $al$, etc. The *zero flag* and *carry flag* components of the status register are represented by $zf$, $sf$, and $cf$. The complete status register could be represented, including the necessary invariants to tie the values of $zf$, $sf$, and $cf$ in with its 32-bit word interpretation but that would not add usefully to the examples that will appear here.

Figure 3.2 shows an even simpler representation of an Intel x86 system. This ignores the legacy registers and uses a function to model the contents of the registers. This is the system specification that will be used for the remainder of this thesis, principally because using the function model allows the use of the Z function overriding operator $\oplus$ to change one register value whilst leaving the others unchanged. This will occur often in the instruction set specifications and the requirement to explicitly state the unchanged nature of all the other registers would vastly increase the textual size of the specifications. Once the analysis process is being run automatically this will be irrelevant since much of the repetition will be removed by the simplification stage, but the simplified version is used for the sake of readable examples whilst the details of the analysis are being described.

Figure 3.3 shows a simple definition of the state of a PowerPC system with its 32-bit memory address space and its 32 registers ($r0$ through $r31$ — each holding a 32-bit integer value). The PowerPC assembly code represents registers with integer values rather than names, so this function representation will make for clearer instruction set specifications that will be instantiated to statements like $registers(1)' = memory(43)$.

All of these system specifications are inadequate to capture the behaviour of the entire instruction sets of the processors in question, but they are adequate to capture the behaviour of the small subset of the instruction sets that will appear in the examples in subsequent sections. The analysis process will have no direct involvement with the *System* schema. Users of the analysis process are free so specify as large or small as *System* schema as they need for their verification, but it is likely that they will similarly restrict themselves to only those components that are necessary to verify the properties of interest.

### 3.3.2 Sequential instruction specifications

The analysis process converts the instructions in a disassembled executable file into a formal model of their behaviour by referring to a specification of the behaviour of the CPU in response to its instruction set. The instruction set is separated into *sequential* and *branch* instructions. This section considers the first of these and presents a *template specification* format that allows the instruction set to be specified in a general way, such that it can be instantiated automatically for each instruction that is encountered.

$$INT32 == \{x : \mathbb{Z} \mid -2^{31} < x < 2^{31}\}$$
$$INT16 == \{x : \mathbb{Z} \mid -2^{15} < x < 2^{15}\}$$
$$INT8 == \{x : \mathbb{Z} \mid -2^{7} < x < 2^{7}\}$$
$$WORD32 == \{x : \mathbb{N} \mid x < 2^{32}\}$$
$$WORD16 == \{x : \mathbb{N} \mid x < 2^{16}\}$$
$$WORD8 == \{x : \mathbb{N} \mid x < 2^{8}\}$$
$$BIT == \{x : \mathbb{N} \mid x \leq 1\}$$

*System*

$memory : INT32 \rightarrow INT32$
$eax : INT32$
$ax : INT16$
$ah, al : INT8$
$ebx : INT32$
$bx : INT16$
$bh, bl : INT8$
$ecx : INT32$
$cx : INT16$
$ch, cl : INT8$
$edx : INT32$
$dx : INT16$
$dh, dl : INT8$
$esp : INT32$
$ebp : INT32$
$zf : BIT$
$cf : BIT$
$sf : BIT$

---

$ax = (ah * 256) + al$
$eax \bmod 2^{16} = ax$
$bx = (bh * 256) + bl$
$ebx \bmod 2^{16} = bx$
$cx = (ch * 256) + cl$
$ecx \bmod 2^{16} = cx$
$dx = (ah * 256) + dl$
$edx \bmod 2^{16} = dx$

Figure 3.1: A simple model of the state of an x86 system

$REGNAMES == \{eax, ebx, ecx, edx, esp, ebp\}$

*System*

$memory : INT32 \to INT32$
$registers : REGNAMES \to INT32$
$ioports : INT32 \to INT32$
$zf : BIT$
$cf : BIT$
$sf : BIT$

Figure 3.2: An even simpler model of the state of an x86 system

*System*

$memory : WORD32 \to INT32$
$registers : WORD32 \nrightarrow INT32$

$\mathrm{dom}\, registers = \{x : \mathbb{N} \mid x < 32\}$

Figure 3.3: A simple model of the state of a PowerPC system

Having specified the state of the system in §3.3.1, it is now necessary to specify the operation of each of the instructions. An assembly language program consists of large numbers of instruction instances, each of which contains an *op code mnemonic* and some number of arguments. The mnemonic identifies the particular instruction from the instruction set. This work is not concerned with the binary encoding of these components (RISC vs CISC, little-endian bytes, big-endian words, etc.) as these details are handled by GNU objdump in the disassembly stage. Consequently, the instruction instances will have identifiable mnemonics and parsable argument values. The argument encoding varies between different platforms. In some cases there are multiple different encodings available for one platform[1].

This analysis process separates the instruction set into two parts: those instructions that alter the control flow of the program, and those that do not. Branch instructions will be discussed in §3.3.3; this section deals with purely sequential instructions. The analysis will identify all the branch instructions in

[1]Intel platforms can use the Intel assembly syntax, as used in the Intel instruction set specification documents, or the AT&T syntax, which is the default output format from objdump. The examples that follow expect the analysis input to be objdump output files, so use the AT&T syntax. The argument ordering is the significant difference between the two — *mov ebx*, 0*ffh* in Intel syntax becomes *movl* \$0*xff*, %*ebx* in AT&T syntax. GNU objdump is capable of outputting Intel syntax, but the explicit typing of the AT&T syntax arguments is a useful aid to the parser.

the program in the *Branch Identification* stage (see §5.2) by identifying all the mnemonics presented in the branch instruction set specification. Any remaining instructions will be assumed to be sequential instructions.

### Sequential instruction specifications

The structure of a processor's Arithmetic Logic Unit (ALU) is such that the op code defines the behaviour that will be performed. In order to specify the behaviour of the instruction set, the analysis process user must supply one Z operation schema for each op code. These will be identified by being named for the assembly mnemonic, for example the *leave* mnemonic in Figure 3.4.

Figure 3.4: A simple specification for an instance of the x86 *leave* instruction

### Sequential blocks

All instructions that alter the control flow of the program should be handled as branch instructions, which are discussed in §3.3.3. Consequently those remaining as sequential instructions can be represented by Z sequential composition of the individual operation schema.

Figure 3.5 shows an example of a short block of sequential instructions. The subscripts on the schema names are the hexadecimal addresses of the instructions in the disassembler output. The analysis process retains these addresses to aid debugging during verification. If a problem is detected when the produced model is compared to the design specification it will be possible to trace the problem back to a small portion of the assembly code. It could be argued that this is of only limited value if the executable has been compiled from a high level language but modern compilers can include considerable debugging information to aid traceability back to the higher level language. Additionally, this analysis process is intended for use in specialist, hardware dependent software. In such an environment the developers are likely to be more comfortable than most with reading the assembly code.

The example in Figure 3.5 is relatively short but blocks of sequential instructions can be fairly long. The techniques presented in §4 were created specifically to simplify these blocks. The mechanics of the simplification process are detailed in §5.4.

All of these examples import the entire *System* schema and only operate on part of it without specifying that the other elements are unchanged. In

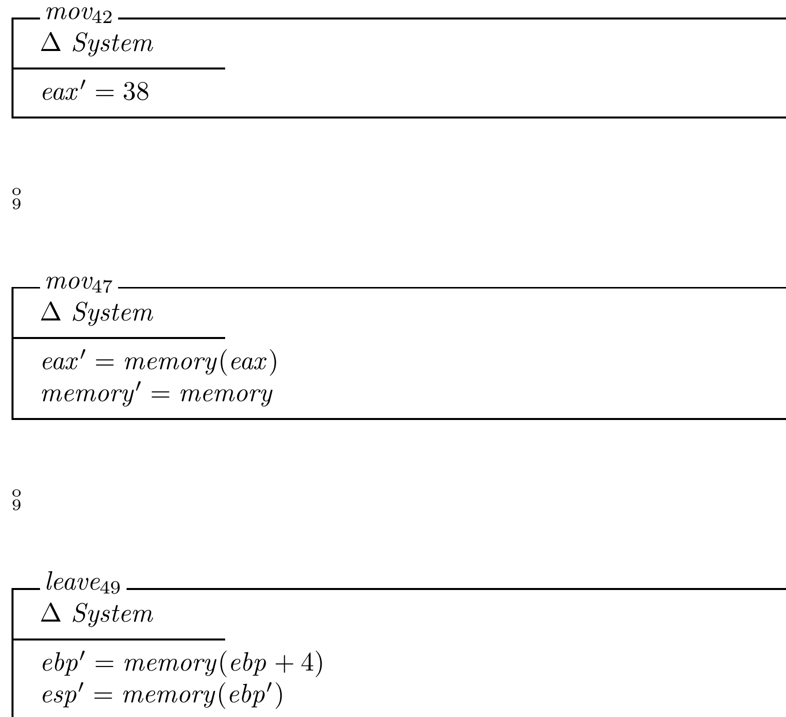



Figure 3.5: A short block of sequential instructions

Z these operations do not automatically maintain the state of elements whose post-condition is not explicitly set, but this detail is omitted from these and all subsequent examples for brevity and clarity. This behaviour of the Z notation was one of the explicit choices in the selection of CSP-OZ as the language for this project since some processor instructions will *not* maintain the status of some components of the system that are used for temporary data storage in the execution of the instruction.

### 3.3.3 Branch instruction set specification

Template specifications for sequential instructions were introduced in §3.3.1, along with a machine readable template specification representation. This section takes a similar approach to *branch instructions* and expands the template representation to include such information as is necessary to capture the control flow information present in these instructions. This will be used in §5.5 (*Program encapsulation*) to produce the CSP component of the final model, but is also used in §5.2 (*Branch identification*) to break the assembly instructions up into sequential blocks.

The analysis process is as independent as possible from the model of the target system but program control flow was considered sufficiently important to be handled explicitly. Whilst the behaviour of the sequential components can be represented by Z operation schema, the control flow and branching behaviour is most elegantly modeled in CSP (see §2.4.4 and Hoare [36]). The CSP-OZ language (§2.4.6 and Fischer [30]) allows for the combination of the CSP control flow components with the Z state and sequential operation schema.

Once the sequential blocks have been identified and instantiated the CSP can be used to thread them together. For unconditional branching it is a simple matter of defining the CSP process. The assembly instruction:

```
e: eb 03                    jmp    13 <max+0x13>
```

becomes:

$$Branch_e = Block_{13} \rightarrow Leave$$

In this case the sequential block at address 13 ends with a function return (which is discussed in detail in §3.3.3). The entire control flow within a function can be represented by a sequence of such CSP processes that execute the sequential operations and the evolve to the next branch.

**Branch and no-branch prefixes**

Most branch instructions are *conditional* — that is, control flow may branch to a different point or continue sequentially depending on some condition on the system state. The combination of CSP and Z also provides a way to represent conditional branching with the control flow modeled in CSP and the state-derived conditions modeled in Z. The CSP external choice operator can compose two processes and allow a systematic choice between the paths. By prefixing the two paths with a Z schema containing precondition invariants it is possible to make the choice in the formal model correspond with the choice made by the implementation.

The specification fragment Figure 3.6 represents the conditional branch instruction ($b$ is the hexadecimal address of the instruction immediately following this one, used for the *NoBranch* condition):

```
9: 76 05                    jbe    10 <max+0x10>
```

Retaining the address of instructions in the names of sequential blocks serves a useful purpose here, as well as its use in debugging. So long as the target of the branch can be resolved to an instruction address it is simple to automatically produce the CSP representations. The analysis process assumes that such a resolution is possible and that it is performed by the disassembler. In the case of instructions with literal values this is simple. Instruction sets also contain branch instructions with more complex arguments — dynamic values from registers or memory addresses for example. In this case it will be more difficult

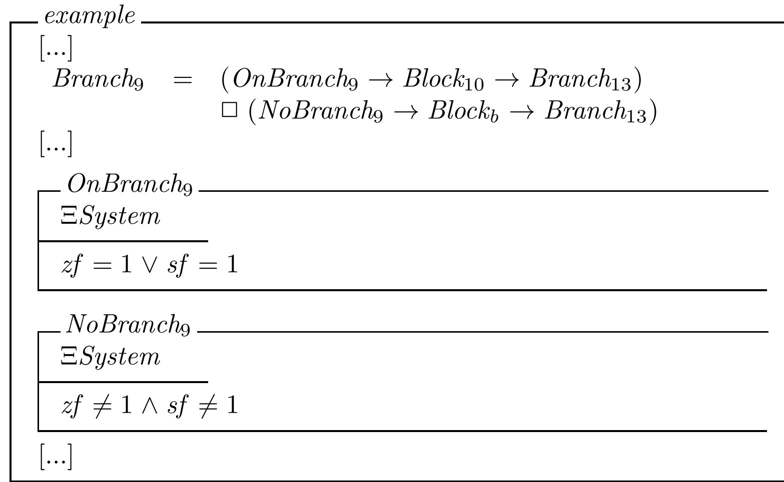


Figure 3.6: A conditional branch specification

to evaluate the instruction. It should not be impossible; static evaluation of the program up to this point should yield either an exact value or a limited set of possibilities. This level of analysis is not considered further in this document. Given that the target for this analysis is sub-components of highly safety critical systems it is reasonable to require a restriction on this style of programming. Byzantine structures such as dynamic program branching can be excluded by coding standards in safety critical applications, without adversely affecting the development effort.

### Function call instructions

In addition to short program jumps within a function, the analysis process considers function calls and function returns. Most instruction sets have explicit instructions for this behavior that differ from simple jump instructions. In principle the same behaviour can be performed with jump instructions and it is possible to represent this by specifying *long jump* instructions to be function call instructions, or specific patterns, such as jumps to the address at the stack base pointer, as function returns. In most cases — especially those derived from compiled C or other high level language programs — there will be a clear separation between local branching and function calls. Since GNU objdump is capable of interpreting executable file symbol tables with enough detail to identify original functions it is advantageous to retain this information in the structure of the produced model.

In any well-engineered program functions should represent self-contained blocks of program that perform some clear operation. It is here that the third

component of CSP-OZ is used: the Object Orientation. Representing functions as CSP-OZ classes suitably collects the sequential and branching behaviour into operational units. It only remains to represent the entry and return from these functions. The operation of imperative programs is for each function to alter the system state in some way defined on the parameters provided as part of the system state when the function is called.

Figure 3.7 presents the CSP-OZ class that all functions will extend. The *System* schema from §3.3.1 is included as the state for the class. The mechanism of function calls and returns makes use of schema promotion to pass the system state to the called function, allow it to alter the system state as designed, and then return the new system state, which is unified into the calling function's local state before execution continues.

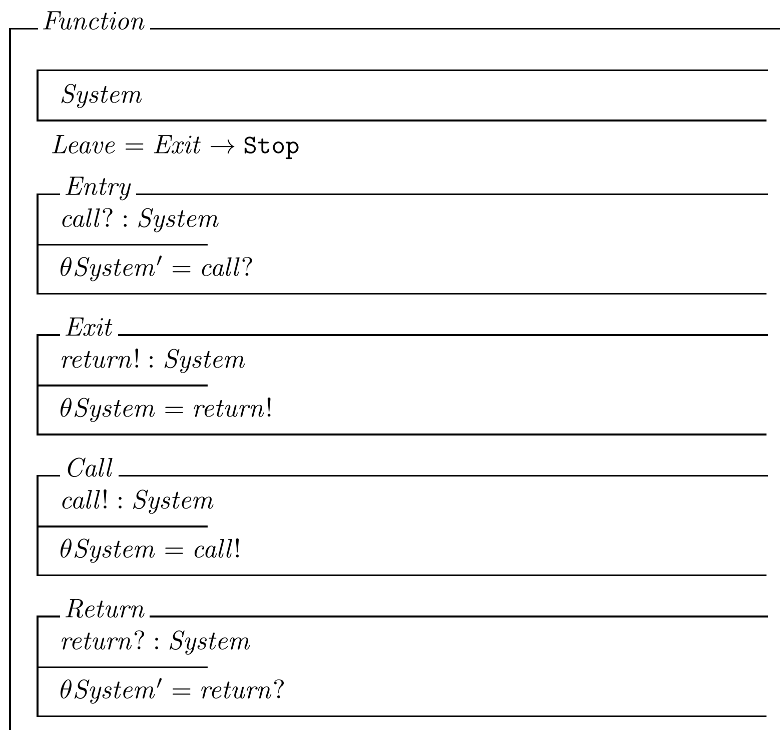


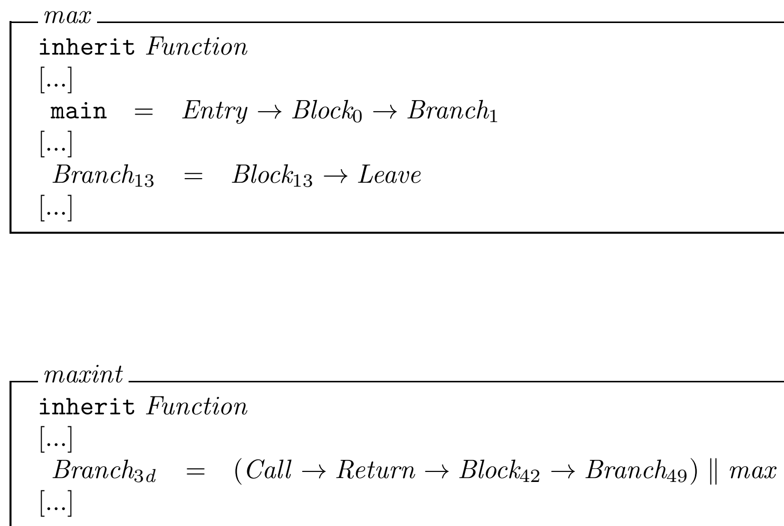

Figure 3.7: Function call operations

An example of a function call is this instruction:

```
80483fd: e8 be ff ff ff        call   80483c0 <max>
```

This has been taken from a complete executable file so that the target address is fully resolved, and objdump has identified that 80483*c*0 is the start of the *max*

function. Assuming that the *max* function has been analysed into a suitable CSP-OZ class this instruction can be represented by the specification components shown in Figure 3.8. The CSP sequence ($Call \rightarrow Return \rightarrow Block_{42} \rightarrow Branch_{49}$) is executed in parallel with the *max* class. This will synchronise on the *call* communication and pass the *System* state into the *max* function. Since the CSP sequence immediately executes the *Return* event, which synchronises on the *return* input, it will wait for the *max* function to return. When the *max* function executes the *Leave* event it will return the updated *System* to the calling function, which can then continue with the next *Block* event in its control flow.



*max*

inherit *Function*
[...]
main $= Entry \rightarrow Block_0 \rightarrow Branch_1$
[...]
$Branch_{13} = Block_{13} \rightarrow Leave$
[...]

*maxint*

inherit *Function*
[...]
$Branch_{3d} = (Call \rightarrow Return \rightarrow Block_{42} \rightarrow Branch_{49}) \parallel max$
[...]

Figure 3.8: Function call example

## 3.4 Automated inference

The automatic analysis of a given executable to produce a model of the form described in §3.3 will be broken into discrete stages that allow for as much parallel processing as possible. This allows problems of pure scale to be tackled most efficiently by the available resources and allows for the greatest impact of increased resources.

The phases in the analysis workflow are:

- **Loading of 3 specification sets**
  - System specification

  - Sequential instruction set
  - Branch instruction set

- **Disassembly of executable and parsing of objdump output** The objdump information provides the functions and some symbol information.
- **Branch identification** Identifying branch instructions and their potential targets to separate the code into sequential blocks.
- **Formal instantiation** Each sequential instruction is matched to a template from the instruction set specification and that template instantiated for the individual instruction.
- **Simplification** The compression of sequential blocks into as few Z operation schemas as possible — ideally a single schema per block.
- **Program encapsulation** Collection of the sequential blocks and the instantiation of branch instructions into CSP-OZ representing the internal control flow and the external function call mechanisms.

The output of the final stage is a set of CSP-OZ classes that model the complete behaviour of the executable to a level of detail determined by the supplied specifications.

# Chapter 4

# Separation in Z

## 4.1 Overview

One of the critical difficulties with the analysis of executable files is the sheer number of instructions that must be analysed and whose behaviour must be incorporated into the produced model. The concept of *sequential blocks* is introduced in §3.3.2. These consist of chains of instructions that are sequentially composed as they do not influence the control flow of the program. It makes sense to compress these into one single Z operation schema that represents the effect in the system state of this chain of sequential operations. To correctly determine the effect of a chain of Z operation schema requires some reasoning about the semantics of each operation and its influence on the system state that forms the precondition of the following operation, but this level of reasoning requires considerable computing power and so impacts on the scalability of the process. To counter this problem this section presents a technique to determine whether the precondition of a Z operation is impacted by the preceding operation using only syntactic analysis. This can be implemented with a very limited Z parser and a series of string comparison operations that can be highly optimised. Where two schema do not interact they can be compressed into one by simply concatenating their state and invariant statements. Not all sequential compositions will be independent but if this method can remove a considerable percentage of the compositions — as seems likely with simple operations as might be expected in assembly instructions — then the remaining specification will have gained comprehensibility for a limited computation expense.

The contents of this section were presented as a short talk at ABZ 2008[55]

## 4.2 Introduction

When trying to reason about a long chain of sequentially composed operations (such as a program in a sequential, imperative language) it would be helpful to be able to group together sequences of instructions and consider their overall

effects. In order to determine the overall effect of a sequence it must first be determined how the individual effects of the operations interact. The simplest but most crucial question is whether they interact at all. Here we shall attempt to develop a test for Z operations that will determine whether the effects of an operation are altered by the effects of the preceding operation.

Where an operation is unaffected by the preceding operation we shall say that they are *Separate* and shall use the symbol '&' after Reynolds [50].

## 4.3 Basic case

For any Z operation we can require that every state variable referenced is brought into scope explicitly. We can then collect these variables together into two state schemas that we shall call $State_\Delta$ and $State_\Xi$. The second of these, $State_\Xi$, will contain all the variables that are unchanged by the operation. The first schema, $State_\Delta$, will contain all the others - i.e. all the variables that are changed or whose resulting state is left unspecified.

This will allow us to rewrite an operation as follows:

```
┌─ P ─────────────────────────
│ x, x' : ℕ
│ y, y' : ℕ
├──────────
│ x' = x + y
│ y' = y
└─────────────────────────────
```

$$State_\Delta == [x : \mathbb{N}]$$
$$State_\Xi == [y : \mathbb{N}]$$

Rewriting $P$:

```
┌─ P' ────────────────────────
│ Δ State_Δ
│ Ξ State_Ξ
├──────────
│ x' = x + y
│ y' = y
└─────────────────────────────
```

From now on we shall treat any operation as possessing a $State_\Delta$ and $State_\Xi$ and shall use the notation $Op.State_\Delta$ and $Op.State_\Xi$ to refer to them. We shall also use set notation on these schemas, so $Op_1.State_\Delta \cap Op_2.State_\Delta$ refers to those variables that are changed by both $Op_1$ and $Op_2$.

Being able to refer to all the variables changed and unchanged by an operation allows us to write the most general definition of our version of *Separation*. The notation we shall use is $Op_1 \& Op_2$ if, in the sequential composition $Op_1 \mathbin{⨟} Op_2$, the effect of $Op_2$ is unaffected by the actions of $Op_1$. The important part of this

for our aim is that this should mean that the precondition of the whole composition is simply the conjunction of the preconditions of the two operations, and the post condition is simply the conjunction of the two post conditions.

The naively simple definition would be just:

$$Op_1.State_\Delta \cap Op_2.State_\Delta = \varnothing$$

That is: the two operations change entirely different variables and so are unaffected by each other. This is not adequate, however, as our $P$ operation above demonstrates. If we introduce another operation, $Q$, that alters only $y$ we would expect to - and our naive rule would allow us to - be able to sequentially compose $Q$ and $P$ without problems:

| $Q$ |
|---|
| $y, y' : \mathbb{N}$ |
| $y' = y + 1$ |

Now consider the composition $Q \fatsemi P$. Our naive rule is satisfied, since $Q$ changes only $y$ and $P$ changes only $x$ the intersection is empty. However the effect of $P$ is altered by the composition with $Q$ since it *depends* on the value of $y$ when increasing $x$. Consequently we need to check that the first operation does not change *any* variables that are used *anywhere* in the second operation.

This is easily done with our *State* definitions since $Op.State_\Delta \cup Op.State_\Xi$ will include all the variables in scope in the operation. Our new definition for separation is then:

$$Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi) = \varnothing \quad \langle \texttt{Total Separation} \rangle$$

For our $Q \fatsemi P$ example this new rule will correctly fail to prove separation since $P.State_\Delta \cup P.State_\Xi$ includes both $x$ and $y$, and $Q.State_\Delta$ includes $y$, so the intersection is non-empty $(y)$. However the opposite ordering, $P \fatsemi Q$, *is* separate according to this new rule - and our natural language description above.

$$P.State_\Delta = \{x\}$$
$$P.State_\Xi = \{y\}$$
$$Q.State_\Delta = \{y\}$$
$$Q.State_\Xi = \varnothing$$

The `Total Separation` definition is then instantiated:

$$\begin{array}{ll} & \{x\} \cap (\{y\} \cup \varnothing) = \varnothing \\ \equiv & \langle \texttt{union with emptyset} \rangle \\ & \{x\} \cap \{y\} = \varnothing \\ \equiv & \langle \texttt{definition of intersection} \rangle \\ & \varnothing = \varnothing \\ \equiv & \langle \texttt{reflexivity of } = \rangle \\ & \text{true} \end{array}$$

QED

This also serves as an example of the difference between separation and commutativity. When first presented with the idea of separation used here many people suggest it is only the same as commutativity - that is if two operations don't affect each other's behaviour they can be sequentially composed in either order. Our $Q \fatsemi P$ example illustrates why this is not the case. In our study of separation here we are not interested in showing that two operations *never* interfere with each other, simply that in the presented usage their effects on the state of the system are independent. Our model of separation also requires that the precondition of the second operation is not modified by the first. In this case a failure to prove separation may be a positive thing for analysis since it suggests an interplay between the operations that could simplify the overall conditions - such as the post-condition of the first *necessarily* satisfying a precondition of the second, and so removing it from consideration elsewhere.

## 4.4 A relaxation of the basic case

Although the `Total Separation` definition presented above does satisfy our requirements for identifying independent operations it is too restrictive in some cases. Consider the following operations:

$R$

$$x, x' : \mathbb{N}$$

$$x' = x + 1$$

$S$

$$x, x' : \mathbb{N}$$
$$y, y' : \mathbb{N}$$

$$x' = x$$
$$y' = y + 1$$

Although the use of $x$ in $S$ seems foolishly Byzantine, it is a situation that occurs often in Z specifications since system state is often contained in reasonably sized state schemas and the entire schema is brought into scope when a change is made.

In this case our `Total Separation rule` will not allow us to prove $R \& S$, that is we can't show that the operation of $S$ is unaffected by $R$ in the composition $R \fatsemi S$. This is clearly an over-restriction, since $S$ doesn't use the value of $x$ in determining any of its effects. Also, since $S$ leaves $x$ unchanged it doesn't affect our other requirement: if we want to consider the overall effect of the composition we need to know that any effects produced by $R$ aren't undone by $S$. With `Total Separation` this was implicit, since the second operation wasn't allowed to even *mention* anything changed by the first. When relaxing

the rule to allow situations such as $R\&S$ we must be careful not to violate this requirement.

A naive approach would be to require that, if a variable is mentioned by both of the operations then one of them must leave it unchanged. Just as we created $State_\Delta$ and $State_\Xi$, we can imagine that a Z operation has a property called *Propositions* that is the set of propositions in the schema. We can continue to abuse set notation, so we can talk about a proposition being in this set, e.g. $p \in Op.Propositions$. We can use this for our relaxed definition:

$$\begin{array}{l} \forall\, x \mid x \in (Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \bullet \\ \quad ([x' = x] \in Op_1.Propositions) \vee ([x' = x] \in Op_2.Propositions) \end{array}$$

This seems ok - and it will correctly prove $R\&S$ - but it is now too weak. Although it requires $x$ to be unchanged it no longer prevents the value of $x$ being used in determining the value of another variable. The example of $Q\&P$ from §4.3 would be proven using this rule, since $P$ does contain the proposition $[y' = y]$. Unfortunately, since it relies on the value of $y$ when modifying $x$, it should not be considered *Separate*.

We need to specify that one of the operations should *only* refer to the variable in the proposition that leaves it unchanged. So, if both operations refer to $x$ one of them should only do so to state that it is unchanged; that is it should contain $[x' = x]$ and no other propositions that refer to $x$. The neatest way to identify this property is to say that, if we removed the statement $[x' = x]$ from one of the operations it would no longer refer to $x$ in any way. This gives us the following new definition for separation:

$$\begin{array}{l} \forall\, x \mid x \in (Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \bullet \\ \quad \exists\, Op_{2a} \mid Op_2 == Op_{2a} \wedge [\Xi x] \bullet Op_1 \& Op_{2a} \\ \qquad\qquad\qquad\qquad\qquad\qquad \langle \texttt{Effective Separation} \rangle \end{array}$$

Clearly the separation requirement in these predicates can be satisfied either by the `Total Separation` definition from §4.3 or by recursive application of this definition if there are multiple variables to be considered.

We can now prove the separation $R\&S$:

$\forall x \mid x \in (Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \bullet$
$\quad \exists Op_{2a} \mid Op_2 == Op_{2a} \land [\Xi x] \bullet Op_1 \& Op_{2a}$
$=$ ⟨`Instantiating for` $R$ `and` $S$⟩
$\forall x \mid x \in (\{x\} \cap (\{y\} \cup \{x\})) \bullet$
$\quad \exists S_a \mid S == S_a \land [\Xi x] \bullet R \& S_a$
$=$ ⟨`Set union and intersection`⟩
$\forall x \mid x \in \{x\} \bullet$
$\quad \exists S_a \mid S == S_a \land [\Xi x] \bullet R \& S_a$
$=$ ⟨`Universal quantification over a single element set`⟩
$\exists S_a \mid S == S_a \land [\Xi x] \bullet R \& S_a$
$=$ ⟨`One point rule using` $[y, y' : \mathbb{N} \mid y' = y + 1]$ `for` $S_a$⟩
$R \& [y, y' : \mathbb{N} \mid y' = y + 1]$

**Proof of $R \& [y, y' : \mathbb{N} \mid y' = y + 1]$ by `Total Separation`**

$Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi) = \varnothing$
$=$ ⟨`Instantiating for` $R$ `and` $[y, y' : \mathbb{N} \mid y' = y + 1]$⟩
$\{x\} \cap (\{y\} \cup \varnothing) = \varnothing$
$=$ ⟨`Union with empty set`⟩
$\{x\} \cap \{y\} = \varnothing$
$=$ ⟨`Set intersection`⟩
$\varnothing = \varnothing$
$=$ ⟨`Reflexivity of` $=$⟩
*true*
QED

## 4.5 A simple function case

We now have two rules for separation that give us the appropriate results for simple Z operations where we require that the variables modified and referred to are different between the operations. However, it is quite common for parts of a system to be specified as relations. First we will look at functions - which are only a restricted type of relation - then we will generalise this to relations in §4.6

$A == \{1, 2\}$

*T*
$f, f' : A \rightarrow \mathbb{N}$
---
$f'(1) = 42$
$f'(2) = f(2)$

*U*
$f, f' : A \leftrightarrow \mathbb{N}$
---

$f'(1) = f(1)$
$f'(2) = 1$

If we try to prove $T \& U$ using the separation rules from the previous sections we will fail, since the variable $f$ is mentioned and modified by both operations. However, $T$ modifies $f(1)$ but leaves $f(2)$ unaltered; $U$ modifies $f(2)$ (without reference to $f(1)$) and makes no change to $R(1)$. If we consider each element of the domain of the relation to be a distinct variable it is clear that these operations should be considered *separate* - the changes wrought by $T$ do not influence and are not undone by $U$.

A naive solution would be to simply identify operations that referred to the same variable but the domain of their referral was disjoint. Unfortunately this would fail since leaving an after-state unspecified in Z allows the implementation to modify the value or not as it sees fit. We could not argue that the operation did not alter the value since an implementation could legitimately assign any value to the elements outside the specified range - they are "unspecified"!

Consequently we must construct a rule that requires that any elements of the function used by the second operation are left unchanged by the first and any changes that are made are by the first operation are not changed or referenced by the second operation. This is identical to the requirement we made about variables in previous sections. As we have said above, unless an element of the function is explicitly unchanged we have to assume it has changed. Using a similar style to §4.4 we shall define this using schema conjunction.

A simple solution to the functional case is to define a rule that, for each element of a function's domain, requires that one or the other operation leaves the function unchanged.

$$\forall f : A \to B \mid f \in (Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \bullet$$
$$\bigwedge_{a \in A} \left( \begin{array}{l} \left( \begin{array}{l} \neg \; \exists \, Op_1' \bullet Op_1 == \\ \qquad Op_1' \wedge [f : A \to B \mid f'(a) = f(a)] \end{array} \right) \\ \qquad \Rightarrow \left( \begin{array}{l} \exists \, Op_2' \bullet Op_2 == \\ \qquad Op_2' \wedge [f : A \to B \mid f'(a) = f(a)] \end{array} \right) \end{array} \right)$$

Although this rule requires the invention of an $Op_1'$ or an $Op_2'$ it should be easy to do this syntactically. A Z operation on a function will appear (or can be re-written) as a series of propositions on elements of the function. Simply removing the item that appears as $f'(a) = f(a)$ from the text will leave the required operation.

Unfortunately, this rule isn't complete. It correctly defines our separation rule for the functional element, but we need a recursive part to take care of any other (possibly non-functional) variables in the operations. If the operations have met the separation conditions for the functional variable $f$ then we can simply require - recursively - that the operations meet *some* separation rule without the function f (i.e. for all their other variables). We produce the following rule:

$$
\forall f : A \to B \mid f \in (Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \bullet
\bigwedge_{a \in A} \left( \begin{array}{l} \left( \begin{array}{l} \neg\ \exists\, Op_1' \bullet Op_1 == \\ \quad Op_1' \wedge [f : A \to B \mid f'(a) = f(a)] \end{array} \right) \\ \quad \Rightarrow \left( \begin{array}{l} \exists\, Op_2' \bullet Op_2 == \\ \quad Op_2' \wedge [f : A \to B \mid f'(a) = f(a)] \end{array} \right) \end{array} \right)
\wedge \left( \begin{array}{l} (Op_1.State_\Delta \backslash \{f\} \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \neq \varnothing \\ \Rightarrow \exists\, Op_1' \bullet \\ \quad Op_1'.State_\Delta = Op_1.State_\Delta \backslash \{f\} \\ \quad \wedge\ Op_1'.State_\Xi = Op_1.State_\Xi \backslash \{f\} \\ \quad \wedge \left( \begin{array}{l} \exists\, Op_{1f} \bullet \\ \quad Op_{1f}.State_\Delta = \{f\} \wedge Op_{1f}.State_\Xi = \varnothing \\ \quad \wedge\ Op_1 == Op_1' \wedge Op_{1f} \end{array} \right) \\ \quad \wedge\ Op_1' \& Op_2 \end{array} \right)
$$

$\langle$`Functional Separation`$\rangle$

We should now be able to prove $T \& U$.

$$
\begin{array}{ll}
 & \langle \texttt{Instantiate functional separation for } T \texttt{ and } U \rangle \\
 & \forall f : A \to B \mid f \in (\{f\} \cap (\{f\} \cup \varnothing)) \bullet \\
 & \bigwedge_{a \in A} \left( \begin{array}{l} \left( \begin{array}{l} \neg\ \exists\, T' \bullet T == \\ \quad T' \wedge [f : A \to B \mid f'(a) = f(a)] \end{array} \right) \\ \quad \Rightarrow \left( \begin{array}{l} \exists\, U' \bullet U == \\ \quad U' \wedge [f : A \to B \mid f'(a) = f(a)] \end{array} \right) \end{array} \right) \\
 & \wedge \left( \begin{array}{l} (T.State_\Delta \backslash \{f\} \cap (U.State_\Delta \cup U.State_\Xi)) \neq \varnothing \\ \Rightarrow \exists\, T' \bullet \\ \quad T'.State_\Delta = \{f\} \backslash \{f\} \\ \quad \wedge\ T'.State_\Xi = \varnothing \backslash \{f\} \\ \quad \wedge \left( \begin{array}{l} \exists\, T_f \bullet \\ \quad T_f.State_\Delta = \{f\} \wedge T_f.State_\Xi = \varnothing \\ \quad \wedge\ T == T' \wedge T_f \end{array} \right) \\ \quad \wedge\ T' \& U \end{array} \right) \\
= & \langle \texttt{Set Union and intersection} \rangle \\
 & \forall f : A \to B \mid f \in \{f\} \bullet \\
 & \quad [...]
\end{array}
$$

$=$ ⟨`Universal quantification over a one-element set`⟩

$$\bigwedge_{a \in A} \left( \left( \begin{array}{l} \neg\ \exists\, T' \bullet T == \\ \qquad T' \wedge [f : A \to \mathbb{N} \mid f'(a) = f(a)] \end{array} \right) \Rightarrow \left( \begin{array}{l} \exists\, U' \bullet U == \\ \qquad U' \wedge [f : A \to \mathbb{N} \mid f'(a) = f(a)] \end{array} \right) \right)$$

$$\wedge \left( \begin{array}{l} (T.State_{\Delta} \backslash \{f\} \cap (U.State_{\Delta} \cup U.State_{\Xi})) \neq \varnothing \\ \Rightarrow \exists\, T' \bullet \\ \quad T'.State_{\Delta} = T.State_{\Delta} \backslash \{f\} \\ \quad \wedge\ T'.State_{\Xi} = T.State_{\Xi} \backslash \{f\} \\ \quad \wedge \left( \begin{array}{l} \exists\, T_f \bullet \\ \qquad T_f.State_{\Delta} = \{f\} \wedge T_f.State_{\Xi} = \varnothing \\ \qquad \wedge\ T == T' \wedge T_f \end{array} \right) \\ \quad \wedge\ T' \& U \end{array} \right)$$

$=$ $\left\langle \texttt{Expanding } \bigwedge_{a \in A} \right\rangle$

$$\left( \left( \begin{array}{l} \neg\ \exists\, T' \bullet T == \\ \qquad T' \wedge [f : A \to \mathbb{N} \mid f'(1) = f(1)] \end{array} \right) \Rightarrow \left( \begin{array}{l} \exists\, U' \bullet U == \\ \qquad U' \wedge [f : A \to \mathbb{N} \mid f'(1) = f(1)] \end{array} \right) \right)$$
$$\wedge$$
$$\left( \left( \begin{array}{l} \neg\ \exists\, T' \bullet T == \\ \qquad T' \wedge [f : A \to \mathbb{N} \mid f'(2) = f(2)] \end{array} \right) \Rightarrow \left( \begin{array}{l} \exists\, U' \bullet U == \\ \qquad U' \wedge [f : A \to \mathbb{N} \mid f'(2) = f(2)] \end{array} \right) \right)$$
$$\wedge\ (\ [...]\ )$$

$=$ ⟨`'One point' rule using` $[f : A \to \mathbb{N} \mid f'(1) = 42]$ `as` $T'$ `in the second predicate`⟩

$$(\ [...]\ )$$
$$\wedge$$
$$\left( \neg\ true \Rightarrow \left( \begin{array}{l} \exists\, U' \bullet U == \\ \qquad U' \wedge [f : A \to \mathbb{N} \mid f'(2) = f(2)] \end{array} \right) \right)$$
$$\wedge\ (\ [...]\ )$$

$=$ ⟨`Defn. of` $\neg$⟩

$$(\ [...]\ )$$
$$\wedge$$
$$\left( false \Rightarrow \left( \begin{array}{l} \exists\, U' \bullet U == \\ \qquad U' \wedge [f : A \to \mathbb{N} \mid f'(2) = f(2)] \end{array} \right) \right)$$
$$\wedge\ (\ [...]\ )$$

$$
\begin{array}{l}
= \quad \langle \texttt{Defn. of } \Rightarrow \rangle \\
\quad ( \ [...] \ ) \\
\quad \wedge \\
\quad \mathit{true} \\
\quad \wedge \ ( \ [...] \ )
\end{array}
$$

$$
\begin{array}{l}
= \quad \langle \texttt{Defn. of } \wedge \rangle \\
\quad \left( \begin{array}{l} \left( \begin{array}{l} \neg \ \exists \ T' \bullet T == \\ \qquad T' \wedge [f : A \rightarrow \mathbb{N} \mid f'(1) = f(1)] \end{array} \right) \\ \qquad \Rightarrow \left( \begin{array}{l} \exists \ U' \bullet U == \\ \qquad U' \wedge [f : A \rightarrow \mathbb{N} \mid f'(1) = f(1)] \end{array} \right) \end{array} \right) \\
\quad \wedge \ ( \ [...] \ )
\end{array}
$$

$$
\begin{array}{l}
= \quad \left\langle \begin{array}{l} \texttt{Nothing can be found to instantiate } T', \\ \texttt{so defn. of } \neg \ \exists \end{array} \right\rangle \\
\quad \left( \mathit{true} \quad \Rightarrow \left( \begin{array}{l} \exists \ U' \bullet U == \\ \qquad U' \wedge [f : A \rightarrow \mathbb{N} \mid f'(1) = f(1)] \end{array} \right) \right) \\
\quad \wedge \ ( \ [...] \ )
\end{array}
$$

$$
\begin{array}{l}
= \quad \langle \texttt{Defn. of } \Rightarrow \rangle \\
\quad \left( \begin{array}{l} \exists \ U' \bullet U == \\ \qquad U' \wedge [f : A \rightarrow \mathbb{N} \mid f'(1) = f(1)] \end{array} \right) \\
\quad \wedge \ ( \ [...] \ )
\end{array}
$$

$$
\begin{array}{l}
= \quad \langle \texttt{`One-point' rule using } [f : A \rightarrow \mathbb{N} \mid f'(2) = 1] \texttt{ as } U' \rangle \\
\quad (U == [f : A \rightarrow \mathbb{N} \mid f'(2) = 1] \wedge [f : A \rightarrow \mathbb{N} \mid f'(1) = f(1)]) \\
\quad \wedge \ ( \ [...] \ )
\end{array}
$$

$$
\begin{array}{l}
= \quad \langle \texttt{Schema conjunction and schema equality} \rangle \\
\quad \mathit{true} \\
\quad \wedge \ ( \ [...] \ )
\end{array}
$$

$$
\begin{array}{l}
= \quad \langle \texttt{Defn. of } \wedge \rangle \\
\quad \left( \begin{array}{l} (T.\mathit{State}_\Delta \setminus \{f\} \cap (U.\mathit{State}_\Delta \cup U.\mathit{State}_\Xi)) \neq \varnothing \\ \Rightarrow \exists \ T' \bullet \\ \qquad T'.\mathit{State}_\Delta = \{f\} \setminus \{f\} \\ \qquad \wedge \ T'.\mathit{State}_\Xi = \varnothing \setminus \{f\} \\ \qquad \wedge \left( \begin{array}{l} \exists \ T_f \bullet \\ \qquad T_f.\mathit{State}_\Delta = \{f\} \wedge T_f.\mathit{State}_\Xi = \varnothing \\ \qquad \wedge \ T == T' \wedge T_f \end{array} \right) \\ \qquad \wedge \ T' \& U \end{array} \right)
\end{array}
$$

$$= \langle \texttt{Set hiding} \rangle \left( \begin{array}{l} (\varnothing \cap (U.State_\Delta \cup U.State_\Xi)) \neq \varnothing \\ \Rightarrow \exists\, T' \bullet \\ \qquad [...] \end{array} \right)$$

$$= \langle \texttt{Set intersection on } \varnothing \rangle \left( \begin{array}{l} \varnothing \neq \varnothing \\ \Rightarrow \exists\, T' \bullet \\ \qquad [...] \end{array} \right)$$

$$= \langle \texttt{Defn. of } \neq \rangle \left( \begin{array}{l} \mathit{false} \\ \Rightarrow \exists\, T' \bullet \\ \qquad [...] \end{array} \right)$$

$$= \langle \texttt{Defn. of } \Rightarrow \rangle \; \mathit{true}$$

QED

## 4.6 A simple relational case

Extending our functional separation rule to cover all types of relation can be achieved simply by using the image of the relation instead of the function application. So, $[f : A \rightarrow B \mid f'(x) = f(x)]$ becomes $[r : A \leftrightarrow B \mid r'(\!|\, x \,|\!) = r(\!|\, x \,|\!)]$. The functional separation rule then becomes the relational separation rule.

$$\begin{array}{l} \forall\, r : A \leftrightarrow B \mid r \in (Op_1.State_\Delta \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \bullet \\ \quad \bigwedge_{a \in A} \left( \begin{array}{l} \left( \begin{array}{l} \neg\, \exists\, Op_1' \bullet Op_1 == \\ \quad Op_1' \wedge [r : A \leftrightarrow B \mid r'(\!|\, a \,|\!) = r(\!|\, a \,|\!)] \end{array} \right) \\ \quad \Rightarrow \left( \begin{array}{l} \exists\, Op_2' \bullet Op_2 == \\ \quad Op_2' \wedge [r : A \leftrightarrow B \mid r'(\!|\, a \,|\!) = r(\!|\, a \,|\!)] \end{array} \right) \end{array} \right) \\ \quad \wedge \left( \begin{array}{l} (Op_1.State_\Delta \setminus \{f\} \cap (Op_2.State_\Delta \cup Op_2.State_\Xi)) \neq \varnothing \\ \Rightarrow \exists\, Op_1' \bullet \\ \quad Op_1'.State_\Delta = Op_1.State_\Delta \setminus \{r\} \\ \quad \wedge\, Op_1'.State_\Xi = Op_1.State_\Xi \setminus \{r\} \\ \quad \wedge \left( \begin{array}{l} \exists\, Op_{1r} \bullet \\ \quad Op_{1r}.State_\Delta = \{r\} \wedge Op_{1r}.State_\Xi = \varnothing \\ \quad \wedge\, Op_1 == Op_1' \wedge Op_{1r} \end{array} \right) \\ \quad \wedge\, Op_1' \& Op_2 \end{array} \right) \end{array}$$

$$\langle \texttt{Relational Separation} \rangle$$

This rule has not been demonstrated for all of the relational types in the Z language. It may be that more subtle separation rules could be written for some types, in the same way as the functional rule. However, this general relational rule satisfies the original requirement that operations proven separate certainly do not intefere with each other.

## 4.7 Conclusion

The rules presented here provide a simple test for two sequentially composed operations to determine if the effects of the first modify the behaviour of the second, compared to its operation individually. Although this does not provide any means for analysing the nature of the inteference, it is intended that this form part of a larger analysis. The intention is that the separation tests be used to eliminate from consideration the large number of sequentially composed operations whose behaviours need not be analysed together.

These rules are written in a pure Z form, using existential quantification to identify various arbitray schemas that are used to decompose the original operations. Generating suitable instantiations for these schemas would be a difficult — if not intractable — problem for an automated theorem prover if the rules are considered in isolation. However, it is obvious to any human reader that the schemas are almost invariably formed from the original operation with some stated part removed. Future work on these rules will be directed at both representing the necessary rules in a format suitable for automated proving, and providing heuristics or proof tactics to allow the tools to perform the obvious, syntactic modifications to allow easy instantiation in these cases.

# Chapter 5

# Analysis

## 5.1 Disassembly

### 5.1.1 Overview

The source material for this analysis process will be compiled, assembled program files. This section discussed the methods used to convert this binary format into a stream of instruction representations and other structural information that can be used as input to the later stages of the analysis process.

### 5.1.2 Executable files, object code, and disassembly

The four stages between high-level source code and execution are *compilation* which produces assembly code, *assembly* which produces relocatable machine code object files, *linking* which collects object files into executable files and resolves the symbols in the function calls, and *loading* where the executable file is loaded into a virtual address space ready for execution.

For all the reasons discussed in §1, this process needs to operate on the level closest to execution, however capturing the image of the virtual address space after loading is impractical. Producing a formally-verified simulation of the loader in the target system would be ideal but the development of such a system is beyond the scope of this work. Consequently, it is the executable file that is as close as is practical and that will be the source material for this analysis.

Executable files contain various information as well as the program itself — specifically information about the layout of the virtual address space that the program requires. To retain complete formality of the process it would be ideal if the disassembly of the executable into a processable form was a formal, traceable process. The development of this system is beyond the scope of this work, so it is necessary to depend on existing technology. The GNU objdump[2] program includes a disassembler, which converts executable files into a more readable representation of their contents. It is capable of accepting input files in a wide

variety of forms and for a large range of platforms. An example of the output from objdump is shown below, along with the original C program that was compiled.

```
int max(unsigned int x, unsigned int y) {
  if(x > y) {
    return x;
  } else {
    return y;
  }
}

int maxint(unsigned int *ints) {
  unsigned int result, *ptr;

  result = ints[0];
  ptr = (ints + 1);
  while(*ptr != 0) {
    result = max(*ptr, result);
    ptr++;
  }
  return result;
}
```

```
maxint:     file format elf32-i386

Disassembly of section .text:

080480d8 <max>:
 80480d8: 55                      push   %ebp
 80480d9: 89 e5                   mov    %esp,%ebp
 80480db: 8b 45 08                mov    0x8(%ebp),%eax
 80480de: 3b 45 0c                cmp    0xc(%ebp),%eax
 80480e1: 76 05                   jbe    80480e8 <max+0x10>
 80480e3: 8b 45 08                mov    0x8(%ebp),%eax
 80480e6: eb 03                   jmp    80480eb <max+0x13>
 80480e8: 8b 45 0c                mov    0xc(%ebp),%eax
 80480eb: 5d                      pop    %ebp
 80480ec: c3                      ret

080480ed <maxint>:
 80480ed: 55                      push   %ebp
 80480ee: 89 e5                   mov    %esp,%ebp
 80480f0: 83 ec 18                sub    $0x18,%esp
 80480f3: 8b 45 08                mov    0x8(%ebp),%eax
 80480f6: 8b 00                   mov    (%eax),%eax
```

```
80480f8: 89 45 f8                mov    %eax,-0x8(%ebp)
80480fb: 8b 45 08                mov    0x8(%ebp),%eax
80480fe: 83 c0 04                add    $0x4,%eax
8048101: 89 45 fc                mov    %eax,-0x4(%ebp)
8048104: eb 1b                   jmp    8048121 <maxint+0x34>
8048106: 8b 45 fc                mov    -0x4(%ebp),%eax
8048109: 8b 00                   mov    (%eax),%eax
804810b: 8b 55 f8                mov    -0x8(%ebp),%edx
804810e: 89 54 24 04             mov    %edx,0x4(%esp)
8048112: 89 04 24                mov    %eax,(%esp)
8048115: e8 be ff ff ff          call   80480d8 <max>
804811a: 89 45 f8                mov    %eax,-0x8(%ebp)
804811d: 83 45 fc 04             addl   $0x4,-0x4(%ebp)
8048121: 8b 45 fc                mov    -0x4(%ebp),%eax
8048124: 8b 00                   mov    (%eax),%eax
8048126: 85 c0                   test   %eax,%eax
8048128: 75 dc                   jne    8048106 <maxint+0x19>
804812a: 8b 45 f8                mov    -0x8(%ebp),%eax
804812d: c9                      leave
804812e: c3                      ret
```

This displays each instruction in the *text*[1] segment of the executable file and the address in the virtual address space at which the instruction will be loaded. Objdump also processes the *symbol table* of the executable file to add useful labels to some of the addresses — notably, the function names from the original C source file are displayed. This file was *not* compiled with explicit debugging information included. Unless the symbol table is deliberately stripped from executable files this level of useful symbols is present in all gcc compiled executables.

Most significant for this analysis process is the action of *disassembly* that objdump performs on the instructions. Each instruction is represented by a line of text containing the mnemonic and the parameters that are applied to it. Objdump presents its output for Intel executables in AT&T syntax, so that will be the standard used for the examples in the remainder of this document.

It is significant that the disassembly output above was produced from a compiled and linked executable file. Compilers can perform the compilation and assembly steps but stop before linking to produce *object files.* These intermediate files are used in large systems where many modules may be compiled separately and then linked together to form the executable. References between functions cannot be resolved until linking. References to functions in other

[1] An executable file contains all of the information necessary for the loader to produce the virtual memory image in which the program will run. The *text* segment conventionally contains the program code. Other segments such as *bss* have historical names, but contain pre-initialised stack and heap space that may be filled with initial values that were defined as part of the high level language program. These values are not considered in the examples presented here, but if their content was important then objdump can create output files that include all sections

modules are impossible to determine, but references within the object file also cannot be resolved as the load address of the target is not determined until the entire executable is available and the linker determines the layout in the virtual address space.

```
/home/ramsay/maxint.o:     file format elf32-i386


Disassembly of section .text:

00000000 <max>:
   0: 55                      push   %ebp
   1: 89 e5                   mov    %esp,%ebp
   3: 8b 45 08                mov    0x8(%ebp),%eax
   6: 3b 45 0c                cmp    0xc(%ebp),%eax
   9: 76 05                   jbe    10 <max+0x10>
   b: 8b 45 08                mov    0x8(%ebp),%eax
   e: eb 03                   jmp    13 <max+0x13>
  10: 8b 45 0c                mov    0xc(%ebp),%eax
  13: 5d                      pop    %ebp
  14: c3                      ret

00000015 <maxint>:
  15: 55                      push   %ebp
  16: 89 e5                   mov    %esp,%ebp
  18: 83 ec 18                sub    $0x18,%esp
  1b: 8b 45 08                mov    0x8(%ebp),%eax
  1e: 8b 00                   mov    (%eax),%eax
  20: 89 45 f8                mov    %eax,-0x8(%ebp)
  23: 8b 45 08                mov    0x8(%ebp),%eax
  26: 83 c0 04                add    $0x4,%eax
  29: 89 45 fc                mov    %eax,-0x4(%ebp)
  2c: eb 1b                   jmp    49 <maxint+0x34>
  2e: 8b 45 fc                mov    -0x4(%ebp),%eax
  31: 8b 00                   mov    (%eax),%eax
  33: 8b 55 f8                mov    -0x8(%ebp),%edx
  36: 89 54 24 04             mov    %edx,0x4(%esp)
  3a: 89 04 24                mov    %eax,(%esp)
  3d: e8 fc ff ff ff          call   3e <maxint+0x29>
  42: 89 45 f8                mov    %eax,-0x8(%ebp)
  45: 83 45 fc 04             addl   $0x4,-0x4(%ebp)
  49: 8b 45 fc                mov    -0x4(%ebp),%eax
  4c: 8b 00                   mov    (%eax),%eax
  4e: 85 c0                   test   %eax,%eax
  50: 75 dc                   jne    2e <maxint+0x19>
  52: 8b 45 f8                mov    -0x8(%ebp),%eax
```

```
  55: c9                    leave
  56: c3                    ret
```

This example shows the output of objdump if used to disassemble the max-int example compiled as an object file. Although the *jmp* instructions resolve correctly the *call* instruction at address $3d$ is erroneously disassembled to direct execution to address $3e$. This value is actually an offset into the relocation table of the object file, which lists all the references that must be resolved once the linker has assigned locations in the virtual address space.

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000003e R_386_PC32        max
```

The consequence of this is that the analysis process must operate on complete, linked executable files if it is to work directly on the output of the objdump disassembler. The analysis could read the relocation table and resolve the references but this would require far more processing of the files and a far more complex understanding of the executable file format for the target architecture of each system, so this was not implemented as part of this work.

### 5.1.3 Parsing

If objdump is used to disassemble a linked executable then its output contains all the information necessary to produce a model of the software's behaviour. Before the instructions can be instantiated in the formal model the output format must be parsed to extract the relevant information. The general layout of the output format is simple and well structured so producing a basic parser using a parser generator such as JavaCC[6] is straightforward, but some features of the instructions and their parameters deserve special consideration.

Objdump produces assembly language that is appropriate for the target architecture, including the language syntax. For Intel systems it uses the AT&T syntax, but for PowerPC it uses the standard PowerPC syntax, as in this example:

```
00000000 <max>:
   0:   94 21 ff e0     stwu    r1,-32(r1)
   4:   93 e1 00 1c     stw     r31,28(r1)
   8:   7c 3f 0b 78     mr      r31,r1
   c:   90 7f 00 08     stw     r3,8(r31)
  10:   90 9f 00 0c     stw     r4,12(r31)
  14:   81 3f 00 08     lwz     r9,8(r31)
  18:   80 1f 00 0c     lwz     r0,12(r31)
  1c:   7f 89 00 40     cmplw   cr7,r9,r0
  20:   40 9d 00 0c     ble-    cr7,2c <max+0x2c>
  24:   80 1f 00 08     lwz     r0,8(r31)
  28:   48 00 00 08     b       30 <max+0x30>
```

```
  2c:   80 1f 00 0c     lwz     r0,12(r31)
  30:   7c 03 03 78     mr      r3,r0
  34:   39 7f 00 20     addi    r11,r31,32
  38:   83 eb ff fc     lwz     r31,-4(r11)
  3c:   7d 61 5b 78     mr      r1,r11
  40:   4e 80 00 20     blr
```

The syntax for each architecture is sufficiently distinct that it requires a specific parser but the analysis process has been designed such that a suitable general semantic representation can be constructed that will be adequate for most mainstream processors. The parser attempts to retain not only information that is necessary to produce a semantically accurate formal model, but also information such as function names and instruction addresses that makes the formal model more readable and, critically, more tracable back to the original executable. Retaining function names will allow the inferred model to have a readable structure with model components retaining names appropriate to the component they represent in the original system. Retaining instruction addresses allows faults identified in the model to be traced back to particular sections of the original code, and so direct fault repair most efficiently.

Since the details of the operation of each instruction will be provided as an input to the analysis in the instruction set specification, it is sufficient to parse the disassembly output only far enough to provide values that can be textually substituted into the instruction set specification to produce the correct specification for an instruction instance. More detail on this substitution process is presented at §5.3.3, but the parameters that are given to most instructions on most modern processors fall into a small number of distinct categories, including:

- *literals* — exact values.
- *registers* — the identifier for a register whose contents should be used.
- *register indirect* — a combination of a register identifier and an offset, where the value to be used is found at the memory address produced by adding the offset value to the value contained in the register.

Both the Intel and PowerPC examples here show all of these classes of parameter. Intel registers have identifying names such as *eax* and *ecx*, while the registers in the PowerPC CPU are numbered from 0 to 31. The Intel AT&T syntax denotes register names with a prefix percent symbol, and literals with a prefix dollar symbol, as in this example:

```
18: 83 ec 18                sub    $0x18,%esp
```

which subtracts the literal value 24 (hexadecimal 0x18) from the content of the *esp* register (the stack pointer). The PowerPC architecture prefixes registers with the letter *r* and has no prefix for literal values.

For indirect addressing both syntaxes wrap the register identifier (including prefix) in brackets and then prefix the brackets with the offset value (without an identifying prefix), as in these examples:

```
 1b:  8b 45 08       mov    0x8(%ebp),%eax
```

```
  c:  90 7f 00 08     stw    r3,8(r31)
```

The final parameter type of significance is *branch targets*. Branch instructions are analysed differently from sequential instructions, as is discussed further in §5.2. The objdump disassembler produces the virtual address location of the target of the branch instruction, along with a helpful description based on an offset from a function name from the symbol table. Both Intel and PowerPC syntaxes follow a similar convention:

```
   e: eb 03           jmp    13 <max+0x13>
```

```
  28:   48 00 00 08     b      30 <max+0x30>
```

Since the analysis process parsing will retain the virtual addresses of all instructions it is only this component that is necessary to produce a correct model of the branch behaviour.

### 5.1.4 Summary

This section has covered the process of disassembling an executable file into a structured set of instructions, and the process of parsing adequate information about each instruction to support the formal model inference. The parsed instruction representations will be used as input to the subsequent process stages where the branch points will be identified to produce a control flow graph, and the the instructions will be instantiated with formal models from the provided instruction set.

## 5.2 Branch Identification

### 5.2.1 Overview

The disassembly from §5.1 produced a set of instructions and some structural information such as function names and locations. This section presents the process by which the branch instructions are identified. Using the specifications of the branch instruction set the assembly language instructions are partitioned into sequential blocks with the branch instructions forming nodes in a control flow graph.

### 5.2.2 Branch instructions

The disassembly process described in §5.1 converted an executable file into a set of functions, each containing a sequence of instructions. At the most general level these instructions can be divided into two classes: *sequential instructions*, that affect the state of the system and then allow execution to proceed to the

next instruction in the function, and *branch instructions*, that direct execution to proceed from a different place in the program, possibly only when some condition is met. The analysis phase presented in this section will identify the branch instructions and their target locations in the program. This will allow the program to be segmented into *sequential block* — contiguous sequences of sequential instructions where execution is certain to proceed through the complete sequence.

The two inputs to this process are:

- A function from the assembly language output of the disassembler
- A set of branch instructions with enough information to identify the instruction instances and the potential targets of the branch.

### 5.2.3 Branch template format

For most branch instructions only two pieces of information are necessary: the mnemonic and identification of which parameter forms the target address of the jump. A LaTeX-style representation of the *jmp* instruction, specifying the mnemonic and that the first argument contains the target of the branch, is written this way:

```
\begin{binst}{jmp}

Target: FIRST

\end{binst}
```

Since the *jmp* instruction is an *unconditional* branch this is all the information necessary to process instances in the assembly code. For conditional branch instructions it is necessary to specify the *OnBranch* and *NoBranch* prefix schema, as discussed in §3.3.3. The *jne* instruction from the x86 instruction set, which exhibits conditional branching behaviour, is specified in this form:

```
\begin{binst}{jne}

Target: FIRST

\begin{schema}{branch}
\Xi System
\where
zf = 1
\end{schema}

\begin{schema}{nobranch}
\Xi System
\where
```

```
zf \neq 1
\end{schema}
\end{binst}
```

This analysis of branch instructions has assumed that the branch target will be within the same function. Calls to other functions in the system are generally handled by different processor instructions, which this process will refer to as *call instructions*. These often have additional system effects such as pushing return addresses onto the stack. They transfer execution to the beginning of the target function, and are matched by *return* instructions that transfer execution back to the next instruction in the calling function. In principle it is possible to write assembly language programs which use the regular jump instructions to branch out of the function and into another — possibly at an arbitrary point in the other function — but it is unlikely that any compiler would compile a high level language to use this type of construction. The attitude that this work takes to such constructions mirrors the MISRA-C attitude: that the presence of such constructions in high-integrity code is itself hazardous. If such code must be analysed then it would have to be analysed as s single, large function, but it will be inherently difficult to comprehend and is best discouraged by the overarching software development management.

```
\begin{callinst}{call}

Target: FIRST

\end{callinst}

\begin{returninst}{ret}
\end{returninst}
```

The *call* instruction shown above is unconditional, and the *ret* instruction takes no parameters, so this is enough information to process instances of these two instructions. This model of the *ret* instruction produces no effect in the system other than the return of execution. The correct operation of the *ret* instruction requires that the return address is at the top of the stack. This analysis process opts not to model this behaviour specifically as it would complicate the formal model of function calls and returns that is created in §5.5.3. For the model to be correct it is required that the return address is not overwritten during the operation of the function (the stack overflow attack). This requirement should be confirmed as part of the verification of requirements on the model.

### 5.2.4 Example

The *maxint* function from the example in §5.1 illustrates all of these features.

```
080480ed <maxint>:
 80480ed: 55                    push   %ebp
```

```
 80480ee: 89 e5                 mov    %esp,%ebp
 80480f0: 83 ec 18              sub    $0x18,%esp
 80480f3: 8b 45 08              mov    0x8(%ebp),%eax
 80480f6: 8b 00                 mov    (%eax),%eax
 80480f8: 89 45 f8              mov    %eax,-0x8(%ebp)
 80480fb: 8b 45 08              mov    0x8(%ebp),%eax
 80480fe: 83 c0 04              add    $0x4,%eax
 8048101: 89 45 fc              mov    %eax,-0x4(%ebp)
 8048104: eb 1b                 jmp    8048121 <maxint+0x34>
 8048106: 8b 45 fc              mov    -0x4(%ebp),%eax
 8048109: 8b 00                 mov    (%eax),%eax
 804810b: 8b 55 f8              mov    -0x8(%ebp),%edx
 804810e: 89 54 24 04           mov    %edx,0x4(%esp)
 8048112: 89 04 24              mov    %eax,(%esp)
 8048115: e8 be ff ff ff        call   80480d8 <max>
 804811a: 89 45 f8              mov    %eax,-0x8(%ebp)
 804811d: 83 45 fc 04           addl   $0x4,-0x4(%ebp)
 8048121: 8b 45 fc              mov    -0x4(%ebp),%eax
 8048124: 8b 00                 mov    (%eax),%eax
 8048126: 85 c0                 test   %eax,%eax
 8048128: 75 dc                 jne    8048106 <maxint+0x19>
 804812a: 8b 45 f8              mov    -0x8(%ebp),%eax
 804812d: c9                    leave
 804812e: c3                    ret
```

There are four branch instructions in this function (including the *ret* at the end). The following shows the function broken around the branch instructions.

```
080480ed <maxint>:
 80480ed: 55                    push   %ebp
 80480ee: 89 e5                 mov    %esp,%ebp
 80480f0: 83 ec 18              sub    $0x18,%esp
 80480f3: 8b 45 08              mov    0x8(%ebp),%eax
 80480f6: 8b 00                 mov    (%eax),%eax
 80480f8: 89 45 f8              mov    %eax,-0x8(%ebp)
 80480fb: 8b 45 08              mov    0x8(%ebp),%eax
 80480fe: 83 c0 04              add    $0x4,%eax
 8048101: 89 45 fc              mov    %eax,-0x4(%ebp)
----------------------------------------------------------------------
 8048104: eb 1b                 jmp    8048121 <maxint+0x34>
----------------------------------------------------------------------
 8048106: 8b 45 fc              mov    -0x4(%ebp),%eax
 8048109: 8b 00                 mov    (%eax),%eax
 804810b: 8b 55 f8              mov    -0x8(%ebp),%edx
 804810e: 89 54 24 04           mov    %edx,0x4(%esp)
 8048112: 89 04 24              mov    %eax,(%esp)
```

```
-----------------------------------------------------------------------
 8048115: e8 be ff ff ff        call   80480d8 <max>
-----------------------------------------------------------------------
 804811a: 89 45 f8              mov    %eax,-0x8(%ebp)
 804811d: 83 45 fc 04           addl   $0x4,-0x4(%ebp)
 8048121: 8b 45 fc              mov    -0x4(%ebp),%eax
 8048124: 8b 00                 mov    (%eax),%eax
 8048126: 85 c0                 test   %eax,%eax
-----------------------------------------------------------------------
 8048128: 75 dc                 jne    8048106 <maxint+0x19>
-----------------------------------------------------------------------
 804812a: 8b 45 f8              mov    -0x8(%ebp),%eax
 804812d: c9                    leave
-----------------------------------------------------------------------
 804812e: c3                    ret
-----------------------------------------------------------------------
```

This forms four sequential blocks. The call instruction targets the *max* function, which will be processed separately; the *ret* instruction is identified as a function return, so will invoke the standard function return mechanism (*Leave* from the *Function* class).

Of particular importance is the *jmp* instruction at address 8048104 that causes program execution to branch to address 8048121. This address is currently in the middle of a sequential block. This must be represented by splitting this block at this point, as shown here:

```
08048195 <maxint>:

[...]
----------------------------------------------------------------
 8048104: eb 1b                 jmp    8048121 <maxint+0x34>
----------------------------------------------------------------
[...]
----------------------------------------------------------------
 804811a: 89 45 f8              mov    %eax,-0x8(%ebp)
 804811d: 83 45 fc 04           addl   $0x4,-0x4(%ebp)
----------------------------------------------------------------
<entry point from 8048104>
----------------------------------------------------------------
 8048121: 8b 45 fc              mov    -0x4(%ebp),%eax
 8048124: 8b 00                 mov    (%eax),%eax
 8048126: 85 c0                 test   %eax,%eax
[...]
```

This produces a block from $804811a$ to $804811d$ that is not followed by an explicit branch instruction but that will continue on into the new block starting

at 8048121. The analysis process inserts a trivial, unconditional *straight through* branch at this point.

The result of this is the control flow graph presented in Figure 5.1.

The branch identification process, like all the analysis phases that follow, is independent of the analysis of other functions. This allows all of the functions in a program to be processed concurrently in systems capable of making use of concurrent execution.

### 5.2.5 Summary

This section has detailed the process by which the sequences of instructions produced from the disassembler can be segmented into sequential blocks, and these blocks arranged into a control flow graph, with the branch instructions forming the nodes. The subsequent phases of the analysis process will operate on this graph and will begin to instantiate the formal model of the program's behaviour.

## 5.3 Formal Instantiation

### 5.3.1 Overview

The sequential blocks produced by the branch identification stage can be converted into formal representations of the behaviour of their constituent instructions. This section presents the process by which the template specifications described in §3.3.2 are instantiated into Z operation schema. These are then sequentially composed to form a formal representation of the sequential block in Z.

### 5.3.2 Template specifications

To produce a formal representation of a block of assembly code it is necessary to convert each of the instructions into a Z operation schema. To do this the analysis process user must provide a template specification that defines the behaviour of each instruction in the instruction set. These template specifications are parameterised in the same way as the instruction set descriptions from the processor manufacturer. As an example, the *sub* instruction is described thus:

```
Subtracts the second operand (source operand) from the first
operand (destination operand) and stores the result in the
destination operand. the destination operand can be a register
or a memory location; the source operand can be an immediate,
register, or memory location. [...]

The SUB instruction performs integer subtraction. It
evaluates the result for both signed and unsigned integer
operands and sets the OF and CF flags to indicate an overflow
in the signed or unsigned result, respectively. The SF flag
```

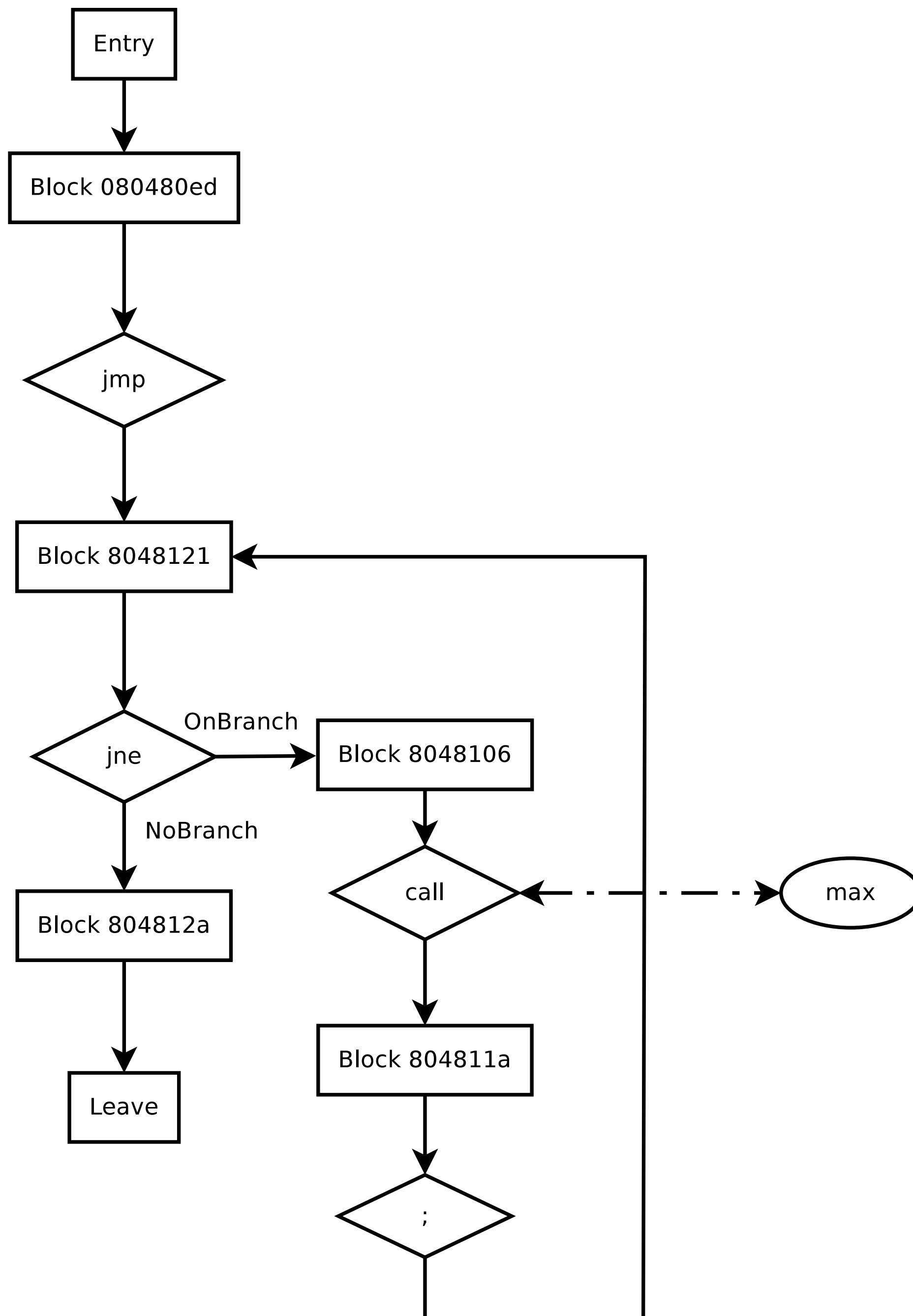



Figure 5.1: The control flow graph of the *maxint* function

```
indicates the sign of the signed result.
```

The supplied sequential instruction set specification should consist of a set of *template schema* — Z operation schema with names that match the instruction mnemonics. In some cases the same mnemonic can represent several different underlying op codes depending on the type of the arguments. The operation schema therefore require a type signature. This is provided as a subscript on the end of the schema name with the pattern $TYPE\#NAME$. The types currently supported are $LIT$ for literal values, $REG$ for register names, and $REGIND$ for register-indirect memory access (where a value in a register is offset by some literal amount before being dereferenced). The parser component will identify the type of the parameters present in the disassembly, and the analysis process will use this information to select the appropriate operation schema template. The name component of the signature should be a text string that will be replaced throughout the schema with the parsed value of a particular instance of the argument. This replacement is done as a naive string match, so it is important that the string does not appear elsewhere in the schema (even as a substring).

$$
\begin{array}{l}
sub_{LIT\#VAL,REG\#TGT} \\
\hline
\Delta\ System \\
\hline
registers' = registers \oplus \{TGT \mapsto registers(TGT) - VAL\} \\
memory' = memory \\
(zflag' = 1) \Leftrightarrow (registers'(TGT) = 0) \\
\hline
\end{array}
$$

To allow for maximum scalability these template specifications should be instantiated with the minimum possible semantic knowledge. The parameter names will be simply textually substituted without any processing of the statements into which it is inserted. This requires some consideration when designing the system specification and the instruction set operations, but it does not need to be overly constrictive. It does create some interesting results in the instantiation: the *sub* example could be given a negative literal, resulting in a double negation in the instantiated operation:

$$registers' = registers \oplus \{eax \mapsto registers(esp) - (-10)\}$$

This is semantically correct but can cause problems with some automatic Z interpreters — including the Z2SAL system used in §7. It is easy to perform a text search and replace on the output if a correction is required.

The $REGIND$ type includes two components: a register and an offset. In the type signature the replacement names for these are separated with a comma:

$$
\begin{array}{l}
mov_{REG\#SRC,REGIND\#TGT;\,OFF} \\
\hline
\Delta\ System \\
\hline
memory' = memory \oplus \{(registers(TGT) + OFF) \mapsto registers(SRC)\}
\end{array}
$$

$$registers' = registers$$

This can present another arithmetic problem if negative offset values are used, but this can again be corrected by text search and replace for those interpreters that fail to handle it correctly.

### 5.3.3 Instantiation

The *maxint* example from §5.2.4 contains an instance of the *sub* instruction in $Block_{80480ed}$. This has the arguments $\langle literal \rangle 0x10$, $\langle register \rangle esp$. The template schema for the *sub* instruction presented above matches both mnemonic and type signature, so can be instantiated to produce in this way:

```
80480ed <maxint>:
[...]
 80480f0: 83 ec 18             sub    $0x18,%esp <<-------
[...]
```

$$sub_{80480f0}$$
$$\Delta\ System$$
$$registers' = registers \oplus \{esp \mapsto registers(esp) - 24\}$$
$$memory' = memory$$
$$zflag' = 1 \Leftrightarrow registers'(esp) = 0$$

Since this process requires only text matching of the mnemonic and type signature, and then text substitution of the parameters it is easily scalable to large numbers of instructions. This process can be repeated for each instruction in the sequential block. Once instantiated the resultant Z operation schema can be sequentially composed to produce a valid (but not minimal) model of the behaviour of the sequential block.

When the instruction template is instantiated the subscript containing the type signature is replaced with a subscript that contains the virtual address of the instruction being instantiated. In this way the traceability between the formal model and the original executable file is maintained. Throughout the rest of the analysis process the formal model components will retain the virtual addresses of the instructions whose behaviour they model. The consequence is that a property violation identified in the formal model can be traced back to at least the sequential block that caused it. This traceability is a key objective of this work.

### 5.3.4 Summary

This section has described the process by which the sequential blocks of the control flow graph can be converted into formal models in Z that represent the

sequence of state changes that the sequential instructions produce in the system. These sequences are modeled by a chain of sequentially composed Z operation schema. The next chapter will discuss a process to compress these chains into a single Z operation schema, or at least a much shorter chain of operations to model the same sequential block. The branch instructions that form the control flow nodes will be considered in §5.5.

## 5.4 Simplification

### 5.4.1 Overview

This section presents the application of the theory from §4 to compress the long sequential compositions produced by the previous analysis stage. This should yield much more comprehensible Z representations of the behaviour of the sequential blocks without compromising either the scalability of the process, or the traceability afforded by the naming conventions in §5.3.

### 5.4.2 Sequential block compression

The result of the *formal instantiation* process described in §5.3 is a series of sequential blocks that are modeled as long chains of sequentially composed Z operation schema representing each instruction. The size of these chains can quickly become unmanageable. The nine line *maxint.c* program produced a 35 line assembly file with only 7 branch instructions, and a 54 line PowerPC assembly file with 7 branch instructions. The PowerPC file included a sequential block with 13 unbroken instructions, and this is a simple function. Some technique is needed to simplify these sequential blocks if the objective is readable formal models.

The balance of readability, accuracy, and automation has to be carefully managed. In principle, if program interruption is to be ignored, then the sequential blocks could be resolved to single Z operation schema but to do this requires some considerable formal analysis of the semantics of the operations which would be prohibitively difficult as the program size increased. This could be engineered if readability was the overriding objective. This may be the case if a stable, existing program is to be reverse engineered to determine its behaviour. In this case the long execution time of an analysis task would be acceptable for the end product, but if the analysis forms part of the feed-back workflow of a development process then timely verification analysis must take precedence. In the latter case a partial composition, where a sequential block of 13 instructions is reduced to perhaps 5 sequentially composed schema in a short time may be adequate. A CSP-OZ proof assistant tool should be able to process a 5 instruction sequential composition and still prove useful properties.

### 5.4.3 Algorithm

This level of concatenation is possible for limited computation expense using the techniques outlined in §4. The process operates by comparing two sequentially composed schemas and determining whether their semantics is altered by simply textually concatenating their invariants into one single operation. Since this process is text-based with only minimal parsing of the Z semantics it can be performed very quickly on large blocks of instructions. Additionally, since the compression of one sequential block has no impact on the behaviour of another it is possible to perform the simplification process on all the sequential blocks in a function concurrently. With the approximate ratio of five sequential instructions to one branch instruction there is considerable scope for concurrent execution to be used efficiently on multi-processor or distributed systems.

Analysis begins with the first pair of operations in the sequence and attempts to compress them into one. Using the rules of separation in Z, the two operations are compared and a separation proof attempted. The separation rules were designed to be performed with limited semantic knowledge, and to be satisfied in those conditions where the two schema can be compressed by simply placing all the variable definitions and invariant statements in one Z operation schema. If the separation test is successful then the two schema are removed from the sequence and replaced with the new operation schema. The process continues with the new schema and the next schema in the sequence. If the separation test fails then the process attempts to merge the second and third schema, and so it continues through the entire sequential block, compressing as many schema as possible. The design of the separation test rules is such that a second pass of the compressed sequential composition is not necessary.

In cases where a sequence does not meet any of the criteria for separation, a human interpreter or a sufficiently well informed proof tool could resolve them into a single schema but they will remain sequentially composed in this implementation. There is a necessary choice between producing the most succinct model theoretically possible and producing a model entirely automatically. Since it will be possible to apply automatic tools to the analysis of the model (for example Z2SAL, see §7) it can be argued that the simplification does not need to be complete if that would require excessive human effort.

### 5.4.4 Summary

The simplification stage takes the Z formal model representing the sequential sections of the executable and compresses the long sequential compositions into shorter Z operation representations or the state effects. The balance of compression with computational speed can be varied to suit different analysis process goals, but the ideas from §4 allow for a moderate reduction in the number of composed operations for a limited computational effort.

The control flow graph is updated with the new, compressed sequential blocks. This will form the input to the final stage, where the control flow elements are rendered to CSP representations and the other structural and

debugging components used to form a complete CSP-OZ specification of the system.

## 5.5 Program Encapsulation

### 5.5.1 Overview

Having identified the branch instructions and sequential blocks, instantiated the formal specifications of the sequential components, and simplified the sequential blocks, the final element of the analysis process is to compose the sequential blocks into a CSP-OZ class that represents the function. The branch instructions and the *straight through* pseudo-instructions must be instantiated to form the CSP components. The *OnBranch* and *NoBranch* schema from the conditional branch instructions are instantiated as Z operation schema where necessary. Function calls and returns must be instantiated with suitable models, as must the entry and exit of the functions.

### 5.5.2 Internal branch instructions

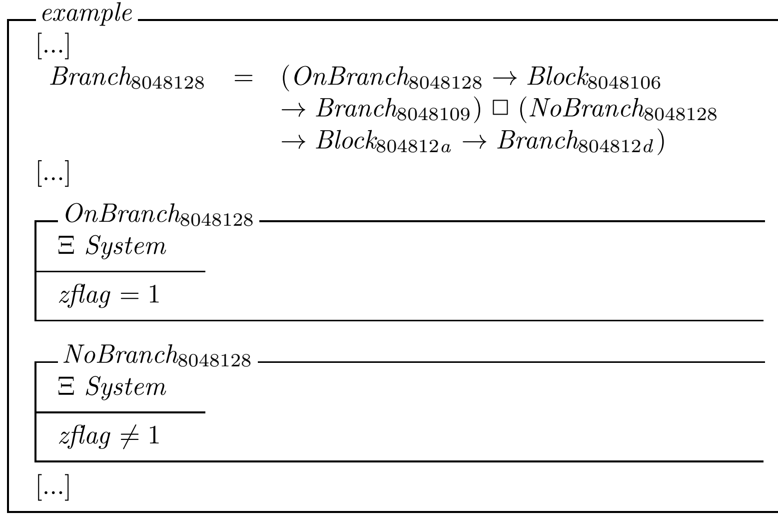
Each unconditional branch instruction (including the *straight through* pseudo-instructions) has a defined target. Similarly, the analysis process records the branch that follows each sequential block as part of the internal model of the sequential block. With this information is is simple to convert unconditional branches and their target blocks into CSP statements. The branch instruction

```
8048104: eb 1b                 jmp    8048121 <maxint+0x34>
```

will always cause execution to transfer to virtual address $0x8048121$. The control flow graph shows that the block beginning at $0x8048121$ ends with the branch instruction at address $0x8048128$. The branch instructions will all be represented by CSP processes named *Branch* with a subscript containing the virtual address of the instruction they represent. Knowing that the branch at address $0x8048128$ will be instantiated elsewhere and named consistently allows this instruction to be instantiated as the CSP process that simply executed the Z operation representing the block from address $0x8048121$ and then evolves to $Branch_{0x8048128}$

$$Branch_{0x8048104} = Block_{0x8048121} \rightarrow Branch_{0x8048128}$$

The *jne* instruction at address $0x8048128$ is a conditional branch instruction. As is discussed in §3.3.3, this will be modeled by instantiating each possible target sequential block as a CSP arrow as before, then prefixing this arrow with a Z operation that serves to constrain the execution of the possible paths according to the conditions of the branch instruction. Finally, the two paths are conjoined with a CSP external choice operator — although the two preconditions should be mutually exclusive, which will collapse the choice for a particular program run to only the path whose prefix is satisfied by the current system state.



Ideally all of the sequential blocks will have been converted to single Z operation schema that represent the block's behaviour. In the event that this hasn't been possible it is necessary to encode their sequential behaviour. The simplest way to do this is with a series of *straight through* branch instructions. These are treated as unconditional branch instructions that execute the next sequential block. They are added to the CSP section in the same way as any unconditional branch, and named for the block that they prefix.

$$Branch_{8048121} = Block_{8048121} \rightarrow Branch_{8048124}$$

### 5.5.3 Function calls

As described in §3.3.3, function calls are modeled by executing the function in parallel, passing the system state using schema promotion, and then synchronising on the communication. The sequence $Call \rightarrow Return$ is common to all function calls, from there the remainder of the process continues exactly as with unconditional branches: the next block is executed, and the process evolves to the next branch instruction.

```
8048115: e8 be ff ff ff         call   80480d8 <max>
```

becomes

$$\begin{aligned} Branch_{8048115} &= (OnBranch_{8048115} \rightarrow Call \rightarrow Return \\ &\quad \rightarrow Block_{804811a} \rightarrow Branch_{804811d}) \mid\mid max \end{aligned}$$

### 5.5.4 Class definition

CSP-OZ classes require a `main` process to begin execution. This will begin with the *Entry* operation that will receive the *System* state schema from a parallel call operation. Then the process continues with the first block and the first branch as any other branch.

All that remains is to collect these components into a CSP-OZ class, which is named according to the function name extracted by the disassembler. This produces a formal model where each function in the analysed system is contained in a CSP-OZ class.

### 5.5.5 Summary

This final stage of the analysis process completes the model inference with two components: the instantiation of the control flow structure in CSP, and the collection of components into Object Z classes. This produces a complete representation of the behaviour of the system as a series of CSP-OZ classes that represent the various functions from the source executable. This model can now be used for verification of system properties using any verification technique that is defined over CSP-OZ. Examples of property verification techniques are presented in §7 and §8.

# Chapter 6

# The Spurinna implementation

## 6.1 Overview

The analysis workflow described in the preceding sections has been implemented as a research prototype named Spurinna[1]. This chapter describes the Spurinna implementation, giving a description of its structure and describing the implementation of the various components of the analysis process.

## 6.2 Input files

The Spurinna implementation includes several parsers for the different inputs. The parsers are all implemented using the JavaCC[6] parser generator.

Both the sequential and branch instruction sets are parsed in a LaTeX Z style with some required structures. The sequential instructions are represented as LaTeX Z schema but with the requirement that their names are the mnemonics of the assembly language, and that the names contain a subscript that details their type signature. More discussion of instruction type signatures can be found in §3.3.

The input version of this *sub* template for a literal parameter and register target is written:

```
\begin{schema}{sub_{LIT#VAL,REG#TGT}}
\Delta~System \\
\where
registers' = registers \oplus {TGT \mapsto registers(TGT) - VAL}
```

[1] Titus Vestricius Spurinna is believed to be the name of the Roman haruspex who warned Caesar to "beware the ides of march". Since the practise of an haruspex was to cut open animals and divine the future by inspecting their entrails this seemed an appropriate name for a disassembly analysis system

```
memory' = memory
(zflag' = 1) \iff (registers'(TGT) = 0)
\end{schema}
```

The branch instruction input format is similar but collects the various branch prefix schema into a LaTeX environment called *binst*. Detailed discussion of branch instruction specifications can be found in §3.3.3. The collection also contains a `Target` attribute that specifies which instruction parameter should contain the branch target address. In the Intel instruction set there is only ever one parameter and this is always the branch target, so they are all defined as `FIRST`. Currently the implementation only supports `FIRST`, `SECOND`, or `THIRD`. The branch prefix schema must be named `onbranch` and `nobranch`.

```
\begin{binst}{jbe}

Target: FIRST

\begin{schema}{onbranch}
\Xi~System
\where
zflag = 1 \lor sflag = 1
\end{schema}

\begin{schema}{nobranch}
\Xi~System
\where
zflag \neq 1 \land sflag \neq 1
\end{schema}
\end{binst}
```

The other source of input to the analysis process is the disassembled executable file. The Spurinna implementation expects to be provided with a text file containing the output of a GNU objdump disassembly. The system contains a parser switcher that reads the beginning of the file to determine the target architecture and to activate the appropriate assembly language parser. The top of the file should contain a line of this form:

```
maxint:     file format elf32-i386
```

This is produced by objdump from the target information in the executable file header. Currently the Spurinna implementation contains a parser for i386, and an untested parser for PowerPC, but the Java class structure contains suitable superclasses to allow rapid development for any other formats.

The remainder of the file should be the output of the objdump disassembly of the text segment. This expects functions to begin with a line containing the start address and the function name.

```
080480d8 <max>:
```

As discussed in §3.2, these function name symbols exist even if debugging was not turned on in the compiler. They exists unless they have been explicitly stripped. Between function name lines should be a sequence of instructions with a pattern that depends on the target platform. Intel instruction lines contain a virtual address, the hexadecimal representation of the binary instruction, the instruction mnemonic, and the parameters.

```
80480f0:       83 ec 18                sub    $0x18,%esp
```

This stage of the system makes no attempt to understand the semantics of the instructions. It parses the lines and created *ASMInstruction* objects that have an address and a mnemonic, and a collection of *Argument* objects for the parameters. The *Argument* class contains three subclasses: *IntLitArgument*, *RegArgument*, and *RegIndirectArgument* for the three classes of parameter that are modeled. The register identifiers are stored as strings and not interpreted further. It is required that the instruction specifications be written so that these strings can be substituted in without modification, hence the BNF type containing the register names as atoms that is used in §3.3.1. Literals other than integer are not considered in this implementation. Where floating point numbers are used they are generally split up into integers representing sign, exponent, and mantissa. String literals will handled by pointers, and characters by their numeric representation.

The collection of *ASMInstruction* objects is collected into an *ASMFunction* object, which contains the name, and start and end addresses of the function. These are then collected into an *ASMFile* object that also contains the name extracted from the first line of the file. This object is then the target of the subsequent analysis phases.

## 6.3 Analysis stages

### 6.3.1 Architecture

All of the analysis phases are child classes of the *ProcessStage* abstract class. This contains the structure needed to break the input down into several independent tasks, that can be represented by subclasses of *ProcessTask*. The subclasses of *ProcessStage* must implement the *makeNewComponent* method that can be given on of the *ProcessTask* objects and returns a subclass of *ProcessComponent*. This sets up a collection of worker objects, where *ProcessComponent* implements the Java interface *Runnable*. The *ProcessStage* class then contains the mechanics necessary to administer running many *ProcessComponent* threads in parallel. It refers to the *Configuration* singleton class to load the number of concurrent threads to execute and then begins to execute that many *ProcessComponent* objects. It monitors their progress and updates a public *percentComplete* value, along with a public *done* Boolean variable.

This collects all of the concurrent processing mechanics into these abstract classes. Each phase of the analysis process is then implemented by a set of

subclasses of these. The progress of the stages is controlled via the interface and some mechanics in the interface classes that prevents the phases being performed out of order and connects the output objects from one phase to the task lists of the next phase.

### 6.3.2 Branch identification

The branch identification phase is described in detail in §5.2. The implementation uses the *ASMFunction* objects contained in the parsed *ASMFile* object as the process tasks. Each *BranchIdentificationComponent* object operates on one function. Due to the way internal and external branch instructions are modeled there is nothing preventing different functions being analysed separately.

The collection of instructions is processed by searching for each instruction's mnemonic in the list of mnemonics available from the *BranchInstructionSpec* object produced by parsing the branch instruction set specification file. Where an instruction matches a branch mnemonic it is removed from the instruction listing and its target identified using the information from the *ASMInstruction* object and the *BranchInst* object that represents the parsed component of the branch instruction set specification. In the case of local branches this may partition another sequential block, as described in §5.2.3.

The process creates a *FunctionGraph* object. This contains *CodeBlock* objects that contain the sequential blocks of *ASMInstruction* objects, and *BranchNode* objects representing the branch instructions. The latter retain the information from the branch instruction specification, including the prefix schema.

```
TERMINATED
File
◀ Sequential Instruction Set | Branch Instruction Set | Branch Identification | Formal Instantiation | Simplification | Program Encapsulation
Start   Re-run
max:
Entry points:
        0x80480d8
Sequential Blocks:
        ====== 80480d8 ======
        0x80480d8:      push    ebp
        0x80480d9:      mov     esp, ebp
        0x80480db:      mov     ebp + 8, eax
        0x80480de:      cmp     ebp + 12, eax
        ====== 80480de ======
        >>> 0x80480e1: jbe ? 0x80480e8 : 0x80480e3 >>>

        ====== 80480e3 ======
        0x80480e3:      mov     ebp + 8, eax
        ====== 80480e3 ======
        >>> 0x80480e6: jmp ? 0x80480eb : 0x80480eb >>>

        ====== 80480e8 ======
        0x80480e8:      mov     ebp + 12, eax
        ====== 80480e8 ======
        >>> 0x80480eb: <straight through> ? 0x80480eb : 0x80480eb >>>

        ====== 80480eb ======
        0x80480eb:      pop     ebp
        ====== 80480eb ======
        >>> 0x80480ec: ret <return> >>>

Branch points:
        0x80480e1: jbe ? 0x80480e8 : 0x80480e3
        0x80480e6: jmp ? 0x80480eb : 0x80480eb
        0x80480eb: <straight through> ? 0x80480eb : 0x80480eb
        0x80480ec: ret <return>

maxint:
Entry points:
```

### 6.3.3 Formal instantiation

The formal instantiation stage has two layers. The *FunctionGraph* objects from the branch identification stage form the process tasks for the top layer. Since the instantiation of each code block in the graph is independent there is a further process stage created for each graph. In this child stage the process tasks are the *CodeBlock* objects from the graph. This allows each multiple blocks to be processed concurrently, as well as multiple graphs.

The process of instantiating a *CodeBlock* object requires iterating through the *ASMInstruction* objects and matching the mnemonics to elements of the sequential instruction set specification. Once a matching mnemonic is found a further check must be performed to match the type signature from the subscript of the specification to the types of the parameters in the *ASMInstruction*.

Once the matching template is found it is instantiated. A clone of the *ZSchema* object is made and then the identifiers from the type signature subscript are textually replaced throughout the body of the schema with the values from the parameters in the *ASMInstruction* object. Finally the subscript is changed to the virtual address of the instruction.

Each schema is added to a *SequentialBlock* object. The higher layer process waits for completion of all of its children and then produces a *FormalFunctionGraph* object using the *BranchNode* objects from the *FunctionGraph* and these *SequentialBlock* objects.

```
TERMINATED
File
◀ Sequential Instruction Set | Branch Instruction Set | Branch Identification | Formal Instantiation | Simplification | Program Encapsulation
Start   Re-run

max:
Entry points:
        0x80480d8
Sequential Blocks:
======== 80480d8 ========
--- push_{80480d8} ---------------------
| \Delta System
|------------------------------
| registers' = registers \oplus {esp \mapsto registers(esp) - 4}
| memory' = memory \oplus {registers'(esp) \mapsto registers(ebp)}
----------------------------------------

    ;

--- mov_{80480d9} ----------------------
| \Delta System
|------------------------------
| registers' = registers \oplus {ebp \mapsto registers(esp)}
| memory' = memory
----------------------------------------

    ;

--- mov_{80480db} ----------------------
| \Delta System
|------------------------------
| registers' = registers \oplus {eax \mapsto memory(registers(ebp) + 8)}
| memory' = memory
----------------------------------------

    ;

--- cmp_{80480de} ----------------------
| \Delta System
|------------------------------
```

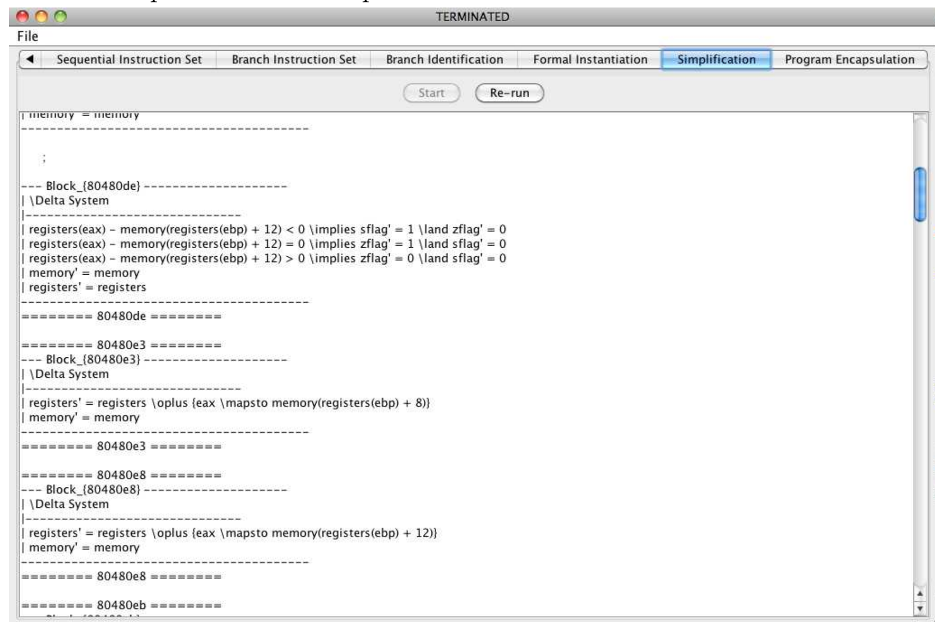
### 6.3.4 Simplification

The simplification stage is layered in a similar way to the formal instantiation stage. Again, the design of the simplification process is such that each block can be considered independently, allowing for maximum use of concurrent processing resources.

Simplification proceeds by processing the *ZSchema* objects from a *SequentialBlock* object and attempting to combine them. Using the rules from §4 it seeks schema that can be simply concatenated. Where the condition is met the two schema are concatenated and the result renamed as *Block* with a subscript containing the first address.

The simplification stage is the least well implemented in the current version of Spurinna. The *Total Separation* and *Effective Separation* conditions from §4 are implemented, but the *Functional Separation* condition is not. As such it is not as effective as it could be on the examples presented in §7. The *BlockSimplification* class is also the place where additional simplification using external tools with an understanding of Z semantics could be added.

The product of this stage is the same *FormalFunctionGraph* but, hopefully, with shorter sequential block components.

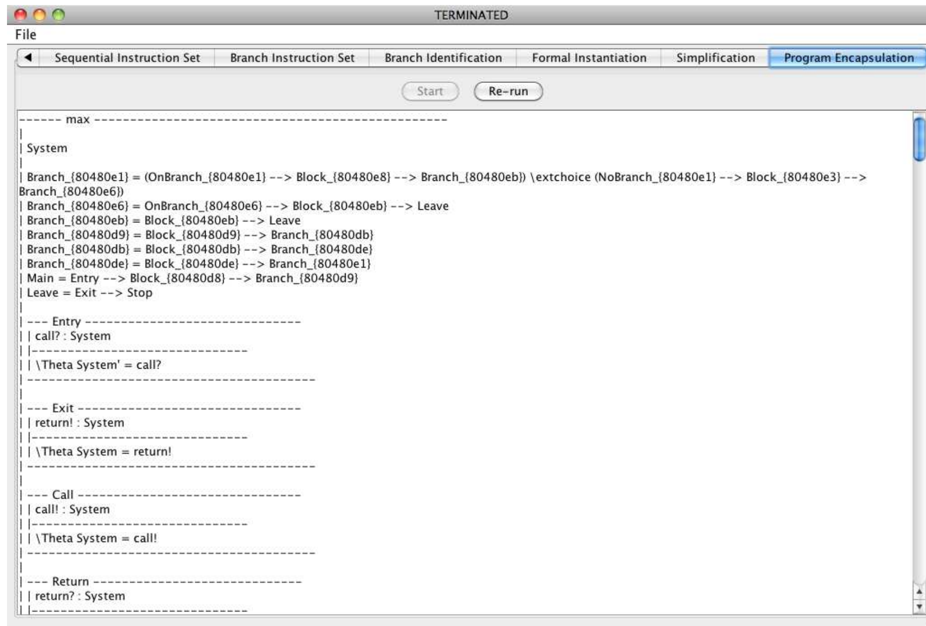



### 6.3.5 Program encapsulation

The final stage implements program encapsulation as described in §5.5. This converts a *FormalFunctionGraph* object into a *CSPOZClass* object. It iterates through the *BranchNode* objects in the graph and performs the necessary instantiation into *CSPProcess* objects. Additionally, it clones the *OnBranch* and

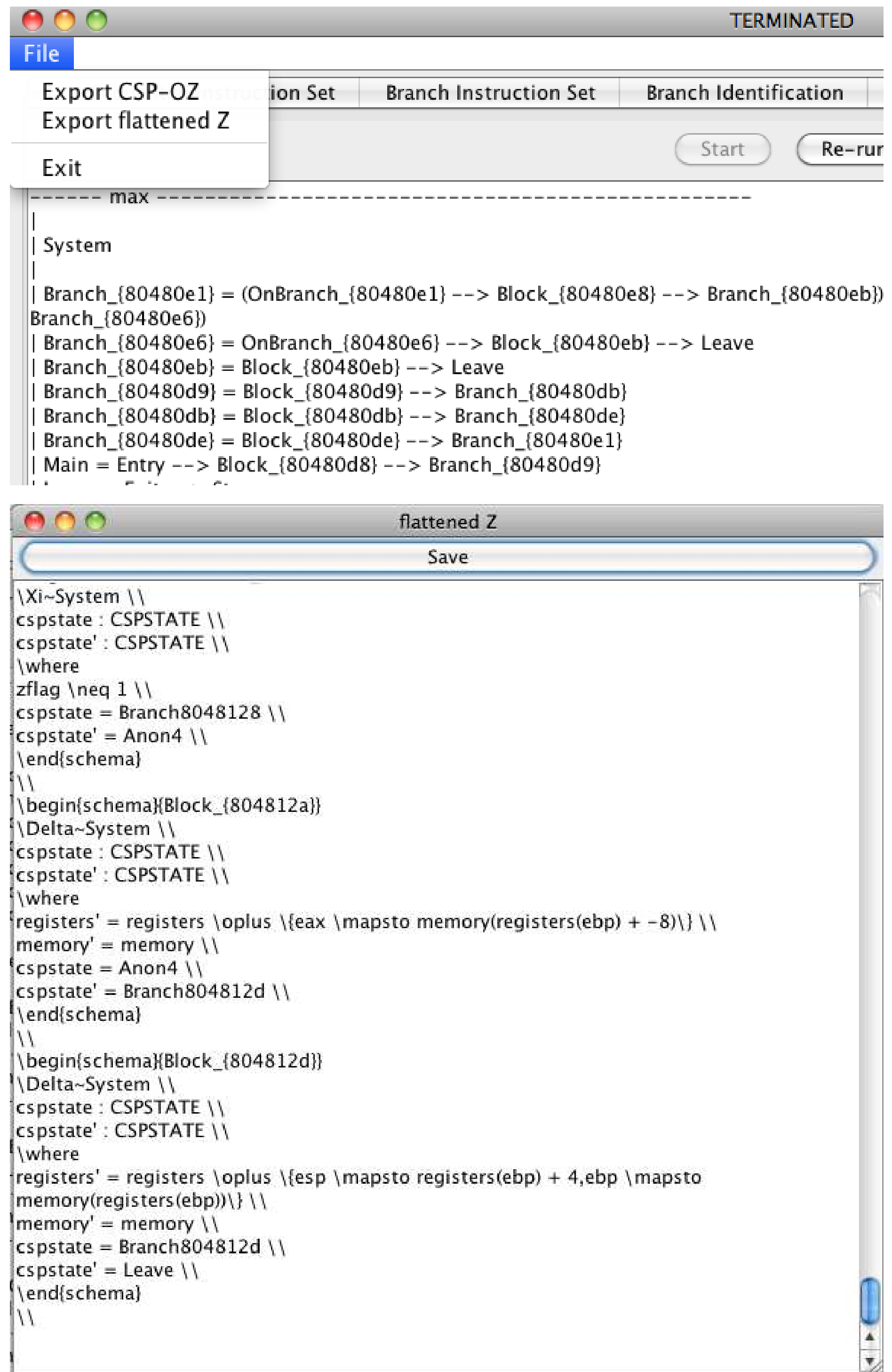
*NoBranch* schema from the branch instruction specification and adds them to the CSP-OZ class with a subscript containing the address of the branch instruction.

Finally, the *Entry*, *Exit*, *Call*, and *Return* schema are added as described in §5.5.3. This *CSPOZClass* object is the final result of the analysis. One CSP-OZ class is produced for each function. The collection of classes forms the formal model of the executable file.




## 6.4 Output formats

Spurinna uses a basic, text based representation of the CSP-OZ schema for its user interface. To aid the application of verification tools to the inferred model it is capable of producing LaTeX files for either CSP-OZ, of plain Z using the flattening algorithm described in Appendix A.


TERMINATED
File
Export CSP-OZ
Export flattened Z
Exit
ion Set
Branch Instruction Set
Branch Identification
Start
Re-run
------ max --------------------------------------------
|
| System
|
| Branch_{80480e1} = (OnBranch_{80480e1} --> Block_{80480e8} --> Branch_{80480eb}) Branch_{80480e6})
| Branch_{80480e6} = OnBranch_{80480e6} --> Block_{80480eb} --> Leave
| Branch_{80480eb} = Block_{80480eb} --> Leave
| Branch_{80480d9} = Block_{80480d9} --> Branch_{80480db}
| Branch_{80480db} = Block_{80480db} --> Branch_{80480de}
| Branch_{80480de} = Block_{80480de} --> Branch_{80480e1}
| Main = Entry --> Block_{80480d8} --> Branch_{80480d9}
flattened Z
Save
\Xi~System \\
cspstate : CSPSTATE \\
cspstate' : CSPSTATE \\
\where
zflag \neq 1 \\
cspstate = Branch8048128 \\
cspstate' = Anon4 \\
\end{schema}
\\
\begin{schema}{Block_{804812a}}
\Delta~System \\
cspstate : CSPSTATE \\
cspstate' : CSPSTATE \\
\where
registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + -8)\} \\
memory' = memory \\
cspstate = Anon4 \\
cspstate' = Branch804812d \\
\end{schema}
\\
\begin{schema}{Block_{804812d}}
\Delta~System \\
cspstate : CSPSTATE \\
cspstate' : CSPSTATE \\
\where
registers' = registers \oplus \{esp \mapsto registers(ebp) + 4,ebp \mapsto memory(registers(ebp))\} \\
memory' = memory \\
cspstate = Branch804812d \\
cspstate' = Leave \\
\end{schema}
\\

## 6.5 Summary

This chapter has presented the Spurinna system, which implements the analysis process developed by in this thesis. The subsequent chapters use Spurinna in conjunction with other verification tools to demonstrate complete verifications of properties of software systems.

The Spurinna software described in this chapter is available from http://www.dcs.shef.ac.uk/ ramsay/Spurinna/ along with the sample specifications developed during this work. As a research prototype, its immediate use in the verification of safety or security critical system cannot be supported.

# Chapter 7

# Model checking for verification

## 7.1 Overview

The aim of this work is to develop a process that can analyse software that interacts with hardware and support verification of specified system properties. The previous sections have provided the techniques necessary to analyse such software and infer formal models of its behaviour. This chapter presents the application of model checking techniques to verify properties about the system using the inferred model. Additionally, it demonstrates how errors can be identified in the model and traced back to the implementation to support corrections.

## 7.2 System, requirements, and analysis

To demonstrate a complete analysis and verification process this section will concentrate on the *maxint* example that has been used in §5.1-§5.5. The source code in C is presented in Figure 7.1. The *maxint* function should be given a pointer to a zero-terminated list of unsigned integers and should return the largest. It uses the *max* function to compare two integers and return the value of the largest. If they are equal it doesn't matter which is used as the source of the return value.

The disassembly and analysis to this example is presented elsewhere, so this chapter will concentrate on verification of the behaviour of the inferred model. The complete CSP-OZ model is documented in Appendix B. The decision to produce CSP-OZ models of system behaviour was based on the large range of available analysis techniques that can be applied to CSP-OZ, and to its constituent languages. This chapter will use the Z2SAL[28] system to translate Z into the input language of the SAL model-checker suite. The system requirements

```
int max(unsigned int x, unsigned int y) {
  if(x > y) {
    return x;
  } else {
    return y;
  }
}

int maxint(unsigned int *ints) {
  unsigned int result, *ptr;

  result = ints[0];
  ptr = (ints + 1);
  while(*ptr != 0) {
    result = max(*ptr, result);
    ptr++;
  }
  return result;
}
```

Figure 7.1: maxint.c

will be represented as Linear Temporal Logic theorems that the SAL suite can then verify or counter with examples.

The Z2SAL system does not support Z's Theta notation, so the function call and return model cannot be converted. Consequently this analysis is limited to a single function. The *max* function was chosen, as its behaviour is not dependent on any other function. The requirements of the *max* function can be defined in natural language as:

- **Termination** The function should always reach a return state, and it should not perform further operations from this state
- **Sensible return value** C functions pass arguments on the stack, along with the return value. For the Intel system modeled here the arguments are found at $esp + 4$ and $esp + 8$ on function entry. The *max* function should return one or other of these, and not some spurious value
- **Correctness of return value** The value returned should be the numerically higher of the two arguments. Since the return is by value it does not matter which of two equal values is used as the source of the assignment

## 7.3 Z2SAL

Z2SAL converts Z specifications into the input language of the SAL suite of model checkers and related tools. SAL models are state machines and so SAL

input models consist of a state specification (with various data type definitions) and a series of operations. SAL machine operations have pre and post conditions with a primed syntax for postconditions that is not dissimilar to that of Z. The various SAL model checkers can then evaluate Linear Temporal Logic (LTL) theorems over these models. Due to the exhaustive behaviour of model checking, the range of datatypes has to be finite and must be constrained as far as possible, since entire state spaces are explored.

Z2SAL is capable of translating a range of Z specifications into SAL models completely automatically. There are some minor limitations to the input parser (such as not supporting subscripts in schema names) but very little needs to be changed from the output format of Spurinna to use Z2SAL immediately on the inferred models.

The most significant limitation is that Z2SAL only processes Z, and not CSP-OZ. Consequently the program branching behaviour that is modeled by the CSP component of the Spurinna models must be "flattened" into pure Z. SAL state machine models contain a set of operations with preconditions, so the natural way to perform this flattening is to add a suitable precondition and postcondition to each operation that will constrain the order of execution to the sequence prescribed by the CSP model. Performing this flattening automatically is possible, and has been implemented in the Spurinna system for the simple CSP cases that occur in the inferred models. The algorithm is discussed in detail in Appendix A. As an example the CSP-OZ combination:

$$Branch_{80480d9} \quad = \quad Block_{80480d9} \rightarrow Branch_{80480db}$$

$Block_{80480d9}$

$\Delta$ *System*

---

$registers' = registers \oplus \{ebp \mapsto registers(esp)\}$
$memory' = memory$

becomes:

$Block80480d9$

$\Delta$ *System*
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

---

$registers' = registers \oplus \{ebp \mapsto registers(esp)\}$
$memory' = memory$
$cspstate = Branch80480d9$
$cspstate' = Branch80480db$

The precondition on the *cspstate* variable ensures that this operation will only execute if the system has reached the state represented by the CSP $Branch80480d9$

process. Some additional complexity occurs where several paths pass the same operation. Concurrency for the CSP parallel composition operator has not yet been implemented because it was not necessary for any of the examples presented here. It could be represented by the introduction of a separate *cspstate* variable for each parallel process, and a shared variable that constrains the execution of both paths until the CSP system enters the parallel section. The possible interleavings should then be explored automatically by the modelchecker, as all the possible interleaved sequences would be executable in the SAL representation.

The Z representation presented at Appendix C was automatically produced by Spurinna. Some minor syntactic modifications had to be performed manually because of limitations in the Z2SAL parser. The *Init* schema can only initialise variables from the System state schema, so the initialisation for the *cspstate* variable had to be added to the SAL file manually. Additionally, the Z2SAL system always includes an *ELSE* transition in its machine that can always execute and which doesn't alter the system state. This produces a mathematically complete transition function but it confounds attempts to prove LTL theorems, since the machine can loop in the *ELSE* transition without ever completing the modeled operation. Removing the *ELSE* transition from the SAL file solved this problem, but then produced another: the end of any CSP process sequence is the *STOP* transition, which represents deadlock, but is modeling correct termination. The SAL suite correctly identified this deadlock condition, but its presence prevents the detection of other deadlock conditions, since it is easy to identify it will always be returned as the counterexample from the deadlock checker, possibly masking other deadlock conditions. To account for this the *STOP* operation was added, using the same operation code as the original *ELSE* transition (i.e. performing no action) but it has the precondition that *cspstate* variable must contain *Leave*, the value assigned to termination. Once the system has reached the *Leave* state this *STOP* operation can execute infinitely, but causes no changes to the system state. Since it is able to execute it does not show up as a deadlock state in the deadlock checker, allowing for an accurate analysis of the rest of the model.

Initial experiments identified an interesting additional limitation of the system as represented. The stack operations on Intel processors grow the stack downwards (that is, they decrement the stack pointer and place the next value at a numerically lower address), so it is conventional to compile programs to be loaded at the middle of the virtual address space and have the heap grow upwards and the stack grow downwards (hence the 80 prefixes on all of the instruction addresses in the small examples). However, the SAL model is using natural numbers and limits their range to as small a set as possible. Consequently, without additional guidance it became very easy for the stack pointer to be close to zero and then decremented. Since decrementing a natural number below zero is not possible it produced unpredictable results. This is an important insight into a possible failure mode of the program on a system with a much more limited address space, but it can be safely excluded if the address space will always be many orders of magnitude too big for it to occur. To represent

this while still limiting the state space for the SAL natural number type, it was necessary to introduce an explicit *Init* function to tell Z2SAL to provide adequate space for the stack to contain two parameters and a return address, and to grow by at least one 32bit value when the base pointer is pushed on entry.

The full SAL file can be found at Appendix D.

## 7.4 Model checking Linear Temporal Logic properties

The principle requirements for the *max* function were defined as:

- **Termination** The function should always reach a return state, and it should not perform further operations from this state
- **Sensible return value** C functions pass arguments on the stack, along with the return value. For the Intel system modeled here the arguments are found at $esp + 4$ and $esp + 8$ on function entry. The *max* function should return one or other of these, and not some spurious value
- **Correctness of return value** The value returned should be the numerically higher of the two arguments. Since the return is by value it does not matter which of two equal values is used as the source of the assignment

The termination condition was encoded in two SAL LTL theorems:

```
alwaysTerminates : THEOREM State |- F(cspstate = Leave);
neverRestarts : THEOREM State |-
      G((cspstate = Leave) => NOT F(cspstate /= Leave));
```

The first theorem states that the system should eventually reach the *Leave* state of the *cspstate* variable. The second theorem requires that this be a terminal state and the system cannot proceed once it reaches this state. Running the SAL symbolic model checker on the first theorem produces the following response:

```
bash-3.2$ sal-smc maxint alwaysTerminates
proved.
WARNING: Your property is only true if it is deadlock free.
You should run sal-deadlock-checker for that.
```

Following the advice and running the deadlock checker confirms that there are no deadlock states possible. This demonstrates the necessity of the *STOP* operation to exclude the terminal deadlock and allow SAL to confirm that no other deadlock conditions occur.

```
bash-3.2$ sal-deadlock-checker maxint State
ok (module does NOT contain deadlock states).
```

The second theorem is completely proven for the state space.

```
bash-3.2$ sal-smc maxint neverRestarts
proved.
```

The definition of the *ret* instruction in §5.2.3 assumes that the return address on the stack has not been altered, so it is important to verify this property.

```
noStackOverflow : THEOREM State |-
   FORALL (VAL:NAT):
     ((cspstate = Anon2) AND ((memory(registers(esp)) = VAL))
     =>
     G((cspstate = Leave) => (memory(registers(esp)) = VAL)));
```

This takes a considerable time but is eventually proven.

```
bash-3.2$ sal-smc maxint noStackOverflow
proved.
```

The requirement that the function returns one of the two parameters on the stack is encoded with the third LTL theorem:

```
returnsSomethingSensible : THEOREM State |-
        G(
          (cspstate = Leave) =>
          (
            (registers(eax) = memory(registers(esp) + 4))
            OR
            (registers(eax) = memory(registers(esp) + 8))
          )
        );
```

This states that once the system reaches the terminal *Leave* state then the value in the *eax* register should be one of the two parameters that are on the stack, and so present in memory at the indirect addresses of $esp + 4$ and $esp + 8$ It is proven by the symbolic model checker in a short time.

```
bash-3.2$ sal-smc maxint returnsSomethingSensible
proved.
```

Finally, the requirement that the function returns the maximum of the two values can be encoded thus:

```
returnsTheRightAnswer : THEOREM State |-
 G((cspstate = Leave) =>
   (
     (
       (registers(eax) = memory(registers(esp) + 4)) =>
```

```
int max(unsigned int x, unsigned int y) {
  if(x < y) {
    return x;
  } else {
    return y;
  }
}
```

Figure 7.2: brokenmaxint.c

```
        memory(registers(esp) + 4) >= memory(registers(esp) + 8)
    )
    AND
    (
      (registers(eax) = memory(registers(esp) + 8)) =>
        memory(registers(esp) + 8) >= memory(registers(esp) + 4)
    )
  )
);
```

This is also proven in a reasonable time.

```
bash-3.2$ sal-smc maxint returnsTheRightAnswer
proved.
```

## 7.5 Model checking for fault detection

Presenting a series of LTL theorems that seem to represent the required properties and then presenting that the SAL suite returns "proved" is not a totally compelling demonstration of a verification system. Proving properties about a system is important, but the behaviour of the analysis technique when identifying defects (deviations from the required properties) provides a better demonstration of its usefulness in a software engineering workflow, and gives more confidence in the validity of the proven properties.

To demonstrate the fault identification abilities of these techniques the *max* function was modified to introduce a deliberate error. Figure 7.2 presents the *max* function with the $>$ symbol replaced by $<$. This sort of typographic error occurs frequently and is easily overlooked in code reviews, especially in the middle of large blocks of complex code.

The resulting C file was saved as "broken-maxint.c" and underwent the same compilation, linking, disassembly, and analysis as was described for *maxint*. It is a reassuring demonstration of the structural compatibility between the CSP-OZ model and the underlying program that the only difference between the two inferred models is in the branch choice. A diff of the two LaTeX files shows only two lines changed:

```
75c75
< sflag = 0 \\
---
> zflag = 1 \lor sflag = 1 \\
81c81
< sflag \neq 0 \\
---
> zflag \neq 1 \land sflag \neq 1 \\
```

The conversion to SAL with the Z2SAL system proceeds easily, with the same small manual changes to the SAL file. Again, diff only highlights the difference that is expected in the branch precondition and the change of file name.

```
1c1
< brokenmaxint : CONTEXT = BEGIN
---
> maxint : CONTEXT = BEGIN
124c124
<           sflag = 0 AND
---
>           (zflag = 1 OR sflag = 1) AND
144c144,145
<           sflag /= 0 AND
---
>           zflag /= 1 AND
>           sflag /= 1 AND
```

The same sequence of theorems can be evaluated for the broken program, since they are the requirements that would be expected of the final system. The change to the logic makes no difference to program termination, or to the program returning one of the two potential results:

```
bash-3.2$ sal-smc brokenmaxint alwaysTerminates
proved.
WARNING: Your property is only true if it is deadlock free.
You should run sal-deadlock-checker for that.
bash-3.2$ sal-deadlock-checker brokenmaxint State
ok (module does NOT contain deadlock states).
bash-3.2$ sal-smc brokenmaxint neverRestarts
proved.
bash-3.2$ sal-smc brokenmaxint returnsSomethingSensible
proved.
```

It is only the final theorem, representing the semantic condition of the result that identifies the failure. The complete counterexample is included at Appendix E. The last step of the counterexample is adequate to demonstrate that there is a flaw in the logic of this *max* implementation:

```
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480eb
    transition at [Context: brokenmaxint, line(215), column(10)]))
------------------------
Step 7:
--- System Variables (assignments) ---
registers(eax) = 12
[...]
registers(esp) = 4
registers(ebp) = 12
[...]
memory(8) = 13
[...]
memory(12) = 12
[...]
cspstate = Leave
invariant__ = true
```

With *esp* set to 4 the two parameters are found at memory addresses 8 and 12. In this example these contain the values 13 and 12, but the register *eax* contains 12. This seems a difficult to read, but the requirements of the style of low level code that this analysis targets are often written at this level. As such, the authors of such code are likely to be familiar with the low level system and can be expected to be used to reviewing similar register and stack "dumps" in the course of standard debugging. The complete counter example trace can be followed to identify the point at which the erroneous value is written to the *eax* register; in this case it is loaded in Step 5 ($Block80480e8$).

```
Step 5:
--- System Variables (assignments) ---
registers(eax) = 13
[...]
------------------------
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480e8
    transition at [Context: brokenmaxint, line(199), column(10)]))
------------------------
Step 6:
--- System Variables (assignments) ---
registers(eax) = 12
[...]
```

The importance of retaining the instruction addresses throughout the entire analysis process is now apparent, as this can be immediately connected back to the disassembly:

```
080480d8 <max>:
 80480d8:55                     push   %ebp
 80480d9:89 e5                  mov    %esp,%ebp

 80480db:8b 45 08               mov    0x8(%ebp),%eax

 80480de:3b 45 0c               cmp    0xc(%ebp),%eax
 80480e1:73 05                  jae    80480e8 <max+0x10>

 80480e3:8b 45 08               mov    0x8(%ebp),%eax
 80480e6:eb 03                  jmp    80480eb <max+0x13>

*80480e8:8b 45 0c               mov    0xc(%ebp),%eax

 80480eb:5d                     pop    %ebp
 80480ec:c3                     ret
```

By directing the user's attention to this line in a short sequence of assembly instructions a rapid understanding can be gained the program component that is being considered. The instruction at $80480db$ loads the first parameter into $eax$, and the immediately following instruction compares this to the other parameter. By reading only a few more instructions the user can identify that the third instruction in the sequence makes a branching decision based on that comparison. The branch target loads the other parameter, whereas the block that would be executed if the branch was not followed skips the instruction at $80480e8$, which was identified as the mistake in the SAL model. Clearly, the program should not have branched on the conditional at $80480e1$, and yet it did, suggesting that the logic of that condition is wrong. This should direct the user to look at the choice logic in the original C file, whereupon they should identify the error.

This level of comprehension is only possible once the user has been directed to the critical three or four lines. In this example the entire assembly program can be reasonably comprehended, but in a system with a million or more instructions that is simply not possible. Additionally, this example depends on the operation of a relatively simple conditional branch. In a different example — perhaps where the error occurred in the application of a more complex instruction — the user could return to the SAL counter example and view the following sequence to appreciate the changes wrought in the system state by the subsequent instructions and so gain a better understanding of the interrelation between state components that was the source of the error.

## 7.6 Summary

The Z2SAL tool offers an easy and efficient way to allow the use of the SAL model checking suite on the model inferred by the analysis process. This chapter

has demonstrated the use of Z2SAL and the SAL suite to verify requirements of the system by encoding them as LTL theorems and applying the SAL tools. Additionally, this chapter has demonstrated the usefulness of the traceability information maintained in the model when requirement violations are identified. The only limitations to this verification process are the limitations of the Z2SAL parser, and the finite nature of the SAL state models. The next chapter will address the state limitations by applying symbolic proof techniques to evaluate properties over infinite domains.

# Chapter 8

# Verification by symbolic proof

## 8.1 Overview

In §7 various properties of the *max* example were confirmed with a model checker. In this section the same example and the same properties will be proven using the formal proof environment Isabelle (see §2.5.4). This requires rendering the model as an Isabelle theorem, using a lightweight Z representation that preserves the traceability data that has been maintained throughout the analysis process.

## 8.2 Symbolic proof in IsabelleHOL

Model checking is an effective way to assess properties of a system with a small state space, but as the state space grows the model checker becomes more time consuming, and quickly exhausts its memory. In the case of a large state space but a relatively straightforward operational model there are advantages to applying symbolic proof techniques. Whilst a model checker can demonstrate that a property holds for a finite range of values, a symbolic proof can show that the property is universally true. This has certain limitations in that not all properties can be proven, and the proof of even simple properties can require a large number of facts about the system. This limits the model complexity that can be realistically handled by formal proof, since more complex models may require intractably large numbers of facts, in the same way that the state space size is the limiting factor for model checkers. The choice between model checking and proof rests on the simplicity of automation. As shown by the Z2SAL tool and the work in §7 it is possible to achieve almost complete automation of the analysis of properties in a model checker, but the verification is limited to finite domains. Formal proof systems like Isabelle can make use of more subtle

approaches to verify properties over infinite domains, but this can require some level of human interaction to direct the proof.

A number of representations of Z exist for Isabelle, including HOLZ[43] but these require higher-order features of Isabelle in order to properly represent the more advanced semantics of Z. A specific intention of this work is to prioritise readability and traceability for the semantically simple machine instructions found in assembly language. As such it was decided that a new, lightweight representation of Z in Isabelle would be constructed for this verification. By using a lightweight, first order model it becomes possible to apply the automated theorem provers that are included in the Isabelle system.

The representation used here models Z state schema as records (in a similar way to the Z schema promotion syntax). The current state of the system at a given point contains two elements: a record containing all of the state variables and their current values, and a Boolean value representing whether this state has been reached but in violation of the precondition of an operation. The Z theory file contains a generic definition of a system state that must be parameterised by the actual system state schema.

The representation of a Z operation schema then contains two propositions: a Boolean proposition that models the precondition of the operation, and a record function that uses the record assignment syntax to update the state in according to the postcondition of the Z operation.

```
record system =
  x :: int
  y :: int

definition S :: "system Op" where
"S St ≡
   ⦇ sys = ⦇ x = (x (sys St)) + 1, y = (y (sys St)) ⦈,
   cond = x < y ⦈"
```

This example defines a system state containing two integer variables and then defines an operation on that state that increments `x` and leaves `y` unchanged, but has the precondition that `x` is less than `y`. The `Op` type is defined in the Z theory file to represent single Z operations.

The Z model also includes composition operators based on the Z and CSP-OZ operators. The semicolon operator models sequential composition and the external choice operator is included. To model the choice operator the operations are expanded with the *AllPos* function to apply across a list of states and produce a list of states. This maps the operation across all of the states in the argument, and then filters the results based on the precondition so that the resulting list is only those states that have not violated the precondition of the operation. The sequential operator then applies the second operation to all of the outputs of the first and performs similar precondition check.

```
definition doubleS :: "system AllPosOp" where
"doubleS ≡ (AllPos S) ; (AllPos S)"
```

The `AllPosOp` type is defined in the Z theory file to represent the mapped operation. It is equivalent to `'a state list ⇒ 'a state list`, where `'a` is the Isabelle syntax for a type parameter.

The choice operator applies both operations to the argument list and produces the list that combines both the output lists, since this model is designed to verify properties of the system it is necessary that the properties hold on *all possible* results.

```
definition maybeS :: "system state ⇒ system state" where
"maybeS ≡ (AllPos S) □ (AllPos R)"
```

This models the external choice between performing operation `S` and some other operation, `R`. In this model, however, the output will be the list containing the result of mapping both operations over the input set, so it will be the list of all possible states that could be reached from this list of initial states.

Both the sequential composition and choice operators' definitions include rules to filter the output lists to only contain states where the precondition has not been violated. This is particularly important in the case of the choice operator. Since it models external choice if it composes two operations, only one of which has its preconditions satisfied, it will fill the output list with only the result state from the branch that is available and not the branch that is refused.

## 8.3 Converting Z to an Isabelle model

The *max* example was converted manually to the Isabelle representation to demonstrate the applicability of this verification technique. In the future is would be possible to automate this rendering process in a similar way to the Z2SAL process for the SAL input language.

Several helper functions are included to simplify access to registers and memory locations in the two-level record structure.

```
definition mem :: "system state ⇒ int ⇒ int" where
"mem St ≡ (memory (sys St))"
declare mem_def [simp]
```

This defines the *mem* operator as simply equivalent to requesting the *memory* element from the *sys* record element of the *St* record parameter. The last line explicitly identifies this as a simplification rule that the Isabelle simplifier is allowed to apply to all instances of the *mem* operator to make their behaviour more explicit.

```
definition reg :: "system state ⇒ Regname ⇒int" where
"reg St ≡ (registers (sys St))"
declare reg_def [simp]
```

```
definition regind :: "system state ⇒ Regname ⇒ int ⇒ int" where
"regind St ≡ λr . λoff . ((mem St) ((reg St r) + off))"
declare regind_def [simp]
```

These rules produce similar operators for the register and register indirect access mechanisms. The first block of the CSP-OZ model, $Block80480d8$, is modeled thus:

$$
\begin{array}{l}
\textit{Block}_{80480d8} \\
\Delta\ \textit{System} \\
\hline
\textit{registers}' = \textit{registers} \oplus \{\textit{esp} \mapsto \textit{registers}(\textit{esp}) - 4\} \\
\textit{memory}' = \textit{memory} \oplus \{\textit{registers}'(\textit{esp}) \mapsto \textit{registers}(\textit{ebp})\}
\end{array}
$$

```
definition Block80480d8 :: "system Op" where
"Block80480d8 St ≡
  (|sys=(sys St)(|
  registers := (registers (sys St))(esp := ((reg St esp) - 4)),
  memory :=
    (memory (sys St))(((reg St esp) - 4) := (reg St ebp))|)
  , cond=True|)"
declare Block80480d8_def [simp]
```

This demonstrates the advantage of this lightweight notation over the more semantically rich HOL Z notations in that it retains a large portion of the syntactic structure of the original, making traceability easier. The lack of the posterior variables forces the only substantial change: that $\textit{registers}'(\textit{esp})$ has to be rendered with the expression used in the other assignment, `(reg St esp) - 4`. These operation definitions are also registered as simplification rules so that the Isabelle simplifier can unravel the compositions that will be created later and produce their behaviors explicitly.

The next few blocks can be written similarly. The complete Isabelle theory file is included at Appendix G. The first three blocks of the program setup the stack after the function call and load some registers from memory. This should always work, there should be no precondition for this sequence. In this model that is represented by the statement that for all possible argument states the *AllPos* operation produces at least one state — that is, the precondition never prevents the composition operator from producing a result state.

```
lemma firstThreeNoPrecond [simp] :
    "∀St::(system state) .
            ¬ ((((AllPos Block80480d8)
                    ;(AllPos Block80480d9)
                    ;(AllPos Block80480db)) [St]) = [])"
by simp
```

The lemma is presented to Isabelle, which typechecks the statement and builds an internal model of the proposition. The proof can be presented as a

sequence of applications of rules from the base logic or from previous parts of this theory file. In this case the simplifier is able to expand all of the proposition elements to a point that the Isabelle system can resolve trivially, so only the line `by simp` is necessary.

The *AllPos* function expands the singular operators to work on lists of states, and the sequential composition combines them as described previously. Since all the operation definitions are declared to be simplification definitions the Isabelle simplifier can apply them to the composition. In this case that is all that is needed to prove that the precondition is never violated. In this case they all have the precondition *true*, so this is not surprising, but it does demonstrate that all of the composition operators work correctly and that this approach to specification and verification is succinct.

The process of converting operations is not complex and a rendering system could be produced for most cases easily, but some Z postconditions are slightly more difficult. For example the block at address $80480de$ includes conditional postconditions:

$$
\begin{array}{l}
Block_{80480de} \\
\Delta\ System \\
\hline
registers(eax) - memory(registers(ebp) + 12) < 0 \\
\quad \Rightarrow sflag' = 1 \land zflag' = 0 \\
registers(eax) - memory(registers(ebp) + 12) = 0 \\
\quad \Rightarrow zflag' = 1 \land sflag' = 0 \\
registers(eax) - memory(registers(ebp) + 12) > 0 \\
\quad \Rightarrow zflag' = 0 \land sflag' = 0 \\
memory' = memory \\
registers' = registers
\end{array}
$$

This is easily modeled using the choice operator. Since the guards are mutually exclusive it can be modeled by representing each line as an operation that uses the left hand site of the implication as the precondition and the right hand side as the postcondition. The model for the complete block is then built as the external choice between them, which will then resolve to apply the postcondition from the one operation whose precondition is satisfied.

```
definition Block80480dePartA :: "system Op" where
"Block80480dePartA St ≡ ⦇sys=(sys St)⦇
  zflag:=zero,sflag:=one⦈
  , cond=((((reg St) eax) - (regind St ebp 12)) < 0)⦈"
declare Block80480dePartA_def [simp]

definition Block80480dePartB :: "system Op" where
"Block80480dePartB St ≡ ⦇sys=(sys St)⦇
  zflag:=one,sflag:=zero⦈
  , cond=((((reg St) eax) - (regind St ebp 12)) = 0)⦈"
```

```
declare Block80480dePartB_def [simp]

definition Block80480dePartC :: "system Op" where
"Block80480dePartC St ≡ (|sys=(sys St)(|
  zflag:=zero,sflag:=zero|)
  , cond=((((reg St) eax) - (regind St ebp 12)) > 0)|)"
declare Block80480dePartC_def [simp]

definition Block80480de :: "system AllPosOp" where
"Block80480de St ≡ (((AllPos Block80480dePartA)
                    □(AllPos Block80480dePartB))
                    □(AllPos Block80480dePartC)) St"
declare Block80480de_def [simp]
```

However, this interpretation requires a fairly deep semantic understanding of the CSP-OZ model, so is more than a simple textual conversion. The remainder of the Z operations can be represented in a similar way. The full Z theory file is presented in Appendix F.

The operations are then combined into sets of sequential compositions, and finally composed with choice operators to form the complete model:

```
(* The first sequential blocks with no branches *)
definition Entry :: "system AllPosOp" where
"Entry ≡ λSt .
      ((AllPos Block80480d8)
      ;(AllPos Block80480d9)
      ;(AllPos Block80480db)
      ;Block80480de) St"
declare Entry_def [simp]

(* The two choices at 189 *)
definition Left :: "system AllPosOp" where
"Left ≡ λSt .
      ((AllPos OnBranch80480e1)
      ;(AllPos Block80480e8)
      ;(AllPos Block80480eb)) St"
declare Left_def [simp]

definition Right :: "system AllPosOp" where
"Right ≡ λSt .
      ((AllPos NoBranch80480e1)
      ;(AllPos Block80480e3)
      ;(AllPos OnBranch80480e6)
      ;(AllPos Block80480eb)) St"
declare Right_def [simp]
```

```
(* The complete spec for the Max function *)
definition MaxOp :: "system AllPosOp" where
"MaxOp St = (Entry;(Left□Right)) St"
declare MaxOp_def [simp]
```

## 8.4 Proofs of properties

The significant properties required of the system were:

- **Termination** The function should always reach a return state, and it should not perform further operations from this state
- **Sensible return value** C functions pass arguments on the stack, along with the return value. For the Intel system modeled here the arguments are found at $esp + 4$ and $esp + 8$ on function entry. The *max* function should return one or other of these, and not some spurious value
- **Correctness of return value** The value returned should be the numerically higher of the two arguments. Since the return is by value it does not matter which of two equal values is used as the source of the assignment

Additionally, the definition of the *ret* instruction required a check that there was no stack overflow possible. These properties can be written and proven as follows:

```
theorem MaxTerminates : "∀St::system state .
         MaxOp [St] ≠ []"
by simp

theorem NoStackOverflow :
  "∀St::system state .  ∀ASt::system state .
  cont ASt (MaxOp [St]) ⟶ (regind St esp 0 = regind ASt esp 0)"
by simp
```

For the last two properties the simplifier requires a few hints before it is able to completely discharge the proof requirements, but these can be supplied in the form of lemmas. The simplifier has a limit to the depth to which it will simplify in one call, so these lemmas simply guide it and allow it to take a number of shorter steps.

```
lemma functionsAreSane [simp] : "∀St . ∀n . ((mem St) n = (mem St) n)"
by simp

lemma intCommute [simp] : "∀n::int . ∀m::int . (n + m) = (m + n)"
by simp

theorem MaxGetsSomethingSensible [simp] :
```

```
  "∀St::system state . ∀ASt::system state .
  cont ASt (MaxOp [St]) ⟶
  (
  let n = (regind St esp 4); m = (regind St esp 8) in
  ((reg ASt eax) = n) ∨ ((reg ASt eax) = m)
  )"
by simp

theorem MaxReturnsMax :
  "∀St::system state . ∀ASt::system state .
  cont ASt (MaxOp [St]) ⟶
  (
  let n = (regind St esp 4); m = (regind St esp 8) in
  (n ≥ m ⟷ (reg ASt eax = n))
  )"
by simp
```

## 8.5 Proof assistant counterexamples for fault detection and tracing

Unlike the model checking example, these proofs are not limited to finite ranges of values and should apply universally. Nonetheless, the ability to detect defects and the level and style of feedback provided is still a more useful measure of a verification process. The same *brokenmax* implementation that was used in the modelchecking example in §7.5, with the same defect, can be rendered in the Isabelle representation. The only significant changes to the correct implementation are the branch prefixes at address $80480e1$:

```
(* Changes in the broken version... *)
definition WrongOnBranch80480e1 :: "system Op" where
"WrongOnBranch80480e1 St = (|sys=(sys St),
  cond=(sflag (sys St) = zero)|)"
declare WrongOnBranch80480e1_def [simp]

definition WrongNoBranch80480e1 :: "system Op" where
"WrongNoBranch80480e1 St = (|sys=(sys St),
  cond=(sflag (sys St) ≠ zero)|)"
declare WrongNoBranch80480e1_def [simp]
```

As with the model checking, this does not alter the first verification conditions.

```
(* The two wrong choices at 189 *)
definition WrongLeft :: "system AllPosOp" where
"WrongLeft ≡ λSt .
```

```
   ((AllPos WrongOnBranch80480e1)
   ;(AllPos Block80480e8)
   ;(AllPos Block80480eb)) St"
declare WrongLeft_def [simp]

definition WrongRight :: "system AllPosOp" where
"WrongRight ≡ λSt .
   ((AllPos WrongNoBranch80480e1)
   ;(AllPos Block80480e3)
   ;(AllPos OnBranch80480e6)
   ;(AllPos Block80480eb)) St"
declare WrongRight_def [simp]

definition WrongMaxOp :: "system AllPosOp" where
"WrongMaxOp St = (Entry;(WrongLeft□WrongRight)) St"
declare WrongMaxOp_def [simp]

(* It should still get an answer *)
lemma WrongMaxTerminates :
   "∀St::system state .
   WrongMaxOp [St] ≠ []"
by simp

(* We should still get one of the two arguments. *)
lemma WrongMaxGetsSomethingSensible [simp] :
  "∀St::system state .
  ∀ASt::system state .
  cont ASt (WrongMaxOp [St]) ⟶
  (let n = (regind St esp 4); m = (regind St esp 8) in
  (((reg ASt eax) = n) ∨ ((reg ASt eax) = m))
  )"
by simp
```

These are all discharged in the same way as the correct versions. Any attempts to prove the correctness theorem, however, will not succeed. The *refute* and *nitpick* tools can be used to find counter examples.

```
theorem WrongMaxReturnsMax :
  "∀St::system state .
  ∀ASt::system state .
  (
  (cont ASt (WrongMaxOp [St])) ⟶
  (let n = (regind St esp 4); m = (regind St esp 8) in
  (n ≥ m) ⟷ (reg ASt eax = n)
  )
  )
```

```
  "
nitpick
apply(simp)
nitpick [verbose=true,show_all=true]
```

## 8.6 Summary

This chapter has demonstrated the feasibility of converting the formal model inferred by the analysis process into a lightweight IsabelleHOL representation. Once converted, it was shown that formal proofs can be constructed that verify the system requirements. The symbolic nature of these proofs makes them applicable over infinite variable domains, unlike the model checking results in §7. However, they require more human interaction than is necessary with the Z2SAL and SAL suite and the language conversion process is not yet automated.

# Chapter 9

# Verifying a hardware usage requirement

## 9.1 Overview

The previous two chapters have demonstrated the use of the analysis process to verify properties of software systems, but these properties have been abstract properties. This chapter presents the verification of a hardware usage property of the kind that cannot be analysed in safe language subsets.

## 9.2 A hardware interaction example

The example used in this chapter is a fictional hardware system. The natural language description of the requirements for the function are:

> The system has an Intel i386 based processor. The device to be controlled has two ports: a control port and a data port. The control port is accessible at the processor's IO port[1] address 0. The data port is accessed at IO port address 4. To request data the driver must write a 1 to the control port, then wait 10ms before the data on the data port is valid. To facilitate the timing there is a clock device available at IO port 8, which presents an integer representing time on an arbitrary scale that increments once per ms.

The designed operation of the device driver function is to write a 1 to the control port, read the clock port and store the value read as the start time, loop reading from the clock port until the value has increased by at least 10 from

[1] Intel processors separate IO port operations from memory accesses and have different instructions for each. The *in* and *out* instructions are used to read from and write to IO ports.

the start time, read the data port and return the value read. The C program to implement this is presented below.

```
#define out(port, value) asm("out %1,%0" : : "dN" (port), "a" (value))
#define in(port, result) asm("in %1,%0" : "=a" (result) : "dN" (port))

#define CONTROL_REG 0
#define DATA_REG 4
#define CLOCK_REG 8


int exdev() {
  int starttime;
  int endtime;
  int now;
  int result;

  // Output a request
  out(CONTROL_REG, 1);
  // read the clock to get the start time.
  in(CLOCK_REG, starttime);
  // Wait 100ms
  endtime = starttime + 10;
  do {
    in(CLOCK_REG, now);
  } while(now < endtime);

  // Return the response register;
  in(DATA_REG, result);
  return result;
}
```

The use of IO port operations is sufficiently target specific that it is not even possible with pure C, additional inline assembler code must be included. The literal addresses of the IO ports are contained in the preprocessor definitions to ease readability.

The process of compilation to an executable, disassembly, analysis with Spurinna, and conversion to a SAL model with the Z2SAL tool is identical to the process used for the *maxint* example in §7, so it is not described here. The only change to the system specification is to add the *ports* function that is defined from natural numbers to natural numbers in the same way as the *memory* function. Unlike the memory function, the instruction operations do not contain any postconditions that ensure that the values present at particular ports are unchanged between operations. Since the posterior state of the *ports* function is undefined the modelchecker and any other verification tool must assume that the ports could assume any value at each step. This neatly represents

the volatile nature of data on IO devices, and demonstrates the reason for the choice of the Z language — which allows variables to be left unspecified — over more incremental state based modeling languages.

$$
\begin{array}{|l}
\hline
\textit{System} \\
\textit{registers} : \textit{REGNAME} \rightarrow \mathbb{N} \\
\textit{memory} : \mathbb{N} \rightarrow \mathbb{N} \\
\textit{ports} : \mathbb{N} \rightarrow \mathbb{N} \\
\textit{zflag} : \{0, 1\} \\
\textit{cflag} : \{0, 1\} \\
\textit{oflag} : \{0, 1\} \\
\textit{sflag} : \{0, 1\} \\
\hline
\end{array}
$$

The C function has four local variables. The GCC compiler creates space for these variables on the stack when the function is entered. Since this is a 32bit system the four integers require 16 bytes, and the 4 byte stack base pointer that is also pushed on function entry totals 20 bytes required for stack space. To simplify the model and optimise the size of the `NAT` type in the SAL representation the system is initialised with the stack pointer at address 20.

$$
\begin{array}{|l}
\hline
\textit{Init} \\
\textit{System}' \\
\hline
\textit{registers}'(\textit{esp}) = 20 \\
\textit{registers}'(\textit{ebp}) = 20 \\
\hline
\end{array}
$$

## 9.3 SAL verification

The SAL model that is produced has 20 transitions and uses a set of natural numbers that runs from 0 to 21, with 22 as the unused bottom element required by the Z2SAL models of functions[2]. It is included at Appendix H.

Unfortunately, using natural numbers up to 22, and the introduction of the *ports* function that is also defined over them, produces a state space that is too large for the symbolic model checker. In principle, this model could be simplified by creating smaller types for the port addresses, containing only those values that are used — 0, 4, and 8 — but this would only solve the problem for a borderline case such as this. Instead, the SAL suite contains a *bounded model checker* that produces all possible traces to some specified depth and then applies a SAT solver to evaluate the required LTL property over the trace set. This has several limitations that are discussed by the SAL FAQ page[5]. Specifically, it is not possible to evaluate the `F` (*future*) and `U` (*until*) operators correctly. For properties that must be true in the future the suite is searching for a counter example, if it does not find the required property in the traces presented it assumes that it may be present if the traces were extended, and so

[2]The SAL modeling of Z functions is fully described in Derrick et al. [28]

does not present this as a counter example. The only violation it can detect is a deterministic cycle that do not contain the property. The strict *until* operator requires that the property is eventually true, and this has the same difficulties as the *future* operator. The `G` (*globally*) and `W` (*weak until*) operators can be evaluated over finite traces, although their evaluation is not technically accurate since they could be violated if the traces were extended.

These limitation prevent the application of the termination properties that were verified in the *maxint* example. The `alwaysTerminates` property can be evaluated on finite traces but its evaluation demonstrates nothing since a trace not ending in the terminal state would result in SAL assuming it would if the trace were extended. This is an incorrect assumption given the arbitrarily volatile specification of the *ports* function. Since the formal specification does not require the values in the clock port to increment it is a legitimate instantiation for the delay loop to never terminate if the clock port never presents a value that is 10 greater than its initial value. The LTL properties could all be prefixed by the requirement that the clock port increments:

```
FORALL (VAL:NAT): G((ports(8) = VAL) => X(ports(8) > VAL)) => <...>
```

This would require the properties to only hold on those traces that can be expected to exist in the real system. Since termination is difficult to determine using the bounded model checker, and the other properties can be evaluated without this requirement, it is not explored further.

The `neverRestarts` property can be evaluated meaningfully but its evaluation only demonstrates that none of the finite traces ever left the terminal state within the bounded number of steps. Since only the `Stop` transition is valid in the terminal `Leave` state it is a reasonable assumption that this property will continue to hold.

That the system ever terminates can be demonstrated by evaluating the inverse property and finding a counter example.

```
neverTerminates : THEOREM State |- G(cspstate /= Leave);
```

A counter example is quickly identified that shows a trace which reaches the terminal state.

Due to the volatile nature of the IO port contents the system properties regarding the hardware usage must use quantified variables that capture the state of the ports at the moment they are read or written by the software. A simple text search can identify the transition in the model that operate on a specific port. This ability to identify the parts of a system that interact with particular state is known as *slicing*[56] when performed in a formal way, and its applicability to CSP-OZ models — which includes the development of automated tools such as developed by Syspect [54] — was another consideration in the choice of CSP-OZ as an output language.

The accesses to the ports are identified as:

- **Control port - ports(0)** written to in `Block80480c3`, which is activated by cspstate `Branch80480c3`

- **Data port - ports(4)** read in `Block80480e0`, which is activated by cspstate `Anon0`
- **Clock port - ports(8)** read in `Block80480c3` (cspstate `Branch80480c3`), and again in `Block80480d3` (cspstate `Branch80480d3`)

Since these are the only uses of the control ports, and it can be textually verified that these are the only ports ever used, the system usage requirements can be formulated as LTLs based on the sequence of activation of these transitions.

```
writeBeforeRead : THEOREM State |-
    W((cspstate /= Anon0),(cspstate = Branch80480c3));
```

The `writeBeforeRead` property guarantees that the control port is written at some point before the data port is read. The SAL syntax uses prefix notation for the *weak until* operator, so this states that the system does not activate the data port read transition — preconditioned to require `cspstate = Anon0` — until the system has activated the control port write transition, which is preconditioned to require `cspstate = Branch80480c3`.

```
delayIsCorrect : THEOREM State |- FORALL (START:NAT):
      G(
      ((cspstate = Branch80480c3) AND (ports(8) = START)) =>
      FORALL (FINISH:NAT):
        G(((cspstate = Branch80480d3) AND (ports(8) = FINISH)) =>
          G(((cspstate = Anon0) AND (registers(eax) = FINISH)) =>
            (FINISH >= (START + 10))
          )
        )
      );
```

In the `delayIsCorrect` property, the clock values at the point of the write to the control register is bound to the variable `START`. The value of the clock register in the loop guard is bound to `FINISH`. The condition that the loop ends (the system reaches state `Anon0` after using this value in *eax* for the *cmp* instruction) requires that the `FINISH` values is at least 10 greater than the `START` value.

Having verified that the read only happens after the control register write, and that the delay is of sufficient magnitude, it only remains to verify that the delay occurs *between* the write and the read.

```
delayBetweenWriteAndRead : THEOREM State |-
  W((cspstate /= Anon0), (cspstate = Branch80480d3));
```

The `delayBetweenWriteAndRead` property uses the *weak until* operator to verify that the delay happens before the data port read. An additional statement would be added to enforce that the control port write occurred before the initial clock port read, but the Spurinna simplifier has merged the two operations

to happen simultaneously. This is a good example of the simplifier merging operations to the limit of correct simultaneous representation, and not beyond, since to merge any further operations from the loop would have altered the properties.

## 9.4 Summary

This chapter has demonstrated that the analysis process can be applied to hardware usage conditions. Some of the limitations in applying the SAL model checking suite to realistic examples were discussed, along with some mitigating approaches.

# Chapter 10

# Conclusions

## 10.1 Software analysis

In the introduction to this thesis it was stated that the principle difficulties that current techniques face when verifying hardware-dependent software are:

- They have no way to statically determine the behaviour of code that interacts directly with the hardware.
- Current techniques are necessarily detached from the hardware (for instance language subsets, discussed in §2.3) or treat hardware in an abstract, general way (such as the heap model in Separation Logic §2.5.3).
- Verification must fit into an industrial workflow and not be overly dependent on expert skills, and must be applicable to large scale systems in reasonable time.

Each of these items has been addressed in this work. The analysis process is entirely static and produces a formal model documenting the behaviour of the executable, so is independent of the high-level language concerns that prevent analysis of hardware interaction. The CSP-OZ model is developed at the level of registers and memory addresses, allowing meaningful representation of the hardware interaction likely to be of interest when verifying requirements on hardware usage.

Finally, the process has been automated in the Spurinna prototype, demonstrating that this technique can be performed without human interaction, and so on large scale projects and without excessive requirements for highly skilled users. When developing the model checking example described in §7, the complete process — from compiling the C code, disassembling the executable, inferring a formal model with Spurinna, creating a SAL model with Z2SAL, and checking the LTL properties — was performed several times. After a brief period of familiarisation with the minor file changes required this entire process could be performed in under 10 minutes. This ability to identify faults, correct

the faults, and confirm the corrections within a reasonable time is vital if a technique is to be usable in an industrial development process.

The Z2SAL demonstration, the IsabelleHOL demonstration, and the hardware usage demonstration show that the inferred model is not only readable and complete, but also practically useful for verifying properties of the system. The properties verified by the three examples were not limited to arbitrary model properties, but show the encoding of useful correctness properties on the behaviour of the software.

## 10.2 Future work

**Processor and instruction set specification repository**

The most significant factor limiting the application of the prototype implementation to larger and more complex examples was the size of the instruction set model available. The model produced for the Intel platform was adequate to demonstrate all of the features necessary to specify instructions for a real processor system, but was limited to only 18 sequential instructions, and 10 branch instructions. The development of more instructions is not difficult or, individually, very time consuming but it had limited research value, so was not performed during this work. To develop more complete specifications would require a commitment of time, but the results could then be shared amongst the entire community of potential users. A valuable future achievement would be a repository of specifications for common processors — possibly multiple specifications for each processor that emphasise analysis of different classes of properties. All interested parties could contribute more instruction specifications to the repository, and could highlight and correct errors in those present. This would spread the time consumption and provide at least a starting point for anyone wishing to begin applying this analysis technique.

**Automatic conversion of Z to the lightweight Isabelle rendering**

The example verification presented in §8 demonstrates the value of the Isabelle proof techniques for verification of system properties. Unfortunately, the creation of the Isabelle model had to be performed manually. A tool similar to Z2SAL could be created to parse Z specifications and render the model in the Isabelle form. This would allow an automated workflow similar to the one present when using Spurinna with Z2SAL.

**Automatic backwards traceability**

A key design objective of the analysis process was to allow traceability of faults from the verification stage back to the original code. The examples in §7 and §8 demonstrate the power and applicability of this technique. This process could be automated, at least as far as the assembly code. The block addresses provide enough information, combined with Spurinna's internal graph model

of the segmented code, to allow for syntax highlighting to be applied to the assembly code. This could be performed with a user directed interface where clicking on the block names in the SAL output, the Isabelle output, or the CSP-OZ model, could activate highlighting of the related area. Ideally, if the cause of a requirement violation could be identified automatically, then the code could be automatically annotated where a particular section is the cause of a violation. The range and style of requirements for which this would be applicable is limited, but it has been shown to be possible and useful in other verification systems[3].

# Bibliography


[1] Automated reasoning group. URL http://www.cl.cam.ac.uk/research/hvg/about.html.

[2] Objdump. URL http://www.gnu.org/software/binutils/.

[3] Polyspace. URL www.polyspace.com.

[4] The symbolic analysis laboratory (SAL), . http://sal.csl.sri.com/.

[5] SRI SAL frequently asked questions, . URL http://sal-wiki.csl.sri.com/index.php/FAQ.

[6] javacc. http://javacc.java.net/.

[7] Ldra testbed. http://www.ldra.com/testbed.asp.

[8] Pc-lint. http://www.gimpel.com/html/pcl.htm.

[9] Altran-praxis ltd. http://www.altran-praxis.com/.

[10] Yices. http://yices.csl.sri.com/.

[11] *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[12] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

[13] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 0521895561, 9780521895569.

[14] J-R. Abrial, S. A. Schuman, and B. Meyer. Specification language. In *On the Construction of Programs*, pages 343–410, 1980.

[15] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[16] S. Agerholm and M. J. C. Gordon. Experiments with zf set theory in hol and isabelle. In *TPHOLs*, pages 32–45, 1995.

[17] J. G. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321136160.

[18] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[19] G. Birtwistle and M. Morely. Case study: specifying and checking tk, an asychronous amulet-like microprocessor. February 2003.

[20] J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010. ISBN 978-3-642-14051-8.

[21] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. pages 460–475. Springer-Verlag, 2006.

[22] Sascha Bhme, Micha Moskal, Wolfram Schulte, and Burkhart Wolff. Hol-boogie – an interactive prover-backend for the verifiying c compiler.

[23] H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 66–77, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250743. URL `http://doi.acm.org/10.1145/1250734.1250743`.

[24] D. A. Carrington, D. Duke, R. Duke, P. King, G. A. Rose, and G. Smith. Object-z: An object-oriented extension to z. In S. Vuong, editor, *Formal Description Techniques II, FORTE'89*, pages 281–296. North-Holland, 1990.

[25] P. Chartier. Formalisation of b in isabelle/hol. In D. Bert, editor, *B*, volume 1393 of *Lecture Notes in Computer Science*, pages 66–82. Springer, 1998. ISBN 3-540-64405-9.

[26] L. de Moura, S. Owre, and N. Shankar. The SAL language manual, 2003.

[27] J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer, May 2001. ISBN 1-85233-245-X. URL `http://www.cs.kent.ac.uk/pubs/2001/1200`.

[28] J. Derrick, S. North, and A. J. H. Simons. Z2SAL: a translation-based model checker for Z. *Formal Aspects of Computing*, 23:43–71, January 2011. ISSN 0934-5043. doi: http://dx.doi.org/10.1007/s00165-009-0126-7. URL `http://dx.doi.org/10.1007/s00165-009-0126-7`.

[29] A. Diller. Z and hoare logics. In J. E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 59–76. Springer, 1991. ISBN 3-540-19780-X.

[30] C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman and Hall, London. URL `citeseer.ist.psu.edu/316011.html`.

[31] A. Fox. An algebraic framework for modelling and verifying microprocessors using hol. Technical report, March 2001.

[32] A. Fox. Formal verification of the ARM6 micro-architecture. Technical report, November 2002.

[33] Stephen B. Furber, P. Day, Jim D. Garside, N. C. Paver, and John V. Woods. Amulet1: A micropipelined arm. In *COMPCON*, pages 476–485, 1994.

[34] Jim D. Garside, Stephen B. Furber, and S.-H. Chung. Amulet3 revealed. In *ASYNC*, pages 51–59. IEEE Computer Society, 1999. ISBN 0-7695-0031-5.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/363235.363259.

[36] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359576.359585.

[37] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *proceedings of POPL'01*, 2001.

[38] ISO. Iso/iec 9899 - programming languages - C. Technical report. URL `http://www.open-std.org/JTC1/SC22/WG14/www/standards`.

[39] ISO. Iso/iec 13568:2002 information technology – Z formal specification notation – syntax, type system and semantics. Technical report, 2002.

[40] D. H. Kemp. Specification of VIPER1 in Z. Technical report, Royal Signals and Radar Establishment, September 1988.

[41] B. W. Kernighan and D. M. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall PTR, March 1988. ISBN 0131103628.

[42] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[43] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of z in isabelle/hol. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs*, volume 1125 of *Lecture Notes in Computer Science*, pages 283–298. Springer, 1996. ISBN 3-540-61587-3.

[44] Formal Systems (Europe) Ltd. Failure divergence refinement: Fdr2 user manual, 1997. URL `http://www.fsel.com/fdr2_manual.html`.

[45] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[46] MISRA. Guidelines for the use of the C language in vehicle based software. Technical report, April 1998.

[47] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, June 1999.

[48] P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01, Paris, 2001. Pages 1-19, LNCS 2142*. Springer-Verlag, 2001.

[49] C. H. Pygott. Formal specification of the viper microprocessor in hol. Technical report, Royal Signals and Radar Establishment, June 1990.

[50] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS 2002*, pages 55–74, 2002.

[51] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Proceedings of the Symposium in Celebration of the Work of C.A.R. Hoare*, 1999.

[52] E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors. *Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, Aspen Grove, UT, USA, September 11-14, 1995, Proceedings*, volume 971 of *Lecture Notes in Computer Science*, 1995. Springer. ISBN 3-540-60275-5.

[53] Wei Song and Doug Edwards. Asynchronous spatial division multiplexing router. *Microprocessors and Microsystems - Embedded Hardware Design*, 35(2):85–97, 2011.

[54] Syspect. Final report of the syspect project. Technical report, Carl von Ossietzky University of Oldenburg, 2006. URL `http://syspect.informatik.uni-oldenburg.de/doc/Endbericht.pdf`.

[55] R. Taylor. Separation of Z operations. In Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, page 350. Springer, 2008. ISBN 978-3-540-87602-1.

[56] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. URL `citeseer.ist.psu.edu/tip95survey.html`.

[57] W. B. Toms and David A. Edwards. Indicating combinational logic decomposition. *IET Computers & Digital Techniques*, 5(4):331–341, 2011.

[58] H. Treharne and S. Schneider. Using a process algebra to control b operations. In *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456, London, UK, 1999. Springer-Verlag. ISBN 1-85233-107-0. URL `http://dl.acm.org/citation.cfm?id=647981.743520`.

[59] D. von Oheimb. Hoare logic for java in isabelle/hol. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[60] T. Weber. Bounded model generation for isabelle/hol. *Electr. Notes Theor. Comput. Sci.*, 125(3):103–116, 2005.

[61] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, US, 1996. ISBN 0-13-948472-8.

[62] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Proceedings of FOSSACS'02*, 2002.

# Appendix A

# Flattening CSP-OZ to pure Z

The models produced by this analysis process are written in the CSP-OZ specification language. CSP-OZ classes can be considered as a state transition system with the state modeled by the Z state schema, the transition effects modeled by the Z operation schema, and the transition matrix — essentially, which transition is allowed when — modeled by the CSP component. Some verification processes are only defined over Z and not over CSP-OZ it is helpful to be able to convert a CSP-OZ model into a pure Z model that encodes the same state transition system.

The Object Z components exist principally to collect schema together and ease readability. This can be accomplished adequately with naming conventions so that operation schema from a particular class all have the class name as a prefix. That was not implemented for these examples since all of the verification is performed on single functions but implementation would be easy. The CSP component encodes the transition acceptance and refusal information. To encode this in Z it is necessary to add an additional variable to the operation schema that tracks the current position in the CSP model. Preconditions can be added to the operations that are defined over this variable to allow an operation to be performed only if the system is at the appropriate point in the transition model. Postconditions update the variable to reflect the movement along edges of the transition graph.

This conversion can be achieved entirely automatically. The process requires walking the CSP process system and modifying the Z operations accordingly. Only three of the CSP operators are used in the models produced by this process so these are the only operators handled here.

The simplest CSP operator is the arrow operator that represents action prefixing.

$$P = Op \rightarrow Q$$

Process $P$ allows the Z operation $Op$ to execute and then evolves to become process $Q$. Z operation $Op$ must be enhanced to require that the system is behaving as process $P$ before it will execute. This pattern of using the process name to permit operation will be repeated for all other operations, so the postcondition of $Op$ can be enhanced to set the *cspstate* variable to $Q$. This will allow any operations that are allowed in the $Q$ process to execute.

```
┌─ Op ──────────────────────────────
│ ...
│ cspstate : CSPSTATE
│ cspstate' : CSPSTATE
├──────────
│ ...
│ cspstate = P
│ cspstate' = Q
└───────────────────────────────────
```

This assumes that the variable *cspstate* is unused elsewhere in the Z specification, but any other unused name could be used instead. If process $Q$ is defined thus:

$$Q = Op_2 \rightarrow R$$

then $Op_2$ can be expanded similarly.

```
┌─ Op2 ─────────────────────────────
│ ...
│ cspstate : CSPSTATE
│ cspstate' : CSPSTATE
├──────────
│ ...
│ cspstate = Q
│ cspstate' = R
└───────────────────────────────────
```

The *CSPSTATE* type can be automatically defined as a BNF type with the various process names as atomic components. These two schema now encode the same transition acceptance information as the CSP-OZ model, but in pure Z. $Op_2$ cannot execute until the *cspstate* variable has been set to $Q$, and that is performed as the postcondition of $Op$. This reflects exactly the allowed trace in the CSP model, where $Op$ occurs as an event in $P$, before the system evolves to $Q$ and allows $Op_2$ to occur.

Since the prefix operator defines a process it can be used in a chain. This pattern is common in CSP models:

$$A = b \rightarrow c \rightarrow D$$

where $A$ and $D$ are processes, $b$ and $c$ are events. In this case, the right hand side — $c \rightarrow D$ — is not explicitly named. To handle this the flattening process

must assign it a name and include the name in the *CSPSTATE* type. The Spurinna implementation uses the string *Anon* with a natural number prefix drawn from an internal, incrementing list.

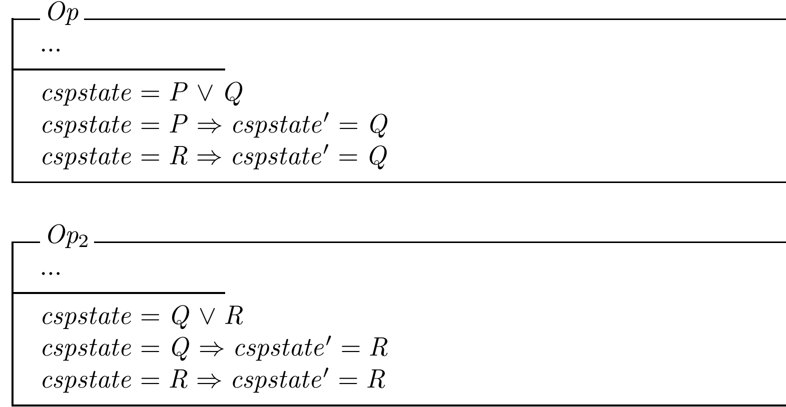
External choice is the second operator that is used. This should allow either of the branches to occur, the decision being made by the environment. Since the external choice operator is defined to conjoin two CSP processes, it is necessary to look at the contents of each of the branches to find the first event.

$$R = P \,\Box\, Q$$

Using the definitions of $P$ and $Q$ from above this identifies the first Z operation schema in the branches as $Op$ and $Op_2$. If these are both preconditioned to allow execution when the system is behaving as $R$, then they can have postconditions to evolve as defined by the branch processes.

This encodes the behaviour of the $R$ process, but it removes the path to $Op_2$ via process $P$, since $Op_2$ no longer responds when the system behaves as $Q$. Additionally, any other processes that evolve to process $P$ cannot activate $Op$.

To complicate things even further, $P$ may not be the only process that uses $Op$ as an event. This combines to require a more subtle set of pre- and postconditions. The flattening process builds an internal model of all the references in and out of an operation and then, if the situation is not trivial, creates a suitable set of implication statements and a precondition disjunction.



$Op$

...

$cspstate = P \lor Q$
$cspstate = P \Rightarrow cspstate' = Q$
$cspstate = R \Rightarrow cspstate' = Q$

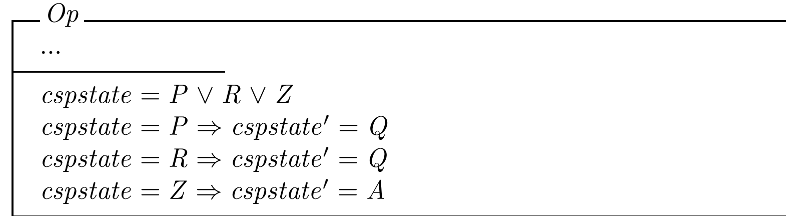
$Op_2$

...

$cspstate = Q \lor R$
$cspstate = Q \Rightarrow cspstate' = R$
$cspstate = R \Rightarrow cspstate' = R$

If an additional process used one of these operations then that could be included as an additional implication.

$$Z = Op \rightarrow A$$



$Op$

...

$cspstate = P \lor R \lor Z$
$cspstate = P \Rightarrow cspstate' = Q$
$cspstate = R \Rightarrow cspstate' = Q$
$cspstate = Z \Rightarrow cspstate' = A$

The final operator used in the inferred models is the parallel composition. This is not used in any of the examples in this thesis, so it is not yet implemented by the Spurinna system. In principle it could be modeled by assigning separate *cspstate* style variables to each component of the composition, and then adding a shared variable that activates both sets of transitions once the system enters the parallel process. The added complexity is that CSP processes must synchronise on communications. The elements of each process could be scanned for communications and then the acceptance variables updated to only allow a transition if one of the corresponding operations — i.e. one with a partner output for each of this operations inputs, and partner input for each output — is ready to execute. The detail of connecting the inputs and outputs together is more complex and would require the use of suitable Z pipe notation. This complexity, and the fact that parallel composition is only used with the schema promotion notation, which is not supported by Z2SAL, is the reason it has not yet been implemented.

# Appendix B

# Derived MaxInt CSP-OZ model

*max*

*System*

$$
\begin{aligned}
Branch_{80480e1} &= (OnBranch_{80480e1} \rightarrow Block_{80480e8} \rightarrow Branch_{80480eb}) \\
&\quad \Box\ (NoBranch_{80480e1} \rightarrow Block_{80480e3} \rightarrow Branch_{80480e6}) \\
Branch_{80480e6} &= OnBranch_{80480e6} \rightarrow Block_{80480eb} \rightarrow Leave \\
Branch_{80480eb} &= Block_{80480eb} \rightarrow Leave \\
Branch_{80480d9} &= Block_{80480d9} \rightarrow Branch_{80480db} \\
Branch_{80480db} &= Block_{80480db} \rightarrow Branch_{80480de} \\
Branch_{80480de} &= Block_{80480de} \rightarrow Branch_{80480e1} \\
Main &= Entry \rightarrow Block_{80480d8} \rightarrow Branch_{80480d9} \\
Leave &= Exit \rightarrow Stop
\end{aligned}
$$

*Entry*

$call? : System$

$\Theta\ System' = call?$

*Exit*

$return! : System$

$\Theta\ System = return!$

*Call*

$call! : System$

$\Theta\ System = call!$

*Return*

$return? : System$

$\Theta\ System' = return?$

$Block_{80480d8}$

$\Delta\ System$

$registers' = registers \oplus \{esp \mapsto registers(esp) - 4\}$
$memory' = memory \oplus \{registers'(esp) \mapsto registers(ebp)\}$

$Block_{80480d9}$

$\Delta\ System$

$registers' = registers \oplus \{ebp \mapsto registers(esp)\}$
$memory' = memory$

$Block_{80480db}$

$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 8)\}$
$memory' = memory$

$Block_{80480de}$

$\Delta\ System$

$registers(eax) - memory(registers(ebp) + 12) < 0$
$\Rightarrow sflag' = 1 \wedge zflag' = 0$
$registers(eax) - memory(registers(ebp) + 12) = 0$
$\Rightarrow zflag' = 1 \wedge sflag' = 0$
$registers(eax) - memory(registers(ebp) + 12) > 0$
$\Rightarrow zflag' = 0 \wedge sflag' = 0$
$memory' = memory$
$registers' = registers$

$OnBranch_{80480e1}$

$\Xi\ System$

$zflag = 1 \vee sflag = 1$

$NoBranch_{80480e1}$

$\Xi\ System$

$zflag \neq 1 \wedge sflag \neq 1$

$Block_{80480e3}$

$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 8)\}$
$memory' = memory$

$OnBranch_{80480e6}$

$\Xi\ System$

$true$

$Block_{80480e8}$

$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 12)\}$
$memory' = memory$

$Block_{80480eb}$

$\Delta\ System$

$registers' = registers \oplus$
$\{ebp \mapsto memory(registers(esp)), esp \mapsto registers(esp) + 4\}$
$memory' = memory$

$maxint$

$System$

$$
\begin{aligned}
Branch_{8048104} &= OnBranch_{8048104} \rightarrow Block_{8048121} \rightarrow Branch_{8048124} \\
Branch_{8048121} &= Block_{8048121} \rightarrow Branch_{8048124} \\
Branch_{8048115} &= OnBranch_{8048115} \rightarrow Call \rightarrow Return \\
&\quad \rightarrow Block_{804811a} \rightarrow Branch_{804811d} \mid\mid max \\
Branch_{8048128} &= (OnBranch_{8048128} \rightarrow Block_{8048106} \rightarrow Branch_{8048109}) \\
&\quad \Box\ (NoBranch_{8048128} \rightarrow Block_{804812a} \rightarrow Branch_{804812d}) \\
Branch_{80480ee} &= Block_{80480ee} \rightarrow Branch_{80480f0} \\
Branch_{80480f0} &= Block_{80480f0} \rightarrow Branch_{80480f3} \\
Branch_{80480f3} &= Block_{80480f3} \rightarrow Branch_{80480f6} \\
Branch_{80480f6} &= Block_{80480f6} \rightarrow Branch_{80480f8} \\
Branch_{80480f8} &= Block_{80480f8} \rightarrow Branch_{80480fb} \\
Branch_{80480fb} &= Block_{80480fb} \rightarrow Branch_{80480fe} \\
Branch_{80480fe} &= Block_{80480fe} \rightarrow Branch_{8048101} \\
Branch_{8048101} &= Block_{8048101} \rightarrow Branch_{8048104} \\
Branch_{8048109} &= Block_{8048109} \rightarrow Branch_{804810b} \\
Branch_{804810b} &= Block_{804810b} \rightarrow Branch_{804810e} \\
Branch_{804810e} &= Block_{804810e} \rightarrow Branch_{8048112} \\
Branch_{8048112} &= Block_{8048112} \rightarrow Branch_{8048115} \\
Branch_{804811d} &= Block_{804811d} \rightarrow Branch_{8048121} \\
Branch_{8048124} &= Block_{8048124} \rightarrow Branch_{8048128} \\
Branch_{804812d} &= Block_{804812d} \rightarrow Leave \\
Main &= Entry \rightarrow Block_{80480ed} \rightarrow Branch_{80480ee} \\
Leave &= Exit \rightarrow Stop
\end{aligned}
$$

**Entry**

$call? : System$

$\Theta\ System' = call?$

**Exit**

$return! : System$

$\Theta\ System = return!$

**Call**

$call! : System$

$\Theta\ System = call!$

**Return**

$return? : System$

$\Theta\ System' = return?$

**$Block_{80480ed}$**

$\Delta\ System$

$registers' = registers \oplus \{esp \mapsto registers(esp) - 4\}$
$memory' = memory \oplus \{registers'(esp) \mapsto registers(ebp)\}$

$Block_{80480ee}$
$\Delta\ System$

$registers' = registers \oplus \{ebp \mapsto registers(esp)\}$
$memory' = memory$

$Block_{80480f0}$
$\Delta\ System$

$registers' = registers \oplus \{esp \mapsto registers(esp) - 24\}$
$memory' = memory$
$zflag' = 1 \Leftrightarrow registers'(esp) = 0$

$Block_{80480f3}$
$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 8)\}$
$memory' = memory$

$Block_{80480f6}$
$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(eax) + 0)\}$
$memory' = memory$

$Block_{80480f8}$
$\Delta\ System$

$memory' = memory \oplus \{ebp - 8 \mapsto registers(eax)\}$
$registers' = registers$

$Block_{80480fb}$
$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 8)\}$
$memory' = memory$

$Block_{80480fe}$
$\Delta\ System$

$memory' = memory \oplus \{4 \mapsto registers(eax)\}$
$registers' = registers$

$Block_{8048101}$
$\Delta\ System$

$memory' = memory \oplus \{ebp - 4 \mapsto registers(eax)\}$
$registers' = registers$

$OnBranch_{8048104}$

$\Xi\ System$

$true$

$Block_{8048106}$

$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + -4)\}$
$memory' = memory$

$Block_{8048109}$

$\Delta\ System$

$registers' = registers \oplus \{eax \mapsto memory(registers(eax) + 0)\}$
$memory' = memory$

$Block_{804810b}$

$\Delta\ System$

$registers' = registers \oplus \{edx \mapsto memory(registers(ebp) + -8)\}$
$memory' = memory$

$Block_{804810e}$

$\Delta\ System$

$memory' = memory \oplus \{esp + 4 \mapsto registers(edx)\}$
$registers' = registers$

$Block_{8048112}$

$\Delta\ System$

$memory' = memory \oplus \{esp \mapsto registers(eax)\}$
$registers' = registers$

$OnBranch_{8048115}$

$\Delta\ System$

$registers' = registers \oplus \{esp \mapsto registers(esp) - 4\}$
$memory' = memory \oplus \{registers'(esp) \mapsto registers(ip)\}$

$Block_{804811a}$

$\Delta\ System$

$memory' = memory \oplus \{ebp - 8 \mapsto registers(eax)\}$
$registers' = registers$

$Block_{804811d}$

$\Delta\ System$

$$memory' = memory \oplus \{ebp \mapsto memory(registers(ebp) + -4) + 4\}$$
$$registers' = registers$$

$Block_{8048121}$

$\Delta\ System$

$$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + -4)\}$$
$$memory' = memory$$

$Block_{8048124}$

$\Delta\ System$

$$registers' = registers \oplus \{eax \mapsto memory(registers(eax) + 0)\}$$
$$memory' = memory$$
$$BITAND(eax, eax) = 0 \Rightarrow zflag' = 1$$
$$BITAND(eax, eax) \neq 0 \Rightarrow zflag' = 0$$
$$cflag' = 0$$
$$oflag' = 0$$

$OnBranch_{8048128}$

$\Xi\ System$

$$zflag = 1$$

$NoBranch_{8048128}$

$\Xi\ System$

$$zflag \neq 1$$

$Block_{804812a}$

$\Delta\ System$

$$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + -8)\}$$
$$memory' = memory$$

$Block_{804812d}$

$\Delta\ System$

$$registers' = registers \oplus$$
$$\{esp \mapsto registers(ebp) + 4, ebp \mapsto memory(registers(ebp))\}$$
$$memory' = memory$$

# Appendix C

# MaxInt for import to Z2SAL

$[CSPSTATE ::=$
$Branch80480e1 \mid Anon2 \mid Branch80480d9 \mid Branch80480db \mid$
$Branch80480de \mid Anon1 \mid Anon0 \mid Branch80480eb \mid Main \mid$
$Leave \mid Branch80480e6 \mid Stop]$
$[REGNAME ::= eax \mid ebx \mid ecx \mid edx \mid esp \mid ebp \mid ip]$

*System*

$registers : REGNAME \rightarrow \mathbb{N}$
$memory : \mathbb{N} \rightarrow \mathbb{N}$
$zflag : \{0, 1\}$
$cflag : \{0, 1\}$
$oflag : \{0, 1\}$
$sflag : \{0, 1\}$

*Init*

$System'$

---

$registers'(esp) = 4$
$registers'(ebp) = 12$

*Block80480d8*

$\Delta\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

---

$registers' = registers \oplus \{esp \mapsto registers(esp) - 4\}$

$memory' = memory \oplus \{registers'(esp) \mapsto registers(ebp)\}$
$cspstate = Anon2$
$cspstate' = Branch80480d9$

**Block80480d9**

$\Delta\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

$registers' = registers \oplus \{ebp \mapsto registers(esp)\}$
$memory' = memory$
$cspstate = Branch80480d9$
$cspstate' = Branch80480db$

**Block80480db**

$\Delta\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 8)\}$
$memory' = memory$
$cspstate = Branch80480db$
$cspstate' = Branch80480de$

**Block80480de**

$\Delta\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

$registers(eax) - memory(registers(ebp) + 12) < 0$
$\quad \Rightarrow sflag' = 1 \land zflag' = 0$
$registers(eax) - memory(registers(ebp) + 12) = 0$
$\quad \Rightarrow zflag' = 1 \land sflag' = 0$
$registers(eax) - memory(registers(ebp) + 12) > 0$
$\quad \Rightarrow zflag' = 0 \land sflag' = 0$
$memory' = memory$
$registers' = registers$
$cspstate = Branch80480de$
$cspstate' = Branch80480e1$

**OnBranch80480e1**

$\Xi\ System$
$cspstate : CSPSTATE$

$cspstate' : CSPSTATE$

$zflag = 1 \lor sflag = 1$
$cspstate = Branch80480e1$
$cspstate' = Anon0$

**$NoBranch80480e1$**

$\Xi\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

$zflag \neq 1 \land sflag \neq 1$
$cspstate = Branch80480e1$
$cspstate' = Anon1$

**$Block80480e3$**

$\Delta\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 8)\}$
$memory' = memory$
$cspstate = Anon1$
$cspstate' = Branch80480e6$

**$OnBranch80480e6$**

$\Xi\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

$true$
$cspstate = Branch80480e6$
$cspstate' = Branch80480eb$

**$Block80480e8$**

$\Delta\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

$registers' = registers \oplus \{eax \mapsto memory(registers(ebp) + 12)\}$
$memory' = memory$
$cspstate = Anon0$
$cspstate' = Branch80480eb$

$Block80480eb$

$\Delta\ System$
$cspstate : CSPSTATE$
$cspstate' : CSPSTATE$

---

$registers' = registers \oplus$
$\quad \{ebp \mapsto memory(registers(esp)), esp \mapsto registers(esp) + 4\}$
$memory' = memory$
$cspstate = Branch80480eb$
$cspstate' = Leave$

# Appendix D

# MaxInt SAL file

```
maxint : CONTEXT = BEGIN

NAT : TYPE = [0..14];

CSPSTATE : TYPE = DATATYPE
  Branch80480e1,
  Anon2,
  Branch80480d9,
  Branch80480db,
  Branch80480de,
  Anon1,
  Anon0,
  Branch80480eb,
  Main,
  Leave,
  Branch80480e6,
  Stop
END;

REGNAME : TYPE = DATATYPE
  eax,
  ebx,
  ecx,
  edx,
  esp,
  ebp,
  ip,
  REGNAME__B
END;

State : MODULE =
```

```
  BEGIN
    LOCAL registers : [REGNAME -> NAT]
    LOCAL memory : [NAT -> NAT]
    LOCAL zflag : [0..1]
    LOCAL cflag : [0..1]
    LOCAL oflag : [0..1]
    LOCAL sflag : [0..1]
    LOCAL cspstate : CSPSTATE
    LOCAL invariant__ : BOOLEAN
    DEFINITION
      invariant__ = (
        function {REGNAME, NAT; REGNAME__B, 14} ! total?(registers) AND
        function {NAT, NAT; 14, 14} ! total?(memory))
    INITIALIZATION [
        registers(esp) = 4 AND
        registers(ebp) = 12 AND
cspstate = Anon2 AND
        invariant__
      -->
    ]
    TRANSITION [
      Block80480d8 :
          registers' = function {REGNAME, NAT; REGNAME__B, 14} !
              insert(registers, (esp, registers(esp) - 4)) AND
          memory' = function {NAT, NAT; 14, 14} !
              insert(memory, (registers'(esp), registers(ebp))) AND
          cspstate = Anon2 AND
          cspstate' = Branch80480d9 AND
          invariant__'
        -->
          registers' IN {x : [REGNAME -> NAT] | TRUE};
          memory' IN {x : [NAT -> NAT] | TRUE};
          zflag' IN {x : [0..1] | TRUE};
          cflag' IN {x : [0..1] | TRUE};
          oflag' IN {x : [0..1] | TRUE};
          sflag' IN {x : [0..1] | TRUE};
          cspstate' IN {x : CSPSTATE | TRUE}
      []
      Block80480d9 :
          registers' = function {REGNAME, NAT; REGNAME__B, 14} !
              insert(registers, (ebp, registers(esp))) AND
          memory' = memory AND
          cspstate = Branch80480d9 AND
          cspstate' = Branch80480db AND
          invariant__'
        -->
```

```
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480db :
    registers' = function {REGNAME, NAT; REGNAME__B, 14} !
        insert(registers, (eax, memory(registers(ebp) + 8))) AND
    memory' = memory AND
    cspstate = Branch80480db AND
    cspstate' = Branch80480de AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480de :
    (registers(eax) - memory(registers(ebp) + 12) < 0 => (sflag' = 1 AND
        zflag' = 0)) AND
    (registers(eax) - memory(registers(ebp) + 12) = 0 => (zflag' = 1 AND
        sflag' = 0)) AND
    (registers(eax) - memory(registers(ebp) + 12) > 0 => (zflag' = 0 AND
        sflag' = 0)) AND
    memory' = memory AND
    registers' = registers AND
    cspstate = Branch80480de AND
    cspstate' = Branch80480e1 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
OnBranch80480e1 :
```

```
        (zflag = 1 OR sflag = 1) AND
        cspstate = Branch80480e1 AND
        cspstate' = Anon0 AND
        registers' = registers AND
        memory' = memory AND
        zflag' = zflag AND
        cflag' = cflag AND
        oflag' = oflag AND
        sflag' = sflag AND
        invariant__'
      -->
        registers' IN {x : [REGNAME -> NAT] | TRUE};
        memory' IN {x : [NAT -> NAT] | TRUE};
        zflag' IN {x : [0..1] | TRUE};
        cflag' IN {x : [0..1] | TRUE};
        oflag' IN {x : [0..1] | TRUE};
        sflag' IN {x : [0..1] | TRUE};
        cspstate' IN {x : CSPSTATE | TRUE}
    []
    NoBranch80480e1 :
        zflag /= 1 AND
        sflag /= 1 AND
        cspstate = Branch80480e1 AND
        cspstate' = Anon1 AND
        registers' = registers AND
        memory' = memory AND
        zflag' = zflag AND
        cflag' = cflag AND
        oflag' = oflag AND
        sflag' = sflag AND
        invariant__'
      -->
        registers' IN {x : [REGNAME -> NAT] | TRUE};
        memory' IN {x : [NAT -> NAT] | TRUE};
        zflag' IN {x : [0..1] | TRUE};
        cflag' IN {x : [0..1] | TRUE};
        oflag' IN {x : [0..1] | TRUE};
        sflag' IN {x : [0..1] | TRUE};
        cspstate' IN {x : CSPSTATE | TRUE}
    []
    Block80480e3 :
        registers' = function {REGNAME, NAT; REGNAME__B, 14} !
            insert(registers, (eax, memory(registers(ebp) + 8))) AND
        memory' = memory AND
        cspstate = Anon1 AND
        cspstate' = Branch80480e6 AND
```

```
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
OnBranch80480e6 :
    cspstate = Branch80480e6 AND
    cspstate' = Branch80480eb AND
    registers' = registers AND
    memory' = memory AND
    zflag' = zflag AND
    cflag' = cflag AND
    oflag' = oflag AND
    sflag' = sflag AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480e8 :
    registers' = function {REGNAME, NAT; REGNAME__B, 14} !
        insert(registers, (eax, memory(registers(ebp) + 12))) AND
    memory' = memory AND
    cspstate = Anon0 AND
    cspstate' = Branch80480eb AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480eb :
```

```
        registers' = function {REGNAME, NAT; REGNAME__B, 14} !
            insert(function {REGNAME, NAT; REGNAME__B, 14} !
            insert(registers, (ebp, memory(registers(esp)))), (esp,
            registers(esp) + 4)) AND
        memory' = memory AND
        cspstate = Branch80480eb AND
        cspstate' = Leave AND
        invariant__'
      -->
        registers' IN {x : [REGNAME -> NAT] | TRUE};
        memory' IN {x : [NAT -> NAT] | TRUE};
        zflag' IN {x : [0..1] | TRUE};
        cflag' IN {x : [0..1] | TRUE};
        oflag' IN {x : [0..1] | TRUE};
        sflag' IN {x : [0..1] | TRUE};
        cspstate' IN {x : CSPSTATE | TRUE}
    []
    Stop :
      cspstate = Leave AND
cspstate' = Leave AND
invariant__'
      -->
registers' = registers;
        memory' = memory;
        zflag' = zflag;
        cflag' = cflag;
        oflag' = oflag;
        sflag' = sflag
  ]
 END;

 alwaysTerminates : THEOREM State |- F(cspstate = Leave);
 neverRestarts : THEOREM State |-
     G((cspstate = Leave) =>
       NOT F(cspstate /= Leave));

 returnsSomethingSensible : THEOREM State |-
           G((cspstate = Leave) =>
             (
               (registers(eax) = memory(registers(esp) + 4))
             OR
               (registers(eax) = memory(registers(esp) + 8))
             )
           );

 returnsTheRightAnswer : THEOREM State |-
```

```
        G((cspstate = Leave) =>
          (
          (
            (registers(eax) = memory(registers(esp) + 4)) =>
            memory(registers(esp) + 4) >= memory(registers(esp) + 8)
          )
          AND
          (
            (registers(eax) = memory(registers(esp) + 8))=>
            memory(registers(esp) + 8) >= memory(registers(esp) + 4)
          )
          )
        );

END
```

**Notes:** The *cmp* instruction that is represented in $Block80480de$ performs a subtraction with two natural number variables and then checks if the result is negative, zero, or positive. Clearly, there is a potential problem with this, since the negative results are not natural numbers. However, since `NAT` is defined as a set of numbers, and the result of the subtraction is never stored in a variable SAL is happy to retain the full numeric value of the expression and evaluate the rest of the statement properly.

# Appendix E

# SAL suite counter example for brokenmaxint

```
Counterexample:
========================
Path
========================
Step 0:
--- System Variables (assignments) ---
registers(eax) = 4
registers(ebx) = 13
registers(ecx) = 6
registers(edx) = 11
registers(esp) = 4
registers(ebp) = 12
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 6
memory(1) = 10
memory(2) = 7
memory(3) = 13
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
memory(13) = 2
```

```
memory(14) = 14
zflag = 0
cflag = 1
oflag = 0
sflag = 1
cspstate = Anon2
invariant__ = true
------------------------
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480d8
    transition at [Context: brokenmaxint, line(54), column(10)]))
------------------------
Step 1:
--- System Variables (assignments) ---
registers(eax) = 4
registers(ebx) = 13
registers(ecx) = 6
registers(edx) = 11
registers(esp) = 0
registers(ebp) = 12
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 12
memory(1) = 10
memory(2) = 7
memory(3) = 13
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
memory(13) = 2
memory(14) = 14
zflag = 1
cflag = 0
oflag = 1
sflag = 1
cspstate = Branch80480d9
invariant__ = true
------------------------
Transition Information:
```

```
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480d9
    transition at [Context: brokenmaxint, line(71), column(10)]))
------------------------
Step 2:
--- System Variables (assignments) ---
registers(eax) = 4
registers(ebx) = 13
registers(ecx) = 6
registers(edx) = 11
registers(esp) = 0
registers(ebp) = 0
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 12
memory(1) = 10
memory(2) = 7
memory(3) = 13
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
memory(13) = 2
memory(14) = 14
zflag = 1
cflag = 0
oflag = 1
sflag = 0
cspstate = Branch80480db
invariant__ = true
------------------------
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480db
    transition at [Context: brokenmaxint, line(87), column(10)]))
------------------------
Step 3:
--- System Variables (assignments) ---
registers(eax) = 13
registers(ebx) = 13
registers(ecx) = 6
```

```
registers(edx) = 11
registers(esp) = 0
registers(ebp) = 0
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 12
memory(1) = 10
memory(2) = 7
memory(3) = 13
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
memory(13) = 2
memory(14) = 14
zflag = 0
cflag = 1
oflag = 1
sflag = 0
cspstate = Branch80480de
invariant__ = true
------------------------
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480de
    transition at [Context: brokenmaxint, line(103), column(11)]))
------------------------
Step 4:
--- System Variables (assignments) ---
registers(eax) = 13
registers(ebx) = 13
registers(ecx) = 6
registers(edx) = 11
registers(esp) = 0
registers(ebp) = 0
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 12
memory(1) = 10
memory(2) = 7
memory(3) = 13
```

```
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
memory(13) = 2
memory(14) = 14
zflag = 0
cflag = 0
oflag = 0
sflag = 0
cspstate = Branch80480e1
invariant__ = true
------------------------
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label OnBranch80480e1
    transition at [Context: brokenmaxint, line(124), column(10)]))
------------------------
Step 5:
--- System Variables (assignments) ---
registers(eax) = 13
registers(ebx) = 13
registers(ecx) = 6
registers(edx) = 11
registers(esp) = 0
registers(ebp) = 0
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 12
memory(1) = 10
memory(2) = 7
memory(3) = 13
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
```

```
memory(13) = 2
memory(14) = 14
zflag = 0
cflag = 0
oflag = 0
sflag = 0
cspstate = Anon0
invariant__ = true
------------------------
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480e8
    transition at [Context: brokenmaxint, line(199), column(10)]))
------------------------
Step 6:
--- System Variables (assignments) ---
registers(eax) = 12
registers(ebx) = 13
registers(ecx) = 6
registers(edx) = 11
registers(esp) = 0
registers(ebp) = 0
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 12
memory(1) = 10
memory(2) = 7
memory(3) = 13
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
memory(13) = 2
memory(14) = 14
zflag = 1
cflag = 0
oflag = 0
sflag = 0
cspstate = Branch80480eb
invariant__ = true
------------------------
```

```
Transition Information:
(module instance at [Context: brokenmaxint, line(257), column(34)]
  (label Block80480eb
    transition at [Context: brokenmaxint, line(215), column(10)]))
------------------------
Step 7:
--- System Variables (assignments) ---
registers(eax) = 12
registers(ebx) = 13
registers(ecx) = 6
registers(edx) = 11
registers(esp) = 4
registers(ebp) = 12
registers(ip) = 10
registers(REGNAME__B) = 14
memory(0) = 12
memory(1) = 10
memory(2) = 7
memory(3) = 13
memory(4) = 10
memory(5) = 9
memory(6) = 8
memory(7) = 11
memory(8) = 13
memory(9) = 1
memory(10) = 0
memory(11) = 0
memory(12) = 12
memory(13) = 2
memory(14) = 14
zflag = 0
cflag = 1
oflag = 0
sflag = 0
cspstate = Leave
invariant__ = true
```

# Appendix F

# Isabelle Z theory file

```
theory Z
imports Main
begin

record 'a state =
  sys :: 'a
  cond :: bool

type_synonym 'a Op = "'a state ⇒ 'a state"
type_synonym 'a Poss = "'a state list"
type_synonym 'a PossOp = "'a state ⇒ 'a Poss"
type_synonym 'a AllPosOp = "'a Poss ⇒ 'a Poss"

(* Some verification shortcuts *)
fun Good :: "'a state ⇒ bool" where
"Good S = cond S"

fun Bad :: "'a state ⇒ bool" where
"Bad S = (¬ (cond S))"

(* Convert plain Ops to AllPosOp *)
definition AllPos :: "'a Op ⇒ 'a AllPosOp" where
"AllPos a ≡ λS::'a state list . filter Good (map a S)"
declare AllPos_def [simp]

definition Semi :: "['a AllPosOp,'a AllPosOp] ⇒ 'a AllPosOp" (infixl ";" 85) where
"a;b ≡ λS::'a state list . b (a S)"
declare Semi_def [simp]

definition Alt :: "['a AllPosOp,'a AllPosOp] ⇒ 'a AllPosOp" (infixl "□" 95) where
```

```
"a□b ≡ λS . concat [(a S),(b S)]"
declare Alt_def [simp]

lemma goodChoice [simp] :
"∀S::'a state . ∀op1::'a Op . ∀op2::'a Op .
¬(cond (op2 S)) ⟶ (((AllPos op1)□(AllPos op2)) [S]) = filter Good [op1 S]"
by simp

lemma goodComp [simp] :
"∀op1::'a Op . ∀op2::'a Op . ∀S::'a state .
¬(cond (op1 S)) ⟶ (((AllPos op1);(AllPos op2)) [S]) = []"
by simp

lemma veryGoodComp :
"∀op1::'a Op . ∀op2::'a Op . ∀S::'a state .
(cond (op1 S)) ⟶ (((((AllPos op1);(AllPos op2)) [S]) = [op2 (op1 S)])
⟶ cond (op2 (op1 S)))"
by simp

lemma goodCompA [simp] :
"∀op1::'a Op . ∀op2::'a Op . ∀S::'a state .
¬(cond (op1 S)) ⟶ (((AllPos op1);(AllPos op2)) [S]) = []"
by simp

lemma associativityOfSemi [simp] :
"∀St . ∀op1::'a Op . ∀op2::'a Op . ∀op3::'a Op .
((AllPos op1);(AllPos op2);(AllPos op3)) St =
(((AllPos op1);(AllPos op2));(AllPos op3)) St"
by simp

lemma AllPosRequiresGood [simp] : "∀St . ∀op1::'a Op .
((AllPos op1) St) = filter Good ((AllPos op1) St)"
by (simp del: Good.simps)


end
```

# Appendix G

# Max example Isabelle theory file

```
theory Max
imports Z
begin

declare [[ smt_solver = remote_z3]]
declare [[ smt_timeout = 60 ]]
declare [[ z3_options = "-memory:500" ]]

datatype Regname = eax | ebx | ecx | edx | esp | ebp | ip

datatype bit = zero | one

record system =
  cflag :: bit
  zflag :: bit
  sflag :: bit
  memory :: "int ⇒ int"
  registers :: "Regname ⇒ int"

(* Some helpful shortcuts for ASM programs... *)

definition mem :: "system state ⇒ int ⇒ int" where
"mem St ≡ (memory (sys St))"
declare mem_def [simp]

definition reg :: "system state ⇒ Regname ⇒int" where
"reg St ≡ (registers (sys St))"
```

```
declare reg_def [simp]

definition regind :: "system state ⇒ Regname => int ⇒ int" where
"regind St ≡ λr . λoff . ((mem St) ((reg St r) + off))"
declare regind_def [simp]

(* Z Operation schema converted from the Max example *)

definition Block80480d8 :: "system Op" where
"Block80480d8 St =
  (|sys=(sys St)(|
  registers := (registers (sys St))(esp := ((reg St esp) - 4)),
  memory := (memory (sys St))(((reg St esp) - 4) := (reg St ebp))|)
  , cond=True|)"
declare Block80480d8_def [simp]

definition Block80480d9 :: "system Op" where
"Block80480d9 St =
  (|sys=(sys St)(|
  registers := (registers (sys St))(ebp := reg St esp),
  memory := (memory (sys St))|)
  , cond=True|)"
declare Block80480d9_def [simp]

definition Block80480db :: "system Op" where
"Block80480db St =
  (|sys=(sys St)(|
    registers := ((registers (sys St))(eax :=
      (memory (sys St)) (((registers (sys St)) ebp) + 8)))|)
    , cond=True|)"
declare Block80480db_def [simp]

lemma firstThreeNoPrecond [simp] : "∀St::(system state) .
    ¬ ((((AllPos Block80480d8)
    ;(AllPos Block80480d9)
    ;(AllPos Block80480db)) [St]) = [])"
by simp

definition Block80480dePartA :: "system Op" where
"Block80480dePartA St = (|sys=(sys St)(|zflag:=zero,sflag:=one|)
  , cond=((((reg St) eax) - (regind St ebp 12)) < 0)|)"
declare Block80480dePartA_def [simp]

definition Block80480dePartB :: "system Op" where
"Block80480dePartB St = (|sys=(sys St)(|zflag:=one,sflag:=zero|)
  , cond=((((reg St) eax) - (regind St ebp 12)) = 0)|)"
```

```
declare Block80480dePartB_def [simp]

definition Block80480dePartC :: "system Op" where
"Block80480dePartC St = (|sys=(sys St)(|zflag:=zero,sflag:=zero|)
  , cond=((((reg St) eax) - (regind St ebp 12)) > 0)|)"
declare Block80480dePartC_def [simp]

definition Block80480de :: "system AllPosOp" where
"Block80480de St = (((AllPos Block80480dePartA)
                   □(AllPos Block80480dePartB))
                   □(AllPos Block80480dePartC)) St"
declare Block80480de_def [simp]

definition OnBranch80480e1 :: "system Op" where
"OnBranch80480e1 St = (|sys=(sys St),
  cond=((zflag (sys St) = one) ∨ (sflag (sys St) = one))|)"
declare OnBranch80480e1_def [simp]

definition NoBranch80480e1 :: "system Op" where
"NoBranch80480e1 St = (|sys=(sys St),
  cond=((zflag (sys St) ≠ one) ∧ (sflag (sys St) ≠ one))|)"
declare NoBranch80480e1_def [simp]

(* Choices should be total - i.e. there should be at least one right answer,
and in this model there should be exactly one right answer *)

lemma TotalBranch80480e1 : "∀St . ∃S .
    hd (((AllPos OnBranch80480e1)
    □(AllPos NoBranch80480e1)) [St]) = S"
by simp

lemma OnlyOneBranch80480e1 :
"∀St . (cond (OnBranch80480e1 St) ⟷
¬ cond (NoBranch80480e1 St))"
by simp

definition Block80480e3 :: "system Op" where
"Block80480e3 St = (|sys = (sys St)(|
    registers:=(registers (sys St))(eax:=
      (memory (sys St))(((registers (sys St)) ebp) + 8))|),
  cond = True|)"
declare Block80480e3_def [simp]

definition OnBranch80480e6 :: "system Op" where
"OnBranch80480e6 St = (|sys = sys St, cond = True |)"
declare OnBranch80480e6_def [simp]
```

```
(* OnBranch80480e6 always occurs and does nothing
to the system state (its a JMP command...) *)
lemma OnBranch80480e6Simple : "∀St .
Good St ⟶ OnBranch80480e6 St = St"
by simp

definition Block80480e8 :: "system Op" where
"Block80480e8 St = ⦇sys = (sys St)⦇
  registers:=(registers (sys St))(eax:=
    (memory (sys St))(((registers (sys St)) ebp) + 12))⦈,
  cond = True⦈"
declare Block80480e8_def [simp]

definition Block80480eb :: "system Op" where
"Block80480eb St = ⦇sys = (sys St)⦇
  registers:=(registers (sys St))(
  esp:=(reg St esp) + 4,
  ebp:=regind St esp 0
  )⦈,
  cond = True⦈"
declare Block80480eb_def [simp]

(* The first sequential blocks with no branches *)
definition Entry :: "system AllPosOp" where
"Entry ≡ λSt .
   ((AllPos Block80480d8)
   ;(AllPos Block80480d9)
   ;(AllPos Block80480db)
   ;Block80480de) St"
declare Entry_def [simp]

(* The two choices at 189 *)
definition Left :: "system AllPosOp" where
"Left ≡ λSt .
   ((AllPos OnBranch80480e1)
   ;(AllPos Block80480e8)
   ;(AllPos Block80480eb)) St"
declare Left_def [simp]

definition Right :: "system AllPosOp" where
"Right ≡ λSt .
   ((AllPos NoBranch80480e1)
   ;(AllPos Block80480e3)
   ;(AllPos OnBranch80480e6)
   ;(AllPos Block80480eb)) St"
```

```
declare Right_def [simp]

(* The complete spec for the Max function *)
definition MaxOp :: "system AllPosOp" where
"MaxOp St = (Entry;(Left□Right)) St"
declare MaxOp_def [simp]

(* There is a right answer... *)
lemma EntryTermiintes : "∀St::system state .
   Good St ⟶ Entry [St] ≠ []"
by simp

lemma MaxTermiintes : "∀St::system state .
   MaxOp [St] ≠ []"
by simp

(* I can’t find the Isabelle List version of this... *)
fun cont :: "’a ⇒ ’a list ⇒ bool" where
"cont val [] = False" |
"cont val list = ((val = (hd list)) ∨ (cont val (tl list)))"

lemma NoStackOverflow :
  "∀St::system state .
  ∀ASt::system state .
  cont ASt (MaxOp [St]) ⟶
  (regind St esp 0 = regind ASt esp 0)
"
by simp


(* Helpful, if obvious lemma *)
lemma functionsAreSane [simp] : "∀St . ∀n . ((mem St) n = (mem St) n)"
by simp

(* This must be a int rule somewhere... *)
lemma intCommute [simp] : "∀n::int . ∀m::int . (n + m) = (m + n)"
by simp

lemma MaxGetsSomethingSensible [simp] :
  "∀St::system state .
  ∀ASt::system state .
  cont ASt (MaxOp [St]) ⟶
  (
  let n = (regind St esp 4); m = (regind St esp 8) in
  ((reg ASt eax) = n) ∨ ((reg ASt eax) = m)
  )"
```

```
by simp

(* We should get the right answer - i.e. the value returned
should be the greater of the two. Since the previous lemma
shows that it is either the value at esp+4 or esp+8 we can just say
that it is esp+4 iff that is the larger (or they are equal) *)

lemma MaxReturnsMax : "∀St::system state . ∀ASt::system state .
  cont ASt (MaxOp [St]) ⟶
  (
  let n = (regind St esp 4); m = (regind St esp 8) in
  (n ≥ m ⟷ (reg ASt eax = n))
  )"
by simp

(* Changes in the broken version... *)
definition WrongOnBranch80480e1 :: "system Op" where
"WrongOnBranch80480e1 St = ⦇sys=(sys St),
  cond=(sflag (sys St) = zero)⦈"
declare WrongOnBranch80480e1_def [simp]

definition WrongNoBranch80480e1 :: "system Op" where
"WrongNoBranch80480e1 St = ⦇sys=(sys St),
  cond=(sflag (sys St) ≠ zero)⦈"
declare WrongNoBranch80480e1_def [simp]

lemma TotalWrongBranch80480e1 : "∀St . ∃S .
  hd (((AllPos WrongOnBranch80480e1)
  □(AllPos WrongNoBranch80480e1)) [St]) = S"
by simp

lemma OnlyOneWrongBranch80480e1 : "∀St .
  (cond (WrongOnBranch80480e1 St) ⟷
  ¬ cond (WrongNoBranch80480e1 St))"
by simp

(* The two wrong choices at 189 *)
definition WrongLeft :: "system AllPosOp" where
"WrongLeft ≡ λSt .
   ((AllPos WrongOnBranch80480e1)
   ;(AllPos Block80480e8)
   ;(AllPos Block80480eb)) St"
declare WrongLeft_def [simp]

definition WrongRight :: "system AllPosOp" where
"WrongRight ≡ λSt .
```

```
   ((AllPos WrongNoBranch80480e1)
   ;(AllPos Block80480e3)
   ;(AllPos OnBranch80480e6)
   ;(AllPos Block80480eb)) St"
declare WrongRight_def [simp]

(* The complete spec for the Max function *)
definition WrongMaxOp :: "system AllPosOp" where
"WrongMaxOp St = (Entry;(WrongLeft□WrongRight)) St"
declare WrongMaxOp_def [simp]

(* It should still get an answer *)
lemma WrongMaxTermiintes : "∀St::system state . WrongMaxOp [St] ≠ []"
by simp

(* We should still get one of the two arguments... *)
lemma WrongMaxGetsSomethingSensible [simp] :
  "∀St::system state .
  ∀ASt::system state .
  cont ASt (WrongMaxOp [St]) ⟶
  (let n = (regind St esp 4); m = (regind St esp 8) in
  (((reg ASt eax) = n) ∨ ((reg ASt eax) = m))
  )"
by simp

(* We should not get the right answer - i.e. the value returned
should not be the greater of the two - so this should fail to
prove and the counter example should be an informative description
of the problem. *)

theorem WrongMaxReturnsMax : "∀St::system state . ∀ASt::system state .
  (
  (cont ASt (WrongMaxOp [St])) ⟶
  (let n = (regind St esp 4); m = (regind St esp 8) in
  (n ≥ m) ⟷ (reg ASt eax = n)
  )
  )
  "
nitpick
apply(simp)
nitpick [verbose=true,show_all=true]

end
```

# Appendix H

# Exdev SAL file

```
exdev : CONTEXT = BEGIN

NAT : TYPE = [0..22];

CSPSTATE : TYPE = DATATYPE
  Branch80480de,
  Anon1,
  Branch80480b9,
  Branch80480bb,
  Branch80480be,
  Branch80480c3,
  Branch80480c7,
  Branch80480ca,
  Branch80480cd,
  Branch80480d0,
  Branch80480d3,
  Branch80480d5,
  Branch80480d8,
  Branch80480db,
  Anon0,
  Branch80480e2,
  Branch80480e5,
  Branch80480e8,
  Main,
  Leave,
  Stop
END;

REGNAME : TYPE = DATATYPE
  eax,
  ebx,
```

```
  ecx,
  edx,
  esp,
  ebp,
  ip,
  REGNAME__B
END;

State : MODULE =
  BEGIN
    LOCAL registers : [REGNAME -> NAT]
    LOCAL memory : [NAT -> NAT]
    LOCAL ports : [NAT -> NAT]
    LOCAL zflag : [0..1]
    LOCAL cflag : [0..1]
    LOCAL oflag : [0..1]
    LOCAL sflag : [0..1]
    LOCAL cspstate : CSPSTATE
    LOCAL invariant__ : BOOLEAN
    DEFINITION
      invariant__ = (
        function {REGNAME, NAT; REGNAME__B, 22} ! total?(registers) AND
        function {NAT, NAT; 22, 22} ! total?(memory) AND
        function {NAT, NAT; 22, 22} ! total?(ports))
    INITIALIZATION [
        cspstate = Anon1 AND
        registers(esp) = 20 AND
        registers(ebp) = 20 AND
        invariant__
      -->
    ]
    TRANSITION [
      Block80480b8 :
          registers' = function {REGNAME, NAT; REGNAME__B, 22} !
              insert(registers, (esp, registers(esp) - 4)) AND
          memory' = function {NAT, NAT; 22, 22} !
              insert(memory, (registers'(esp), registers(ebp))) AND
          cspstate = Anon1 AND
          cspstate' = Branch80480b9 AND
          invariant__'
        -->
          registers' IN {x : [REGNAME -> NAT] | TRUE};
          memory' IN {x : [NAT -> NAT] | TRUE};
          ports' IN {x : [NAT -> NAT] | TRUE};
          zflag' IN {x : [0..1] | TRUE};
          cflag' IN {x : [0..1] | TRUE};
```

```
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480b9 :
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (ebp, registers(esp))) AND
    memory' = memory AND
    cspstate = Branch80480b9 AND
    cspstate' = Branch80480bb AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480bb :
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (esp, registers(esp) - 16)) AND
    memory' = memory AND
    (zflag' = 1) = (registers'(esp) = 0) AND
    cspstate = Branch80480bb AND
    cspstate' = Branch80480be AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480be :
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, 1)) AND
    memory' = memory AND
    cspstate = Branch80480be AND
    cspstate' = Branch80480c3 AND
    invariant__'
```

```
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480c3 :
    memory' = memory AND
    ports(0) = registers(eax) AND
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, ports(8))) AND
    cspstate = Branch80480c3 AND
    cspstate' = Branch80480c7 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480c7 :
    memory' = function {NAT, NAT; 22, 22} !
        insert(memory, (registers(ebp) - 16, registers(eax))) AND
    registers' = registers AND
    cspstate = Branch80480c7 AND
    cspstate' = Branch80480ca AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480ca :
```

```
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, memory(registers(ebp) - 16))) AND
    memory' = memory AND
    cspstate = Branch80480ca AND
    cspstate' = Branch80480cd AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480cd :
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, registers(eax) + 10)) AND
    memory' = memory AND
    cspstate = Branch80480cd AND
    cspstate' = Branch80480d0 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480d0 :
    memory' = function {NAT, NAT; 22, 22} !
        insert(memory, (registers(ebp) - 12, registers(eax))) AND
    registers' = registers AND
    cspstate = Branch80480d0 AND
    cspstate' = Branch80480d3 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
```

```
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480d3 :
    memory' = memory AND
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, ports(8))) AND
    cspstate = Branch80480d3 AND
    cspstate' = Branch80480d5 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480d5 :
    memory' = function {NAT, NAT; 22, 22} !
        insert(memory, (registers(ebp) - 8, registers(eax))) AND
    registers' = registers AND
    cspstate = Branch80480d5 AND
    cspstate' = Branch80480d8 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480d8 :
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, memory(registers(ebp) - 8))) AND
    memory' = memory AND
    cspstate = Branch80480d8 AND
    cspstate' = Branch80480db AND
    invariant__'
  -->
```

```
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480db :
    (registers(eax) - memory(registers(ebp) - 12) < 0 => (sflag' = 1 AND
        zflag' = 0)) AND
    (registers(eax) - memory(registers(ebp) - 12) = 0 => (zflag' = 1 AND
        sflag' = 0)) AND
    (registers(eax) - memory(registers(ebp) - 12) > 0 => (zflag' = 0 AND
        sflag' = 0)) AND
    memory' = memory AND
    registers' = registers AND
    cspstate = Branch80480db AND
    cspstate' = Branch80480de AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
OnBranch80480de :
    sflag = 1 AND
    cspstate = Branch80480de AND
    cspstate' = Branch80480d3 AND
    registers' = registers AND
    memory' = memory AND
    ports' = ports AND
    zflag' = zflag AND
    cflag' = cflag AND
    oflag' = oflag AND
    sflag' = sflag AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
```

```
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
NoBranch80480de :
    sflag /= 1 AND
    cspstate = Branch80480de AND
    cspstate' = Anon0 AND
    registers' = registers AND
    memory' = memory AND
    ports' = ports AND
    zflag' = zflag AND
    cflag' = cflag AND
    oflag' = oflag AND
    sflag' = sflag AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480e0 :
    memory' = memory AND
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, ports(4))) AND
    cspstate = Anon0 AND
    cspstate' = Branch80480e2 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
```

```
Block80480e2 :
    memory' = function {NAT, NAT; 22, 22} !
        insert(memory, (registers(ebp) - 4, registers(eax))) AND
    registers' = registers AND
    cspstate = Branch80480e2 AND
    cspstate' = Branch80480e5 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480e5 :
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (eax, memory(registers(ebp) - 4))) AND
    memory' = memory AND
    cspstate = Branch80480e5 AND
    cspstate' = Branch80480e8 AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
    ports' IN {x : [NAT -> NAT] | TRUE};
    zflag' IN {x : [0..1] | TRUE};
    cflag' IN {x : [0..1] | TRUE};
    oflag' IN {x : [0..1] | TRUE};
    sflag' IN {x : [0..1] | TRUE};
    cspstate' IN {x : CSPSTATE | TRUE}
[]
Block80480e8 :
    registers' = function {REGNAME, NAT; REGNAME__B, 22} !
        insert(function {REGNAME, NAT; REGNAME__B, 22} !
        insert(registers, (esp, registers(ebp) + 4)), (ebp,
        memory(registers(ebp)))) AND
    memory' = memory AND
    cspstate = Branch80480e8 AND
    cspstate' = Leave AND
    invariant__'
  -->
    registers' IN {x : [REGNAME -> NAT] | TRUE};
    memory' IN {x : [NAT -> NAT] | TRUE};
```

```
          ports' IN {x : [NAT -> NAT] | TRUE};
          zflag' IN {x : [0..1] | TRUE};
          cflag' IN {x : [0..1] | TRUE};
          oflag' IN {x : [0..1] | TRUE};
          sflag' IN {x : [0..1] | TRUE};
          cspstate' IN {x : CSPSTATE | TRUE}
      []
      Stop :
        cspstate = Leave AND
cspstate' = Leave AND
invariant__'
      -->
        registers' = registers;
        memory' = memory;
        ports' = ports;
        zflag' = zflag;
        cflag' = cflag;
        oflag' = oflag;
        sflag' = sflag
    ]
  END;


  neverTerminates : THEOREM State |- G(cspstate /= Leave);
  neverRestarts : THEOREM State |-
    G((cspstate = Leave) => NOT F(cspstate /= Leave));

  writeBeforeRead : THEOREM State |-
    W((cspstate /= Anon0),(cspstate = Branch80480c3));

  delayIsCorrect : THEOREM State |- FORALL (START:NAT):
        G(
        ((cspstate = Branch80480c3) AND (ports(8) = START)) =>
        FORALL (FINISH:NAT):
          G(((cspstate = Branch80480d3) AND (ports(8) = FINISH)) =>
            G(((cspstate = Anon0) AND (registers(eax) = FINISH)) =>
              (FINISH >= (START + 10))
            )
          )
        );


  delayBetweenWriteAndRead : THEOREM State |-
    W((cspstate /= Anon0), (cspstate = Branch80480d3));

END
```