Verification of hardware interaction properties of software

Ramsay Taylor

Department of Computer Science, The University of Sheffield

Abstract. Many high-integrity software development processes prevent any assumptions about the system hardware, but this makes it impossible to use these techniques on software that must interact with the hardware, such as device drivers. This work takes the opposite approach: if the analyst accepts that the analysis will only be valid for a particular target system 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 specifications of the target platform to produce CSP-OZ formal models of the software's behaviour. This analysis process is implemented in a prototype called Spurinna. This is demonstrated in conjunction with the verification tools Z2SAL and the SAL suite to demonstrate the verification of properties of an example program.

1 Introduction

Many projects make use of static analysis to give a measure of assurance for the safe functioning of the code. To facilitate static analysis these projects are often based on "safe" language subsets such as MISRA-C [13] and SPARK Ada [1]. These use restricted versions of common programming languages to make the code behaviour determinable without knowledge of the context. However, by restricting the language they necessarily make themselves unusable for applications that rely on the features that have been removed or restricted, and prevent analysis of requirements that are defined in terms of a particular context. Hardware control applications are a significant instance of this, and so they are the focus for the work presented here.

Hardware control and interaction is an area that is central to many safety-critical systems as it is often the device control aspect that gives them the potential to cause harm. Many restricted languages remove any feature that interacts with the hardware, since these features prevent deterministic reasoning about the code without making assumptions about the behaviour of the hardware. However, in the case of device drivers it is reasonable 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.

Projects such as [12] and [3] have provided complete verification of system stacks, but these have required considerable manual effort that would have to be repeated for each application. The objective of this work is to provide a general process for verifying the behaviour of low level software that is independent of the particular underlying system, and that is sufficiently automatic that it can be easily and frequently repeated as part of a software development process.

The principal 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.

The following example of a hardware usage scenario is used in this paper:

> The device to be controlled has two ports: a control port and a data port. The control port is accessible at IO port 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 at least 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 a scale that increments once per ms. The system has an Intel i386 based processor.

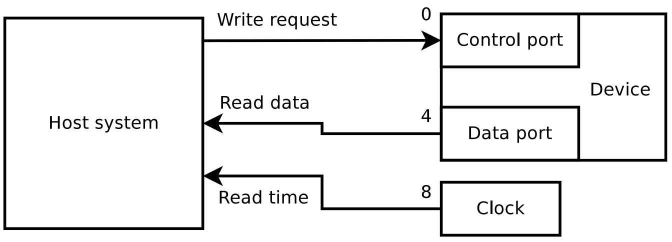

Fig. 1. The example device

A device driver that controls this device and presents the available data to a software system must ensure that the device is used in the required fashion. In this example that produces a set of specific behavioural properties that the device control software must satisfy:

- A request value of 1 must be written to the control port at address 0, before data is read from address 4.
- There must be a delay between the writing of this request and the reading of the data.
- The delay must be 10ms long. Specifically, the value present in the clock port at address 8 must have incremented by at least 10 between the writing of the request and the reading of the data

These properties make explicit statements about IO addresses, and about the sequence and content of interactions with these hardware features. The design of the formal model presented in Section 3 must support the specification of the combined software/hardware system such these details are present and properties of their use can be written easily.

1.1 Document outline

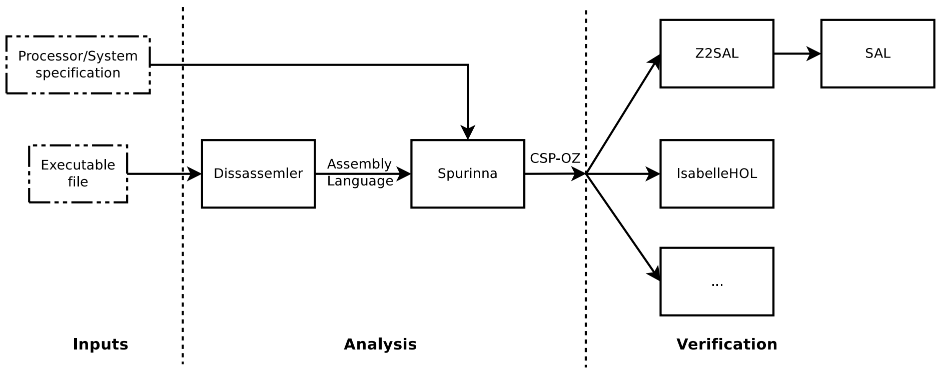

Fig. 2. The analysis process outline

The analysis process described in this paper has been implemented in a prototype tool named Spurinna[1]. The development of a formal system for interpreting executable file formats and performing disassembly was beyond the scope of this project, so GNU *objdump*[2] is used to convert the executable into assembly language and symbol information. Section 2 discusses the analysed programs and their disassembly. Spurinna takes this, and a supplied formal specification of the processor and target platform as inputs. Section 3 describes the model of the system that is required as input, the model of software behaviour that is produced by the analysis, and the design choices made to accommodate hardware details and low-level software features into a manageable, formal representation. From these inputs it is able to produce a formal model of the behaviour of the software entirely automatically. The process of automatically inferring models of this form from presented disassembly output is detailed in Section 4. This is output in CSP-OZ in LaTeX format, and can be used as input to any verification tools or techniques applicable to Z or CSP. This paper demonstrates verification of requirements using the Z2SAL tool [8] and the SAL model checking suite [7] in Section 5. Conclusions are presented in Section 6.

2 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.

This analysis process needs to operate on the level closest to execution, but 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

[1] http://staffwww.dcs.shef.ac.uk/people/R.Taylor/Spurinna/
[2] http://www.gnu.org/software/binutils/

ideal but the development of such a system is beyond the scope of this work. The object code produced by the assembler but before linking are also not suitable, since the linker makes a number of important decisions about the layout of the program in memory and about the resolution of symbols to absolute addresses and values. Consequently, it is the executable file that is as close as is practical and that are the source material for this analysis.

To illustrate the analysis process a program was developed to interact with the example device described in Section 1. The program was written in C and is shown in Figure 3.

In order to access the IO ports of the processor this program must use inline assembly code statements. This is a violation of the MISRA-C coding standards and is a good example of the impossibility of writing device driver code that stays within a safe language subset.

```
#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;
  out(CONTROL_REG, 1);
  in(CLOCK_REG, starttime);
  endtime = starttime + 10;
  do {
    in(CLOCK_REG, now);
  } while(now < endtime);
  in(DATA_REG, result);
  return result;
}
```

Fig. 3. A C program that implements the device control behaviour

This program was compiled with gcc, the GNU C compiler. The resulting executable file was then disassembled with GNU *objdump* to produce the output show in Figure 4.

This shows the format of assembly instructions that are presented to the following stages of analysis, as well as the symbol information that was extracted from the executable. In this case the `exdev` function from the C program has remained identifiable, beginning at address 08048094. If the file contained multiple functions then these would be separated and identified by name. This example contains only local branch instructions but where function call instructions are present their target addresses are identified and the name of their target functions included. For example, a trivial program to identify the largest integer in a list using a helper function that compares two integers and returns the larger might contain a call instruction of the form: `call 8048180 <max>`.

```
08048094 <exdev>:
 8048094: 55                    push   %ebp
 8048095: 89 e5                 mov    %esp,%ebp
 8048097: 83 ec 10              sub    $0x10,%esp
 804809a: b8 25 00 00 00        mov    $0x25,%eax
 804809f: c7 00 01 00 00 00     movl   $0x1,(%eax)
 80480a5: b8 2b 00 00 00        mov    $0x2b,%eax
 80480aa: 8b 00                 mov    (%eax),%eax
 80480ac: 89 45 f4              mov    %eax,-0xc(%ebp)
 80480af: 8b 45 f4              mov    -0xc(%ebp),%eax
 80480b2: 83 c0 64              add    $0x64,%eax
 80480b5: 89 45 f8              mov    %eax,-0x8(%ebp)
 80480b8: b8 2b 00 00 00        mov    $0x2b,%eax
 80480bd: 8b 00                 mov    (%eax),%eax
 80480bf: 89 45 fc              mov    %eax,-0x4(%ebp)
 80480c2: eb 0a                 jmp    80480ce <exdev+0x3a>
 80480c4: b8 2b 00 00 00        mov    $0x2b,%eax
 80480c9: 8b 00                 mov    (%eax),%eax
 80480cb: 89 45 fc              mov    %eax,-0x4(%ebp)
 80480ce: 8b 45 fc              mov    -0x4(%ebp),%eax
 80480d1: 3b 45 f8              cmp    -0x8(%ebp),%eax
 80480d4: 7c ee                 jl     80480c4 <exdev+0x30>
 80480d6: b8 26 00 00 00        mov    $0x26,%eax
 80480db: 8b 00                 mov    (%eax),%eax
 80480dd: c9                    leave
 80480de: c3                    ret
```

Fig. 4. The C program after compilation, assembly, linking, and disassembly

3 Behaviour model structure

Many current approaches to low-level software verification, such as Separation Logic [14] are able to verify properties about programs by creating suitable, abstract models of pointers, memory addresses, and other hardware interaction that are applicable across all contemporary computer systems. This allows these approaches to explore subtleties of program construction, such as self-modifying programs [5], that are not possible with the model presented here. However, the objective of this work is to create a model that deliberately does *not* abstract the implementation details of the hardware in any way, since it is aimed at verifying properties that make statements about specific hardware features.

The approach taken by this work is derived from the Z models of the state and operation of processors that have been produced since the 1980s [4, 11]. Using process calculi also has a long history [2]. This analysis process uses both approaches and separate the control flow components of the program from the state change instructions. CSP-OZ [10] combines Object-Z [6] with CSP such that the Object-Z defines classes with state and operations on that state, while the CSP defines the possible control flow paths through those operations.

CSP-OZ specifications contain four types of component: A system state specification, operation schema that describe the state altering behaviour of events, CSP processes that define the allowed sequences of events in the system, and

Object-Z classes that collect these components into an Object-Oriented framework.

Where adequate symbol information exists to identify functions in the code these are modeled as separate classes in the CSP-OZ model. This creates a model with a modularised structure that should aid comprehension.

The analysis process separates those instructions that alter control flow from those that do not. The former are referred to as *branch instructions*, while the latter are referred to as *sequential instructions*. Once the branch instructions have been separated, the remaining blocks of sequential instructions represent code that will all be executed if it is begun[3].

Branch instructions are further separated into *local* branch instructions, that alter control flow within a function, and *function call* and *function return* instructions that direct control flow to other identified functions. The distinction between the two is specified by the analysis user as part of the branch instruction set specification. Section 4.1 describes the process of separating the branch instructions from the sequential instructions to form a control flow graph. The nodes of this graph are the branch instructions, while the edges are the *sequential blocks* — the sequences of sequential instructions that contain neither a branch instruction, nor the target of a branch instruction, so are executed in sequence from start to finish. The behaviour of these sequential blocks is represented by the Z operations of the function's class in the CSP-OZ model. The local branch instructions are represented in the CSP part of the function's class definition, specifying the possible sequences of sequential blocks that can be executed. A conditional branch is encoded as an external choice between the two possible sequential blocks. To encode the decision procedure of the branch instructions two additional Z operation schema are added to the class that contain suitable precondition invariants. These operations are prefixed to the two possible sequential blocks such that the preconditions of each choice model the decision behaviour of the instruction.

The model represents the function call and return behaviour using a more abstract, OO notation to make the inferred model more readable and more clearly resemble the structure of the original code, insofar as this can be determined from the information in the executable file. Function call and return behaviour is modelled by running the called function's class in parallel with the calling class. The calling class passes the system state along a channel to the called class, which performs its function on the system state, and then passes the state back to the calling class. The calling class synchronises on these transactions, so does not proceed until the called function has returned, and uses the Z theta notation to replace its current state with that received from the called function. Section 4.4 describes the process of combining the components together into

[3] Interrupts could violate this assumption, but their behaviour is ignored here as many device drivers will be operating as interrupt handlers, or with interrupts disabled. Alternatively, the impact of interrupts could be represented by making sections of the system state *volatile*, that is, its state becomes unspecified between atomic operations.

a complete CSP-OZ representation of both the control flow and state change behaviours.

3.1 System state specification

The formal model produced by this work must contain adequate detail of both the software and hardware behaviour to allow the properties of interest to be verified. Even a simple computer system has considerable detail that could be included, but only parts of this are relevant to the verification of a particular set of requirements. Consequently, the analysis process developed here is deliberately independent of the system specification used.

A simple specification of an Intel i386 based platform could be presented thus:

$$
\begin{array}{l}
BIT == \{0, 1\} \\
INT32 == \{0..2^{32}\} \\
REGNAMES == \{eax, ebx, ecx, edx, esp, ebp\}
\end{array}
$$

System

$$
\begin{array}{|l}
memory : INT32 \mathrel{+\!\!\!\!\!\to} INT32 \\
registers : REGNAMES \to INT32 \\
ioports : INT32 \mathrel{+\!\!\!\!\!\to} INT32 \\
zf, cf, sf : BIT \\
\hline
\end{array}
$$

For a particular Intel-based platform this schema could be augmented with invariants — perhaps identifying sections of ROM, or memory-mapped devices. The register interellations of an Intel processor, where *al*, *ah*, *ax*, and *eax* all refer to different components of the same 32bit value can be clearly represented by invariants, for example. Only a subset of the processor status flags are included here, and only a subset of the valid register names, but these are adequate for the short example used.

3.2 Sequential instruction templates

In addition to the system state specification, the user of this analysis process must also provide two instruction set specifications. The anaylsis process is independent of the system specification used, and is independent of the effects of the instruction set specifications but it does require a standardised format for the instruction set descriptions. The sequential instructions must be specified as *template* Z operation schemas. These are standard Z schema with a specific naming convention: the name of the schema must be the mnemonic of the instruction it represents, with a subscript containing the type signature for which this template defines behaviour.

Processor instructions are often defined with the same mnemonic having subtly different behaviour for different types of parameter. The types of parameter recognised by this analysis process are *literal*, *register*, and *register indirect* (where the value in a register is used as an address into memory, possibly with an offset). These three types are clearly identifiable in *objdump*'s output. The Intel `mov` instruction, applied to load a literal value into a register, can be specified with this template:

$$
\begin{array}{|l}
\multicolumn{1}{l}{mov_{LIT\#SRC,REG\#TGT}} \\
\Delta\ System \\
\hline
registers' = registers \oplus TGT \mapsto SRC \\
memory' = memory \\
\hline
\end{array}
$$

The subscript notation contains the parameters separated by commas, with the type and a placeholder name separated by the hash sign. The processing of the placeholders is described in Section 4.2.

3.3 Branch instruction templates

For branch instructions the `binst` collection is paramaterised with a mnemonic, and must contain a Z operation schema called *OnBranch* and, optionally, one named *NoBranch*. The *OnBranch* and *NoBranch* operation schema are prefixed to the sequential block at the target address, and the sequential block immediately following this instruction respectively. Unconditional branch instructions, such as the i386 `jmp` instruction, do not require a *NoBranch* schema, but the *OnBranch* schema may contain state change effects of the branch instruction, such as updating the program counter, if this is required for the verification.

Function call and return instructions are presented in the same way but in `callinst` and `returninst` collections, respectively. The process of converting these templates into representations of particular instruction instances is described in Section 4.4.

4 Analysis Process

The automatic analysis of a given executable to produce a model of the form described in Section 3 is 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 stages in the analysis work-flow and their inputs and outputs are shown in Figure 5.

4.1 Branch Identification

The *branch identification* stage of the analysis separates the branch instructions from the sequential instructions (as discussed in Section 3) using the supplied formal specification. Each mnemonic in the assembly language is compared to the provided branch instruction set. Where a mnemonic is identified as a branch instruction it is removed from the list of instructions, partitioning the block at that point. Additionally, if it is a local branch, the target of the branch is interpreted. If the target address falls inside an otherwise contiguous block of sequential instructions then that block is also partitioned at that address and a null, unconditional branch to the next block is inserted. This identifies that the second half can be reached by multiple routes.

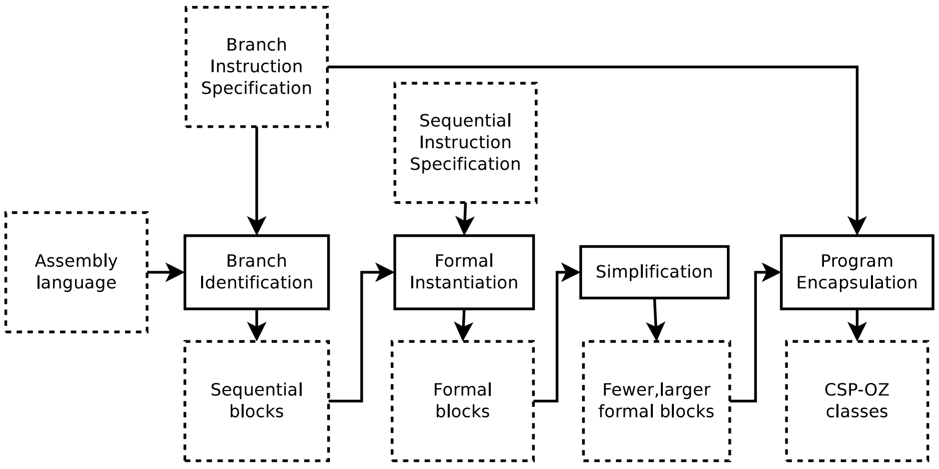

Fig. 5. The analysis workflow

This process produces a graph structure with the branch instructions forming nodes, and the blocks of sequential instructions forming edges. The unbroken lists of sequential instructions are referred to as *sequential blocks* and are named after the address of the first instruction they contain. The Z subscript convention is used, so the block starting at address $80480c4$ is named $Block_{80480c4}$. The branch instructions are also named after their locations. This naming convention retains tracability information throughout the analysis process. When a fault is identified in the completed model it is possible to locate the cause of the fault to a short block of instructions. Since these blocks represent state change with no decision making they are likely to have a clear correspondence to a small section of the original program.

4.2 Formal Instantiation

The sequential blocks produced by the branch identification stage can be converted into formal representations of their behaviour. The instructions can be independently analysed and instantiated into Z operation schema representing their behaviour. All branching behaviour has been removed so these operation schema can be sequentially composed to produce a correct (but not minimal) representation of the system interactions of the block.

The instantiation process makes use of the template specifications described in Section 3.2. Each instruction is classified by mnemonic and by the type of the parameters present in the assembly language. The matching instruction template is identified from the mnemonic and the type signature present in the subscript of the template name. The template is then instantiated to represent a particular instruction but textually replacing the parameter placeholders with the values present in the assembly language at this point. The subscript of the name is replaced with the address of the instruction to maintain tracability.

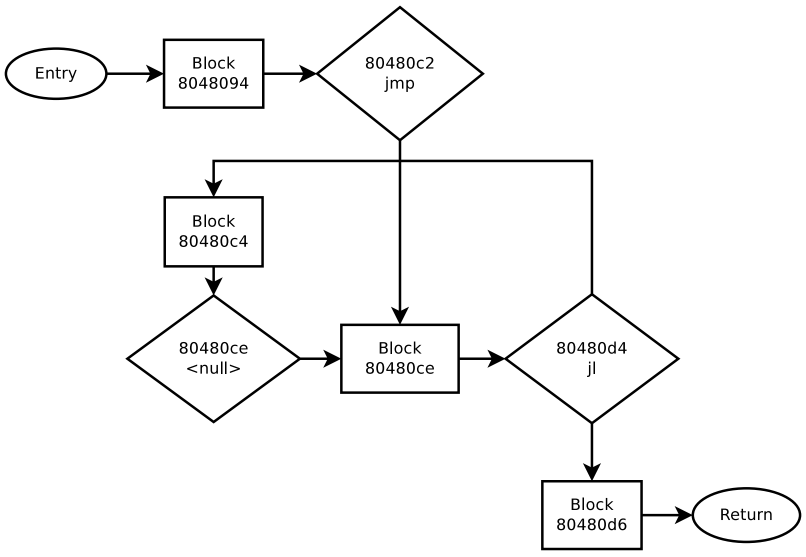

Fig. 6. Partitioning the exdev assembly code at the branch instructions

For example the instruction: `0x80480d6: mov $0x26, %eax` has the mnemonic "`mov`" and a literal parameter, followed by a register parameter. This matches the following template:

$$
\begin{array}{|l}
\multicolumn{1}{l}{mov_{LIT\#SRC,REG\#TGT}} \\
\hline
\Delta System \\
\hline
registers' = registers \oplus \{TGT \mapsto SRC\} \\
memory' = memory \\
\hline
\end{array}
$$

This is then instantiated to form:

$$
\begin{array}{|l}
\multicolumn{1}{l}{mov_{80480d6}} \\
\hline
\Delta System \\
\hline
registers' = registers \oplus \{eax \mapsto 38\} \\
memory' = memory \\
\hline
\end{array}
$$

(Note: the Z convention represents integers in decimal, whilst the assembly language is in hexadecimal, so hexadecimal 26 becomes decimal 38).

4.3 Simplification

The result of the *formal instantiation* process 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 twelve line `exdev` function produced a 25 line assembly file with only 3 branch instructions. Some technique is needed to simplify these sequential blocks if the objective is readable formal models.

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. Some level of concatenation is possible for limited computation expense using the techniques outlined in [16]. The process operates by comparing two sequentially composed schemas and determining whether their composed 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.

There is a necessary choice between producing the most succinct model theoretically possible and producing a model entirely automatically. Since it is possible to apply automatic tools to the analysis of the model (for example Z2SAL, see Section 5) it can be argued that the simplification does not need to be complete if that would require excessive human effort.

4.4 Program Encapsulation

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 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.

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 `jmp 80480ce <exdev+0x3a>` will always cause execution to transfer to virtual address `0x80480ce`. The control flow graph shows that the block beginning at `0x80480ce` ends with the branch instruction at address `0x80480d4`. The branch instructions are all represented by CSP processes named *Branch* with a subscript containing the virtual address of the instruction they represent.

$$Branch_{80480c2} = Block_{80480ce} \rightarrow Branch_{80480d4}$$

The *jl* instruction at address `0x80480d4` is a conditional branch instruction. As is discussed in Section 3, this is 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.

exdev

[...]

$$\begin{aligned}\mathit{Branch}_{80480d4} = {} & (\mathit{OnBranch}_{80480d4} \to \mathit{Block}_{80480c4} \to \mathit{Branch}_{80480c9})\\ & \Box\ (\mathit{NoBranch}_{80480d4} \to \mathit{Block}_{80480d6} \to \mathit{Branch}_{80480db})\end{aligned}$$

[...]

$\mathit{OnBranch}_{80480d4}$

$\Xi\,\mathit{System}$

$\mathit{sflag} = 1$

$\mathit{NoBranch}_{80480d4}$

$\Xi\,\mathit{System}$

$\mathit{sflag} \neq 1$

[...]

As described in Section 3, function calls are modeled by executing the function in parallel, passing the system state using schema promotion, and then synchronising on the communication. All classes that use function call instructions include the *Call* and *Return* operations, which model this synchronisation. The sequence *Call* $\to$ *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. From the `maxint` example: `call 80480d8 <max>` becomes

$$\begin{aligned}\mathit{Branch}_{8048115} = {} & (\mathit{OnBranch}_{8048115} \to \mathit{Call} \to \mathit{Return}\\ & \to \mathit{Block}_{804811a} \to \mathit{Branch}_{804811d}) \mid\mid \mathit{max}\end{aligned}$$

All functions contain an *Entry* operation and a *Leave* operation that synchronise with the calling function and receive the system state, and then return then modified system state to the calling function at the end. CSP-OZ classes require a `main` process to begin execution. This begins with the *Entry* operation that 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 An example verification

To demonstrate the usefulness of the inferred models the model produced for the example driver function was processed with the Z2SAL [8] tool. This produced an input file for the SAL suite of model checking tools [7]. The requirements specified in Section 1 were encoded as Linear Temporal Logic statements over this model and were verified using the SAL bounded model checker.

The Z2SAL tool does not accept CSP-OZ so the CSP-OZ had to be "flattened" to pure Z. The system state schema was augmented with a *cspstate* variable, defined with a BNF type that contains an atom for each of the processes in the CSP definition. The CSP control flow restrictions were converted to preconditions on this variable such that any given Z operation could only execute if the *cspstate* variable contained the name of a process that begins with this operation. The post condition of the operation then sets the *cspstate* variable to the name of the CSP process that follows this operation. This flattening is performed automatically by Spurinna.

Although a verification in SAL was developed this was limited to the bounded model checker, as even the small example state was too large for the symbolic model checker. To complement this, the CSP-OZ elements have been converted to a lightweight representation in Isabelle/HOL that allows properties to be verified symbolically over universally quantified state representations. Further details of this verification are presented in [17].

6 Conclusion

The principal difficulties that current techniques face when verifying hardware-dependent software are that: they have no way to determine statically the behaviour of code that interacts directly with the hardware; current techniques are necessarily detached from the hardware; and verification must fit into an industrial work-flow and not be overly dependent on expert skills, and must be applicable to large scale systems in reasonable time.

This work presents a technique that uses knowledge of the behaviour of specific hardware in order to allow the verification of its control software. 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 analyses operates on disassembled executables, and uses a formal specification of their target architecture as a guide to infer a model of the behaviour of the software. 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 verification of properties on the inferred model has been demonstrated using Z2SAL and Isabelle/HOL. The conversion to Isabelle/HOL has not yet been automated, and still requires manual identification of the elements of the model that interact with variables of interest. A more complete and automatic embedding of the inferred models into Isabelle/HOL is intended as the continuation of this work.

The original decision to use CSP-OZ was influenced by the Syspect tool [15] that allows slicing techniques to be applied to CSP-OZ specifications. Slicing is intended specifically to highlight elements of a program that interact with particular state components, so this would address the identification problem in a larger system model. Syspect has since been expanded to model timing behaviour [9], which could also be valuable in verifying hardware control systems.

It has not yet been possible to import the CSP-OZ specifications produced by Spurinna into Syspect, but this is a potential target for future work.

Acknowledgements

The author would like to thank his PhD supervisor, John Derrick, for his continous support, and also the PhD examiners, Georg Struth and Jonathan Bowen for their valuable and detailed suggestions.

References

1. J. G. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., 2003.
2. G. Birtwistle. Control state in asynchronous micropipelines. In A. Yakovlev and R. Nouta, editors, *AINT*, pages 45–55, 2000.
3. S. Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Computer Science Department, 2008.
4. J. P. Bowen. Formal specification and documentation of microprocessor instruction sets. *Microprocessing and Microprogramming*, 21(15):223 – 230, 1987.
5. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In J. Ferrante and K. S. McKinley, editors, *PLDI*, pages 66–77. ACM, 2007.
6. 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.
7. L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev.2), 2003. http://sal.csl.sri.com/doc/language-report.pdf [Accessed 14th March 2012].
8. J. Derrick, S. North, and A. J. H. Simons. Z2SAL: a translation-based model checker for Z. *Formal Aspects of Computing*, 23:43–71, 2011.
9. J. Faber, S. Linker, E.-R. Olderog, and J.-D. Quesel. Syspect - modelling, specifying, and verifying real-time systems with rich data. *International Journal of Software and Informatics*, 5(1-2):117–137, 2011. ISSN 1673-7288.
10. C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *FMOODS*, pages 423–438. Chapman and Hall, London, 1997.
11. D. H. Kemp. Specification of VIPER1 in Z. Technical report, Royal Signals and Radar Establishment, 1988.
12. 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. *Commununications of the ACM*, 53(6):107–115, 2010.
13. MISRA. Guidelines for the use of the C language in vehicle based software. Technical report, 1998.
14. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
15. Syspect. Final report of the syspect project. Technical report, Carl von Ossietzky University of Oldenburg, 2006.
16. R. Taylor. Separation of Z operations. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *LNCS*, page 350. Springer, 2008.
17. R. Taylor. *Verification of hardware dependent software*. PhD thesis, University of Sheffield, 2012.