Incremental Development of a High Integrity Compiler: experience from an industrial development

Susan Stepney
Logica UK Ltd., Betjeman House, 104 Hills Road, Cambridge, UK, CB2 1LQ.
stepneys@logica.com

Abstract

We have developed and successfully applied a technique to build a high integrity compiler from Pasp, a Pascal-like language, to Asp, the target language for a high integrity processor designed for the UK's Atomic Weapons Establishment at Aldermaston.

We overview the technique itself, including a description of how it can be extended to separate compilation. We also describe some of our experiences whilst implementing this compiler, how successful the whole process has been, and the lessons we have learned.

We have cost-effectively developed a compiler to high integrity by using mathematical specification and proof techniques.

1. Introduction

When developing high integrity, or safety critical, applications, no stage of the development process can be a weak link. There are many techniques for developing correct applications in high level language, using mathematical techniques for specification and refinement. Similarly, there are techniques for designing high integrity processors that execute correctly.

This paper addresses one high integrity method for bridging the gap: producing a high integrity compiler that correctly translates source code into object code. We have developed and successfully applied this technique to build a high integrity compiler from Pasp, a Pascal-like language, to Asp, the target language for a high integrity processor designed for the UK's Atomic Weapons Establishment at Aldermaston.

With this technique we use the mathematical specification language Z [3] to capture the formal semantics of the compiler, and use a process to correctly recast the Z specification of the compiler into Prolog, thereby providing an executable compiler. We also capture the formal semantics of both the source and target languages, and use these to prove that the compiler performs the translation task correctly, thus proving its integrity. The technique itself is overviewed in section 3.

In later sections we describe some of our experiences whilst implementing this compiler, how successful the whole process has been, and the lessons we have learned.

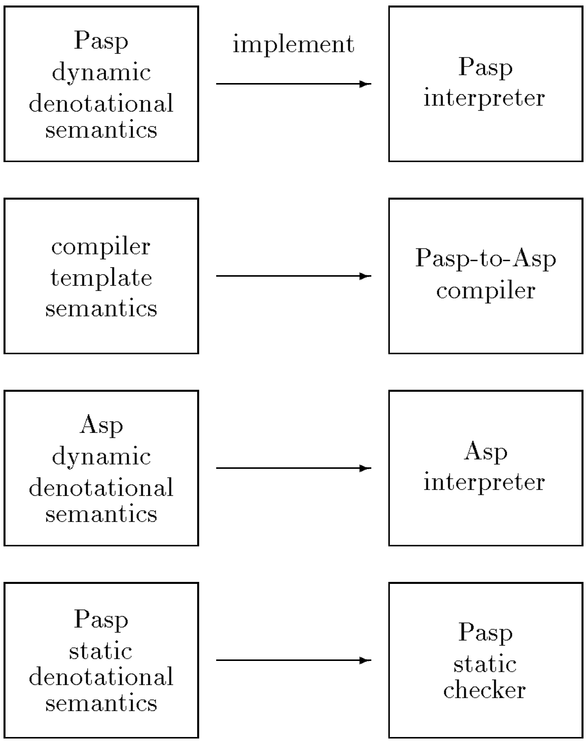
2. Historical background

In 1990 our Formal Methods Team performed a study for RSRE (now DERA Malvern) into how to develop a compiler for high integrity applications that is itself of high integrity. In that study, the source language was Spark, a subset of Ada designed for safety critical applications, and the target was Viper, a high integrity processor. We developed a mathematical technique for specifying a compiler and proving it correct, and developed a small proof of concept prototype. The study is described in [5], and the small case study is worked up in full, including all the proofs, in [4].

Engineers at AWE read about the study and realised the technique could be used to implement a compiler for their own high integrity processor, called the ASP (Arming System Processor). They contacted us, and since then we have been using these techniques, along with incremental development, to deliver a high integrity compiler, integrated in a development and test environment, for a progressively larger subset of Pascal. The current status of that compiler is described here.

3. Overview of compiler development

In this section we give a brief overview of our technique of using Z specification and Prolog implementation to implement a high integrity compiler. More detail can be found in [4].



Pasp dynamic denotational semantics
implement
Pasp interpreter
compiler template semantics
Pasp-to-Asp compiler
Asp dynamic denotational semantics
Asp interpreter
Pasp static denotational semantics
Pasp static checker

Figure 1. The various mathematical Z specifications and what their Prolog implementations provide

3.1. Implementing the correct compiler

The development of a high integrity compiler by this method has the following components (see also figure 1):

1. **Compiler specification**: The operational semantics of the source language in the form of a set of target language templates is specified in Z. (In the commercial implementation described below the source language is Pasp and the target language is Asp.)

2. **Compiler implementation**: The Z specification of the compiler is translated into Prolog, where it is executable. Executing this semantics gives a compiler.

3. **Language specifications**: The denotational semantics of both the high level source language and the low level target assembly language are specified in Z.

4. **Proof**: The compiler's operational semantics are demonstrated to be equivalent to the source language's denotational semantics: the compiler transformation is *meaning preserving*, and hence the compiler is correct.

5. **Interpreter implementation**: The Z specification of the source language and target language denotational semantics are each translated into Prolog, where they is executable. Executing a language's semantics gives an interpreter for that language.

For example, the meaning of a source language expression is the value it denotes in the current state. In Z this can be written as[1]

$$State == Name \rightarrow\!\!\!\!\!+ Value$$

State is a function from *Name*s to the *Value*s they denote.

$$\mathcal{M}_e : Expr \rightarrow\!\!\!\!\!+ State \rightarrow\!\!\!\!\!+ Value$$

$$\mathcal{M}_e[\![var(\mathtt{x})]\!]\sigma = \sigma[\![\mathtt{x}]\!]$$

$$\mathcal{M}_e[\![add(\mathtt{e}_1, \mathtt{e}_2)]\!]\sigma = \mathcal{M}_e[\![\mathtt{e}_1]\!]\sigma + \mathcal{M}_e[\![\mathtt{e}_2]\!]\sigma$$

$$\ldots$$

The meaning function for expressions maps a syntactic *Expr* to a function from *State* to the *Value* of the expression in that state. The meaning of a variable reference expression is the value of that name in the current state. The meaning of an addition expression is the values of the two sub-expressions in the current state, added together. (Remember that '*add*' is a syntactic programming language construct, whereas '+' is a mathematical operator. More sophisticated definitions would include a treatment of overflow.)

Similarly, the meaning of a command is a state change.

$$\mathcal{M}_c : Cmd \rightarrow\!\!\!\!\!+ State \rightarrow\!\!\!\!\!+ State$$

$$\mathcal{M}_c[\![assign(\mathtt{x}, \mathtt{e})]\!]\sigma = \sigma \oplus \{x \mapsto \mathcal{M}_e[\![\mathtt{e}]\!]\sigma\}$$

$$\ldots$$

The meaning function for commands maps a syntactic *Cmd* to a function from *State* to the *State* that holds after the command has been executed. The meaning of the assignment command is a state change that updates the current value of x to the value denoted by the expression $\mathtt{e}$.

Translating this Z specification into Prolog gives (part of) an executable interpreter for the source language. The assignment meaning specification can be translated as:

[1]Here $[\![\ldots]\!]$ are used as conventional denotational semantics meaning brackets, used to distinguish syntax (program fragments) from semantics (mathematical constructs). They are not Z's bag brackets.

```
meaning(assign(Name,Expr),Pre,Post):-
    meaning(Expr,Pre,Value),
    update(Name,Value,Pre,Post).
```

Similarly, the meaning of the target language's store instruction (store the contents of the accumulator at the location l) updates its state. (The semantics of the low level target language is more complicated than the source language's, and needs to use 'continuation' arguments, because the assembly language allows arbitrary jumps. See [4] for more explanation.) In Z:

$$\begin{aligned} &AState == Locn \twoheadrightarrow Value \\ &Cont == AState \twoheadrightarrow AState \\ &AEnv == Label \twoheadrightarrow Cont \end{aligned}$$

$$\begin{array}{|l} \mathcal{A} : \operatorname{seq} Instr \twoheadrightarrow AEnv \twoheadrightarrow Cont \\ \qquad\quad \twoheadrightarrow AState \twoheadrightarrow AState \\ \hline \mathcal{A}[\![\langle store\ \mathtt{l}\rangle]\!]\rho\,\theta\,\sigma = \theta(\sigma \oplus \{l \mapsto \sigma\ a\}) \\ \ldots \end{array}$$

The meaning function for instructions maps a syntactic sequence of *Instr* in the current environment and continuation, to a state change. The meaning of the sequence comprising a single *load* instruction is the before state overridden at l, with the value at l now given by the value in the a register in the before state, with the resulting state applied to the current continuation.

The operational semantics defines the sequence of target language instructions corresponding to the translation of each source language instruction. In Z:

$$OEnv == Name \twoheadrightarrow Locn$$

The translation environment maps variable names to the locations where they are stored.

$$\begin{array}{|l} \mathcal{O}_e : Expr \twoheadrightarrow OEnv \twoheadrightarrow \operatorname{seq} Instr \\ \mathcal{O}_c : Cmd \twoheadrightarrow OEnv \twoheadrightarrow \operatorname{seq} Instr \\ \hline \mathcal{O}_c\{\!|assign(\mathtt{x},\mathtt{e})|\!\}\rho = \mathcal{O}_e\{\!|\,\mathtt{e}\,|\!\}\rho \frown \langle store(\rho\,\mathtt{x})\rangle \\ \ldots \end{array}$$

The translation function for expressions and commands maps a syntactic *Expr* or *Cmd* in the current translation environment to the corresponding sequence of instructions. The assignment command is translated into a sequence of instructions to evaluate the expression (and leave its value in the a register), followed by an instruction to store that value at the appropriate location.

Translating this Z specification into Prolog gives (part of) an executable compiler from the source language to the target language. The assignment compilation specification can be translated as:

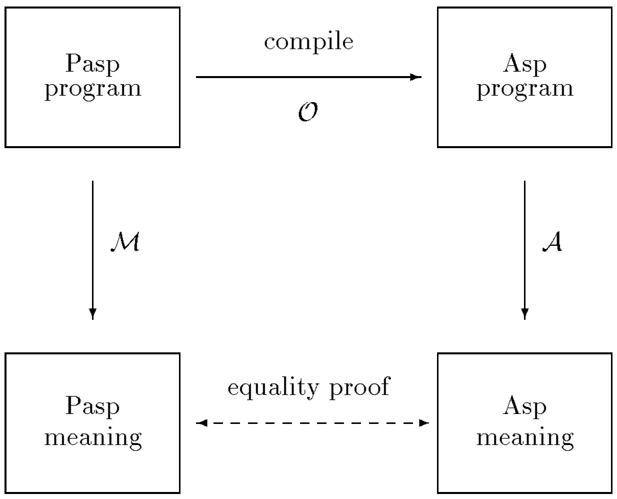


Figure 2. The compiler correctness proof obligation

```
compile(assign(Name,Expr),Env,
        [InstrList,store(Name,Env)]):-
    compile(Expr,Env,InstrList).
```

To show that the compiler is correct, we need to show that the translation into the target language preserves the semantics of the source language instruction. We can do so by proving that the target language meaning of each translation template is the same as the source language meaning of the original instruction (see figure 2). For the assignment example we need to prove that (omitting states and environments here for clarity)

$$\vdash \mathcal{M}_c[\![assign(\mathtt{x},\mathtt{e})]\!] = \mathcal{A}[\![\mathcal{O}_c\{\!|assign(\mathtt{x},\mathtt{e})|\!\}]\!]$$

The proof obligations for separately compiled modules are a little different; see section 5 for more explanation.

As we can see from the above descriptions, the specifications serve (at least) three potentially conflicting purposes:

- **Language definition**: the denotational semantics act as the language definition, and so the specification style should be as clear and abstract as possible [1, Chapter 14] to make the definition comprehensible to human readers.

- **Implementation**: the various semantics are translated into an executable language, and so should be written in a concrete, algorithmic, style that facilitates this translation. The declarative Prolog, rather than some imperative language, was chosen as the target language in order to minimise this implementation step.

- **Proof**: the various semantics need to be manipulated mathematically, in order to perform the correctness proofs, and so should be written as abstractly as possible.

It is important to balance these conflicts to obtain the best benefit from the specifications. For example, using Prolog itself as a specification language would eliminate the implementation problem, but would make the language specification less clear, and would greatly increase the proof burden. Proof is probably the hardest part of the development, so our specification style leans towards easing this.

We ended up writing our specifications in a slightly more algorithmic style than is usual for Z specifications, and kept to a single style throughout, to aid translation. However, we did keep quite a high level of abstraction in the specification, in order to ease the proof burden.

3.2. Static semantics

The particular denotational semantics described above is known as the *dynamic semantics*; it captures the behaviour of a program as it executes. Other *static semantics* can be specified for a language as well. (They are so called because they define a meaning for a program that can be evaluated statically, without executing it.) For example, an expression could be considered to denote a type, rather than a value. Such meanings define when programs are type correct.

$$TState == Name \nrightarrow Type$$

$$\begin{array}{|l}
\mathcal{T}_e : Expr \nrightarrow TState \nrightarrow Type \\
\mathcal{T}_c : Cmd \nrightarrow TState \nrightarrow TCheck \\
\hline
\mathcal{T}_c[\![assign(\mathtt{x}, \mathtt{e})]\!]\tau = \\
\qquad \mathbf{if}\ \tau[\![\mathtt{x}]\!] = \mathcal{T}_e[\![\mathtt{e}]\!]\tau\ \mathbf{then}\ okay\ \mathbf{else}\ wrong \\
\ldots
\end{array}$$

The type-meaning function for expressions maps a syntactic *Expr* to a function from *TState* to the *Type* of the expression in that type-state. The type-meaning function for commands maps a syntactic *Cmd* to a function from *TState* to the check status *TCheck* of the command. The type-meaning of an assignment command is *okay* if the types of the variable x and the expression e in the current type-state are the same, otherwise it is *wrong*.

Translating this Z specification into Prolog gives (part of) an executable type checker for the source language. Other static semantics can be defined as desired, and implemented as static code checkers.

4. The real Pasp compiler

The example sketched above, and worked in full detail in [4], is for a very small language. Over the past several years we have been developing a full compiler for a real language, incrementally adding source language constructs, using these techniques.

4.1. The target language, Asp

The ASP chip is an 8 bit processor with memory mapped input/output, specifically designed for high integrity embedded applications. Emphasis has been placed on correct design and built-in-test capabilities; consequently the instruction set is not over-burdened with complicated functionality. For example, there are few registers, few addressing modes, and no support for floating point numbers.

The meaning function of Asp is not particularly complicated, because of the processor's relatively simple structure. Most of the bulk of the specification is due to capturing every instruction's behaviour at quite a low level of detail.

Our specification of the Asp assembly language (abstract syntax and dynamic semantics) runs to approximately 50 pages of Z, which includes both the mathematics and the natural language commentary.

4.2. The source language, Pasp

The high level language, Pasp, is a Pascal-like language, designed to support high integrity applications running on the ASP chip. The language currently includes:

- data types of 16-bit unsigned numbers, bytes, booleans, and enumerated types, and multi-dimensional arrays of these types
- various unary and binary arithmetic and logical operations for each of the data types, and operations for explicit casting between types
- blocks, while loop, if-then-else choice, and case choice
- procedures and functions, with call by value and call by reference
- input/output, by way of special pragmas to support the memory mapped i/o of the ASP chip
- modules, with import, export, separate compilation and linking (see section 5)

- four separate static semantics, to ensure: declaration before use, type correctness, initialisation before use, and use after declaration (failure of this latter check results in a warning only)

Several features have been deliberately omitted from Pasp, because of its intended use in high integrity applications. For example, it has no recursion, no pointers, and no floating point numbers.

The complexity and size of the real language Pasp, compared to the typical small case study languages such as that in [4], results in quite an involved specification. The presence of block structuring, and procedures and functions, makes the various meaning functions more complicated than those in the example above, as they have to accommodate scope and side effects. The meaning of an expression is no longer simply the value it denotes; if that expression includes a function call, the state can change as well.

Our specification of the current Pasp language (abstract syntax, concrete syntax, four static semantics and the dynamic semantics) runs to approximately 150 pages of Z.

4.3. The compiler and linker specifications

The complexity in the source language and the simplicity of the target language has resulted in a large compiler specification. In particular, the need to specify 16-bit arithmetic operations using 8-bit instructions results in some quite large arithmetic operation templates. Also, the requirement for separate compilation, with a separate link stage, increases the size substantially.

Our specification of the compiler templates (translation of Pasp dynamic semantics into Asp instructions) runs to approximately 100 pages of Z. The linker specification is a further 20 pages of Z.

4.4. The implementation

Currently, the full compilation suite supports the following:

- implementation of the dynamic semantics of Pasp, providing a Pasp interpreter
- implementation of the dynamic semantics of Asp, providing an Asp interpreter
- implementation of the operational semantics of Pasp to Asp templates, providing a compiler
- implementation of a linker, to support separate compilation
- implementation of the four separate static semantics, providing four Pasp code checkers
- support for Pasp assertions and invariants
- automatic test generation harness

In the cases of Prolog implementation from Z specifications, the listing of the Prolog source code is approximately the same size as the Z specification, as would be expected from the roughly one-to-one transcription relationship.

5. Separate compilation

One of the more recent incremental extensions we have supplied to the Pasp language and its compiler is the provision of a simple separate compilation facility. Pasp now has modules; procedures and functions from within a module may be exported, and imported into other modules.

The techniques described in section 3 need to be extended to cope with modules. A module in isolation cannot be ascribed a simple meaning, because its behaviour is parameterised by the meaning of any imported procedures or functions. So we specify the meaning in stages.

First, we specify how modules are *flattened* to produce an unmodularised Pasp program, which itself does have a well-defined meaning.

$$\begin{array}{|l} \mathit{flatten} : \mathbb{P}\,\mathit{Module} \mathrel{+\!\!\!\!\rightarrow} \mathit{Program} \\ \hline \ldots \end{array}$$

We also specify the compiler templates for a single module in terms of an extended Asp assembly language, XAsp, which has extra instructions to capture the links between exported and imported items.

$$\begin{array}{|l} \mathcal{O}_m : \mathit{Module} \mathrel{+\!\!\!\!\rightarrow} \mathrm{seq}\,\mathit{XAsp} \\ \hline \ldots \end{array}$$

We cannot specify the meaning of these extra instructions in isolation, because they are parameterised by the meanings of other compiled modules. Instead, we specify the meaning of these extra instructions indirectly, by defining how compiled modules are *linked* to form a plain Asp program. In the linking process all the extra instructions get converted to plain Asp instructions.

$$\begin{array}{|l} \mathit{link} : \mathbb{P}(\mathrm{seq}\,\mathit{XAsp}) \mathrel{+\!\!\!\!\rightarrow} \mathrm{seq}\,\mathit{Asp} \\ \hline \ldots \end{array}$$

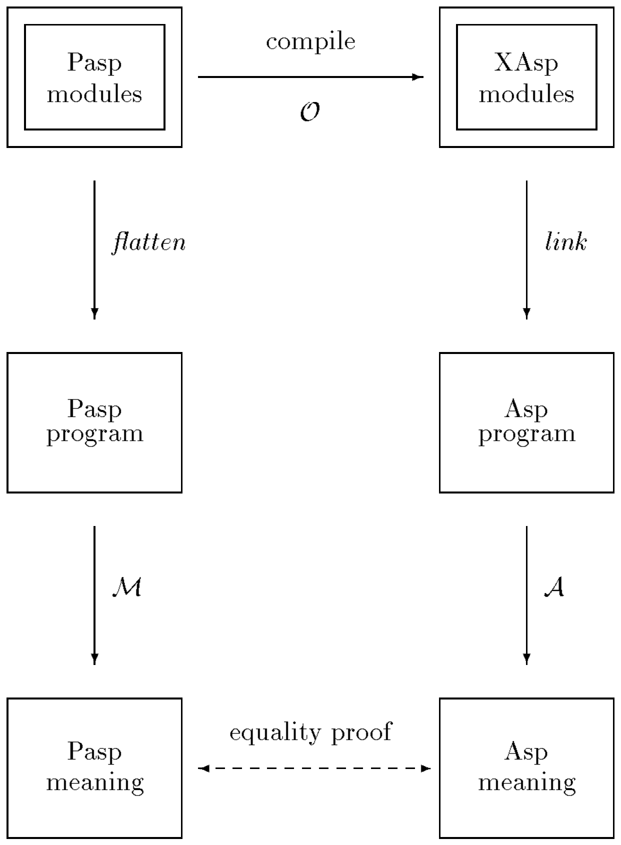


Figure 3. The compiler/linker correctness proof obligation for modules

Then the proof obligation becomes the need to show that the meaning of a bunch of modules when flattened into a Pasp program is the same as the Asp meaning of those modules each compiled, then linked (see figure 3):

$$\begin{aligned}\vdash \mathcal{M}\llbracket \mathit{flatten}\{\mathtt{m}_1, \ldots, \mathtt{m}_n\}\rrbracket = \\ \mathcal{A}\llbracket \mathit{link}\{\mathcal{O}\langle\!|\mathtt{m}_1|\!\rangle, \ldots, \mathcal{O}\langle\!|\mathtt{m}_n|\!\rangle\}\rrbracket\end{aligned}$$

During development, we decided what the module proof obligation should be, and used that to help us to define how the modules would work, and to design the appropriate syntaxes for them.

6. Lessons learned

6.1. The development team

The development team has two parts: two formal specification members, writing the Z specifications and doing the proofs, and two implementors, writing the Prolog from the specifications and adding the non-formal parts of the environment. By historical accident, the two parts of the team are in different offices, separated by 100 miles.

One might expect this separation to have an adverse effect on the development. After all, the implementor has to ensure that the Prolog they write conforms to the specification. Surely there are going to be ambiguities and difficulties that are hard to resolve? In practice, no. The Z specification is sufficiently precise that the implementor and specifier have rarely needed to discuss it together.

6.2. Bugs found

The main aim of developing any software formally is to improve its integrity – its correct operation; in other words, to eliminate bugs. How successful is our implementation?

Transcription errors

Minor transcription error from the Z to the Prolog were the major source of problems. These were caught by in-house testing, because this kind of error usually makes the Prolog fail.

We are currently investigating ways of performing this transcription automatically.

Specification errors caught by proof

While casting the specific form of the proof obligation for the function construct, the specifier realised that the way the stack had been specified would not work if the function call was embedded in an arithmetic expression.

Doubtless this error would have been caught by testing, but it is interesting to note that the error was caught just by thinking about the proof obligation, not actually discharging it! Indeed, we have never found a bug whilst performing a proof (but see the story about division, later); it seems that just realising that proof will be performed puts the specifier in a critical frame of mind that helps eliminate bugs.

NPL's torture tests

AWE decided to give our compiler some rigorous testing, by getting the UK's National Physical Laboratory (http://www.npl.co.uk) to perform its infamous 'torture testing' on it. NPL are justly reknowned for developing test suites that routinely break Ada compilers. They developed a special variant of their tests for Pasp, and aimed them at our compiler.

We were understandably nervous about this process. After all, other supposedly high integrity compilers had been brought to their knees by these tests. However, much to our relief, and joy, only two issues were found.

A different MININT

The first of these is not actually an error at all. Our compiler failed to handle Pascal's MININT properly. However, in order to make our specification and proof task significantly easier, we had specified Pasp's MININT differently from Pascal's:

$$\text{MININT}_{Pasp} = 1 + \text{MININT}_{Pascal}$$

We were able to point to the relevant line in the Z specification, to show that the compiler was in fact being tested on a Pasp program with explicitly undefined behaviour, and hence that the observed behaviour was a valid implementation.

Division algorithm error

NPL's testing also revealed a flaw in the 16-bit division implementation, which did not work for some divisors.

Interestingly, this is one of the few areas of the compiler that, due to the incremental nature of the project, we are still waiting to perform the proofs on. This seems to demonstrate again that proof does provide a useful increase in integrity.

We fixed the problem, which was caused by missing a minor case in Knuth's unsigned long division algorithm [2, section 4.3.1]. Rather than prove the new version, we instead exhaustively tested it for all $2^{16} \times 2^{16}$ cases. (This does not, of course, constitute a mathematical proof. It does demonstrate that our implementation of 16-bit division using 8-bit division is consistent with the implementation of 8-bit and 16-bit division on our test machine.)

Testing conclusions

So, only one bug, the division error, was discovered in the delivered compiler. This is remarkably small for a serious compiler development. NPL's tests routinely break even established compilers; many compilers cannot even parse some of the more deeply-nested constructs. NPL did express surprise at how few issues they found[2].

Testing did not discover any bugs in the parts we had proved. Mathematical specification and proof do seem to have provided higher integrity in this case.

[2]Dave Thomas, AWE, private communication, 1997.

6.3. The role of proof

The more effort put in to formalising a specification, the higher integrity the result.

Specification only: this provides clarity of thought, and a clean design, which in itself produces a better implementation. For example, we discovered that specifying the memory mapped i/o provided better language support for these operations.

Stating the proof obligations: if the specification writers also cast the proof obligations, or at least study them critically, this process can uncover errors, as the specifiers are forced to think in a new way about particular consequences of their work. For example, we discovered an error in our first attempt to specify functions this way. Also, deciding on what the module proof obligation should be helped us to define how the modules would work.

Hand proof: can uncover yet more errors. For example, we are reasonably confident that we would have uncovered the division bug in the proof attempt.

Machine assisted proof: again, more errors get discovered. For example, [6] reports on the use of the PVS proof tool to redo the hand-proofs in [4]. This exercise uncovered a few hidden assumptions and slight hand-wavings in the original proofs (but no errors in the theorems).

Each extra stage in this process adds more assurance, for more cost. A cost-benefit analysis can be used to decide how much proof effort to expend. But it is important to note that proof is not 'all or nothing'.

One aspect to bear in mind is that conventionally-developed compilers may gain a degree of assurance simply by being 'well-known'. If they have been widely used, any bugs or other problems get widely reported. However, this assurance does depend very much on the mode of use. Automatic code generation exercises lesser-known parts of the compiler that hand-written code does not, potentially exposing previously undiscovered bugs. So having an existing well-understood compiler might not give you the degree of assurance that you had thought[3]. If using a compiler with automatically generated code, the balance tips further in favour of a formally developed bespoke compiler.

6.4. Interpreters

Even though this project is mainly focussed on providing a compiler, the various interpreters made available have proved to be very useful.

We implemented the Pasp dynamic denotational semantics to provide a Pasp interpreter. This has been

[3]Ib Sørensen, B-Core, private communication, 1998.

found useful for two reasons. Firstly, the ASP chip was still being fabricated at the start of the project, and so the Pasp interpreter enabled application developers to get a head start on writing and exercising their code. Secondly, the interpreter provides the basis for many of the test harnesses. Pasp programs can be annotated with various assertions and invariants; the compiler strips these out, but the interpreter checks that they hold during program execution. Similarly, Pasp programs can be annotated with automatic test generation commands; again, the compiler ignores these, but the interpreter uses them to generate test data and coverage statistics. So the same source code can be interpreted with testing on, and compiled with testing removed.

We implemented the Asp dynamic denotational semantics to provide an Asp assembly language interpreter. Because the ASP chip was not immediately available, this again provided implementors with a head start in writing and testing programs. Even since the processor has become available, the Asp interpreter is still used, because it provides greater visibility to the internals of an executing program's operation.

6.5. Incremental development

We are developing this compiler incrementally, adding more constructs to the Pasp language, and adding more features to the testing environment.

This approach has had its successes and drawbacks. On the success side, it provides a very flexible approach in a case where requirements can change as the project develops. The early releases of our compiler got used by groups for whom it had not originally been designed; later phases incorporated some new requirements from these groups. Also, the development of the compiler fed back into the processor design. Early on, a 16-bit register was added to simplify array accessing, since the processor has no support for offset addressing. More recently, our work on implementing procedures and functions was used to help design a suitable set of hardware stack instructions.

On the other hand, a phased approach can result in a degree of rework. Each phase has to produce a functioning deliverable, which requires some finishing work that has to be undone for the next phase. Also, some rework can be required to extend definitions.

7. Conclusions

On balance, in this project, an incremental approach has proved to be both cost effective and flexible.

Small, reduced functionality versions of the compiler and its environment have been available from early on in the project, allowing the client to progress with their own work and to influence the design of later phases in ways that could not have been apparent before they had actually used the system for real.

8. Future Work

We are continuing with further phases in this compiler development.

In particular, engineers at AWE develop some of their code formally using the B-tool, which generates Pasp code from B specifications. B has some facilities for sharing across modules not supported by the current version of Pasp. We are adding these facilities in the next phase.

The templates produced by this method can sometimes be longer than strictly necessary, and are sometimes repetitive. We are investigating formally correct optimisation of the templates.

Acknowledgements

Logica UK Ltd carried out the original study work on the compiler method for RSRE (now DERA Malvern). We are continuing the development of the Pasp-to-Asp compiler for AWE plc, and would like to thank Dave Thomas and Wilson Ifill of AWE.

Our specification and implementation team includes Ian Nabney, Tim Wentford, and John Taylor.

References

[1] R. Barden, S. Stepney, and D. Cooper. *Z in Practice.* BCS Practitioners Series. Prentice Hall, 1994.

[2] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, 2nd edition, 1981.

[3] J. M. Spivey. *The Z Notation: a Reference Manual.* Prentice Hall, 2nd edition, 1992.

[4] S. Stepney. *High Integrity Compilation: A Case Study.* Prentice Hall, 1993.

[5] S. Stepney, D. Whitley, D. Cooper, and C. Grant. A demonstrably correct compiler. *BCS Formal Aspects of Computing*, 3:58–101, 1991.

[6] D. W. Stringer-Calvert, S. Stepney, and I. Wand. Using PVS to prove a Z refinement: A case study. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, number 1313 in Lecture Notes in Computer Science, pages 573–588. Springer Verlag, Sept. 1997.