A Verified Formal Model of a VC Generator

R.D. Arthan
Lemma 1 Ltd.
2nd Floor, 31A Chain Street
Reading UK RG1 2HX
rda@lemma-one.com

Abstract

This paper describes some modelling work carried out to inform understanding of an Ada verification system. It presents a simple formal model in Z of a refinement notation comprising a miniature, but complete, imperative programming language annotated with formal specifications. The semantics of that programming language and the notion of correctness relative to the specification annotations is defined. A semantic model of a verification condition generator is given which can be proved to be sound with respect both to the programming language semantics and to the intensional semantics of the specification annotations. The specifications and proofs were prepared using the ProofPower system and all proofs have been fully machine-checked. We argue that the use of appropriate abstractions and good tools make machine-checked proof a realistic and beneficial target.

1 Introduction

The Compliance Tool is a specification and verification system for Ada code annotated with specifications written in the Z notation [14, 13]. The tool is implemented as an application of the ProofPower specification and verification system for Z and HOL [3]. The tool has been used on several large scale verifications of avionics control software [11]. It is the subject of an ongoing programme of enhancements to its ease of use and to its range of applicability. Following a long tradition, the tool implements specification and verification via refinement of Floyd-Hoare style pre- and post-conditions [8, 7, 10, 6].

The implementation of the Compliance Tool is based on a Z specification and that specification has been invaluable in its development [12]. However, the specification is an operational description of the tool's input/output relationship, and is not intended to give an abstract account of the semantics of the notation supported by the tool. In 2002, the main sponsors of the tool, QinetiQ, funded a round of enhancements which included extensions to the range of flow control statements to be supported by the tool, e.g., to allow exit statements which break out of a nested loop. With the original intention of just establishing terminology, simple abstract models of the notation and its semantics were developed. This model proved useful in understanding the problems and designing and implementing the solutions.

As an exercise, I extended the model to include a formal specification of an abstract verification condition generator and, after a little experimentation, found that this was very amenable to a machine-checked proof of correctness. This proof was of some benefit in gaining further understanding of the notation and its implementation.

This paper presents a slightly simplified version of the specification and presents the main lemmas and theorems that make up the correctness proof. The lemmas and theorems are presented as Z conjectures. All of these conjectures have been proved with ProofPower.

The Z specification we give in this paper is a slightly adapted version of the material provided as part of a ProofPower case study [2, 1] . The only difference is that [1] instantiates generic notions of specification and refinement defined in [2]. For brevity here, we just give the specific instances of these notions as we need them. This was in fact what was done in the original work in 2002 — the use of the generic notions was a subsequent adaptation.

The Z specification uses infix notation for the following relation and function symbols:

relation $\_ \sqsubseteq \_, \_ \models \_, \_ \perp \_$
function 60 leftassoc $\_ \rhd_* \_, \_ \mathrel{\rhd\!\!\rhd}_* \_$

The specification is given in sections 2, 3 and 4 below: section 2 introduces our framework for specifying imperative programs and our notion of refinement; section 3 defines the abstract syntax and semantics of the programming language together with its specification annotations and defines our notions of program correctness; section 4 specifies the VC generation algorithm. The various lemmas and theorems proved are stated as Z conjectures interleaved with the specifications.

I make no claims for the novelty of the theory presented here, although the intensional aspects of the work do bring out some interesting properties that seem to be underemphasised in many accounts. However, I do believe it is important to have examples of formal methods being applied to the development of formal methods tools. What was particularly pleasing about the present example was that a very small amount of effort expended in fully machine-checked proof led to a much improved understanding of the problem at hand.

2 Specification and Refinement

2.1 States and State Transformers

Imperative programs work by modifying a state, but the internal structure of states is not relevant for present purposes. For a conventional imperative programming language, the states will be assignments of values to program variables. We just introduce a given set to represent the states:

$[\mathsf{STATE}]$

The commands in our programming language will denote *state transformers*. A state transformer is just a relation on states:

$\mathsf{STATE\_TRANSFORMER} \mathrel{\widehat{=}} \mathsf{STATE} \leftrightarrow \mathsf{STATE}$

We think of a state transformer t as *responding* to a state, s, in its domain, the *before-state*, by non-deterministically selecting some *response* or *after-state*, which is a state s' such that $(\mathsf{s}, \mathsf{s}') \in \mathsf{t}$. For example, an atomic program statement such as the assignment $\mathsf{X} := -1$ denotes the operation which given any before-state s selects the after-state s' in which X has the value -1 and all other variables have the same value as in s.

2.2 Specifications

We choose to use semantic rather than syntactic notions wherever we can. Syntactically, a predicate on a program state would be some logical combination of primitive assertions about the values of program variables. Semantically, a predicate is just a set of states, namely those states in which the corresponding syntactic predicate denotes true. For example, if X is a real-valued program variable the semantic value of the syntactic predicate $\mathsf{X} > 0$ comprises precisely those states in which X is indeed positive.

$\mathsf{PRED} \mathrel{\widehat{=}} \mathbb{P}\mathsf{STATE}$

A pre-condition is just a predicate, in the above semantic sense. We think of a pre-condition as applying to the before-state of a state transformation.

$\mathsf{PREC} \mathrel{\widehat{=}} \mathsf{PRED}$

The notion of post-condition is more complicated. While many accounts are written as if a post-condition as just a predicate on after-states, in general, a post-condition must describe a relationship between before-states and after-states. In syntactic accounts, this is often achieved by the use of auxiliary variables, e.g., see [6]. In our semantic account, a post-condition is just a relation. In giving examples of post-conditions, we will adopt the convention of using a subscript 0 used to distinguish values in the before-state, when necessary. So, for example, the syntactic post-condition $\mathsf{Y} = \mathsf{Y}_0/\mathsf{X}$ denotes the relation which holds between a before-state and an after-state precisely when the value of Y in the after-state is the result of dividing its value in the before-state by the value of X in the after-state.

$\mathsf{POSTC} \mathrel{\widehat{=}} \mathsf{STATE} \leftrightarrow \mathsf{STATE}$

A *specification statement* is a pair comprising a pre-condition and a post-condition.

$\mathsf{SPEC} \mathrel{\widehat{=}} \mathsf{PREC} \times \mathsf{POSTC}$

In examples, we will adopt the convention of the Compliance Notation (cf. also [10]) in which a specification statement has the form $\Delta\ \ \mathcal{W}\ [\ \mathcal{P},\ \ \mathcal{Q}]$ where $\mathcal{W}$ is a list of program variables called the *frame* and $\mathcal{P}$ and $\mathcal{Q}$ are syntactic predicates giving the pre-condition and post-condition respectively. The frame lists the program variables that may be changed by the code being specified I.e., for each variable, V, that is not in the frame, the post-condition implicitly includes a conjunct $V = V_0$.

So, for example, $\Delta\mathsf{Y}[\mathsf{X} > 0,\ \mathsf{Y} = \mathsf{Y}_0/\mathsf{X}]$ specifies a state transformation in which only Y may change, in which the before-state is required to satisfy the pre-condition $X > 0$, and in which the after-state may be obtained from the before-state by giving Y the value of the expression Y/X calculated in the before-state. Note that, because X is not in the frame, $X = X_0$ here and so we do not need to decorate X with a subscript in the post-condition. As another example, $\Delta\mathsf{X},\ \mathsf{Y},\ \mathsf{T}[\mathsf{X} = \mathsf{Y}_0 \wedge \mathsf{Y} = \mathsf{X}_0]$ specifies a state transformation in which the values of the variables X and Y are to be interchanged, possibly with some unspecified side-effect on the variable T.

In the Compliance Notation, Ada procedures are specified by giving a specification statement in the procedure header as in the following examples.

```
procedure SWAP
ΔX, Y [true, X = Y0 ∧ Y = X0)];
procedure NDSWAP
ΔX, Y [true, {X, Y} = {X0, Y0}];
```

Here SWAP interchanges the values of two global variables, whereas NDSWAP either interchanges the values or leaves them unchanged.

2.3 Refinement

We now define the important notion of *refinement*. Refinement is the relation that obtains between a specification and a satisfactory implementation of that specification, where an "implementation" is simply another specification, typically more concrete than the abstract specification it refines. For example, the specification of the procedure SWAP in section 2.2 is a possible refinement of the specification for NDSWAP.

The formal definition of the refinement relation is as follows, in which $\mathsf{s_1} \sqsubseteq \mathsf{s_2}$ is written to mean that s_2 is a refinement of s_1.

$$
\begin{array}{|l}
\_ \sqsubseteq \_ : \mathsf{SPEC} \leftrightarrow \mathsf{SPEC} \\
\hline
\forall \mathsf{prec_1},\ \mathsf{prec_2} : \mathsf{PREC};\ \mathsf{postc_1},\ \mathsf{postc_2} : \mathsf{POSTC} \bullet \\
\quad (\mathsf{prec_1},\ \mathsf{postc_1}) \sqsubseteq (\mathsf{prec_2},\ \mathsf{postc_2}) \\
\Leftrightarrow \quad \mathsf{prec_1} \subseteq \mathsf{prec_2} \\
\land \quad \mathsf{prec_1} \lhd \mathsf{postc_2} \subseteq \mathsf{postc_1}
\end{array}
$$

The two conjuncts in the definition of refinement are known as the liveness and safety conditions (e.g., see [15]). The liveness condition is a weak one corresponding to assertions of partial correctness, i.e., assertions of the form "program p satisfies specification s providing it terminates normally".

In [2], elementary properties of a generic notion of refinement along the above lines is considered; it is shown there that the refinement ordering constitutes a complete lattice and explicit constructions of the meets and joins are given.

3 Programs

3.1 Syntax and Semantics

Our notion of program has five syntactic categories:

Atom	A primitive operation on the state.
Seq	Sequential composition
If	If-then-else
While	While-loop
Spec	A program with a specification annotation

The following free type gives the abstract syntax of programs, in which we mingle semantic and syntactic concepts to simplify later work. We also use a tree structure rather than a linear list for sequential composition, since that is semantically harmless, and, again, helps to keep things simple later on. (The ProofPower syntax for free type definitions is currently slightly non-standard in not using the chevron symbols.)

$$
\begin{array}{ll}
\mathsf{PROG} ::= & \\
 & \mathsf{Atom}\ ((\ \mathsf{STATE\_TRANSFORMER}\ )) \\
| & \mathsf{Seq}\ ((\ \mathsf{PROG} \times \mathsf{PROG}\ )) \\
| & \mathsf{If}\ ((\ \mathsf{PRED} \times \mathsf{PROG} \times \mathsf{PROG}\ )) \\
| & \mathsf{While}\ ((\ \mathsf{PRED} \times \mathsf{PROG}\ )) \\
| & \mathsf{Spec}\ ((\ \mathsf{SPEC} \times \mathsf{PROG}\ ))
\end{array}
$$

In a typical imperative language, the atoms might be the denotations of assignment statements and procedure calls. As discussed in section 2.2, in the Compliance Notation, the denotation of a procedure call is effectively represented by an instance of the formal specification appearing in the procedure header.

The following function gives the semantics of this notion of a program. The semantic value of a program is a state transformer. In the semantics, the specification annotations are just ignored — it is the actual code that determines the semantics, not our aspirations for it.

$$
\begin{array}{|l}
\mathsf{semantics} : \mathsf{PROG} \rightarrow \mathsf{STATE\_TRANSFORMER} \\
\hline
\forall \mathsf{t} : \mathsf{STATE\_TRANSFORMER}; \\
\ \mathsf{p_1},\ \mathsf{p_2} : \mathsf{PROG};\ \mathsf{c} : \mathsf{PRED};\ \mathsf{s} : \mathsf{SPEC} \bullet \\
\quad \mathsf{semantics}\ (\mathsf{Atom}\ \mathsf{t}) = \mathsf{t} \\
\land \quad \mathsf{semantics}\ (\mathsf{Seq}(\mathsf{p_1},\ \mathsf{p_2})) = \\
\qquad \mathsf{semantics}\ \mathsf{p_1} \mathbin{⨾} \mathsf{semantics}\ \mathsf{p_2} \\
\land \quad \mathsf{semantics}\ (\mathsf{If}(\mathsf{c},\ \mathsf{p_1},\ \mathsf{p_2})) = \\
\qquad (\mathsf{c} \lhd \mathsf{semantics}\ \mathsf{p_1}) \cup (\mathsf{c} ⩤ \mathsf{semantics}\ \mathsf{p_2}) \\
\land \quad \mathsf{semantics}\ (\mathsf{While}(\mathsf{c},\ \mathsf{p_1})) = \\
\qquad (\mathsf{c} \lhd \mathsf{semantics}\ \mathsf{p_1})^* ⩥ \mathsf{c} \\
\land \quad \mathsf{semantics}\ (\mathsf{Spec}(\mathsf{s},\ \mathsf{p_1})) = \\
\qquad \mathsf{semantics}\ \mathsf{p_1}
\end{array}
$$

It is in the above that the convenience of dealing with partial correctness becomes apparent. The semantic equation for a while-loop says that the body of the loop is to be executed repeatedly in states satisfying the predicate c until a state which does not satisfy c is reached. If this fails to terminate the result is just the empty relation: we are under no obligation to assign any more complex notion of meaning to the non-terminating execution.

The partial semantics also embraces in an abstract way the possibility of the program failing gracefully. Throughout the sequel, when we talk about non-termination, we include the possibility that via some exception-raising mechanism that is outside the scope of the present model, execution of a command may result in some kind of abnormal termination which is handled properly in the physical environment in which the program is executed. For example, in Ada the assignment $\mathsf{X} := 1/\mathsf{Y}$ will lead to the program raising an exception if Y happens to have the value zero.

3.2 Program Correctness

For a program to be correct every part of it that has a specification must certainly satisfy that specification, which in our setting means that the semantic value of the program must be a refinement of the given specification annotation. We write $\mathsf{p} \models \mathsf{s}$ to mean that program p satisfies specification s.

$$
\begin{array}{|l}
\_ \models \_ : \mathsf{PROG} \leftrightarrow \mathsf{SPEC} \\
\hline
\forall\ \mathsf{prog} : \mathsf{PROG};\ \mathsf{prec} : \mathsf{PREC}; \\
\quad \mathsf{postc} : \mathsf{POSTC} \bullet \\
\qquad \mathsf{prog} \models (\mathsf{prec}, \mathsf{postc}) \\
\Leftrightarrow \quad (\mathsf{prec}, \mathsf{postc}) \sqsubseteq (\mathsf{prec}, \mathsf{semantics\ prog})
\end{array}
$$

However, a good intuitive notion of correctness also requires the pre-condition of each specification in the program to be satisfied whenever the relevant part of the program is executed. For example, every part of a program that has a specification might satisfy its specification, but, the program might still include a reachable specification with an empty pre-condition: clearly, this part of the program is "correct, but for the wrong reasons".

If p is a program and c is a set of states, we will say that p is *upright* on c iff. no pre-condition in p will be violated when p is executed in a starting state in c. We write $p \perp c$ when this holds (reverting to the original mathematical use of the symbol $\perp$ as an infix relation with apologies to those more accustomed to it as denoting the bottom element in a lattice).

$$
\begin{array}{|l}
\_ \perp \_ : \mathsf{PROG} \leftrightarrow \mathsf{PRED} \\
\hline
\forall \mathsf{t} : \mathsf{STATE\_TRANSFORMER};\ \mathsf{p}_1, \mathsf{p}_2 : \mathsf{PROG}; \\
\quad \mathsf{c}_1, \mathsf{c}_2, \mathsf{prec} : \mathsf{PREC};\ \mathsf{postc} : \mathsf{POSTC} \bullet \\
\qquad ((\mathsf{Atom\ t}) \perp \mathsf{c}_1) \\
\wedge \quad ((\mathsf{Seq}(\mathsf{p}_1, \mathsf{p}_2) \perp \mathsf{c}_1) \Leftrightarrow \\
\qquad\qquad \mathsf{p}_1 \perp \mathsf{c}_1 \wedge \mathsf{p}_2 \perp \mathsf{semantics}\ \mathsf{p}_1\ (\!|\mathsf{c}_1|\!)) \\
\wedge \quad ((\mathsf{If}(\mathsf{c}_2, \mathsf{p}_1, \mathsf{p}_2) \perp \mathsf{c}_1) \Leftrightarrow \\
\qquad\qquad \mathsf{p}_1 \perp \mathsf{c}_1 \cap \mathsf{c}_2 \wedge \mathsf{p}_2 \perp \mathsf{c}_1 \setminus \mathsf{c}_2) \\
\wedge \quad ((\mathsf{While}(\mathsf{c}_2, \mathsf{p}_1) \perp \mathsf{c}_1) \Leftrightarrow \\
\qquad\qquad \mathsf{p}_1 \perp \mathsf{c}_1 \cap \mathsf{c}_2 \wedge \\
\qquad\qquad \mathsf{p}_1 \perp \mathsf{semantics}\ \mathsf{p}_1\ (\!|\mathsf{c}_1 \cap \mathsf{c}_2|\!) \cap \mathsf{c}_2) \\
\wedge \quad ((\mathsf{Spec}((\mathsf{prec}, \mathsf{postc}), \mathsf{p}_1) \perp \mathsf{c}_1) \Leftrightarrow \\
\qquad\qquad \mathsf{c}_1 \subseteq \mathsf{prec} \wedge \mathsf{p}_1 \perp \mathsf{c}_1)
\end{array}
$$

I am grateful to one of the referees for pointing out that my original definition of uprightness was too weak: it omitted the second conjunct in the clause for while-statements. This kind of error is rather easy to make but would become apparent in further work to derive useful consequences of the uprightness relation formally. In the present work, such consequences were only drawn informally and so the omission was only detected by review.

Uprightness enjoys the following two useful properties:

$$
\begin{array}{l}
\mathsf{upright\_mono\_thm}\ ?\vdash \\
\forall\ \mathsf{prog} : \mathsf{PROG};\ \mathsf{c}_1, \mathsf{c}_2 : \mathsf{PRED} \bullet \\
\qquad \mathsf{prog} \perp \mathsf{c}_1 \wedge \mathsf{c}_2 \subseteq \mathsf{c}_1 \Rightarrow \mathsf{prog} \perp \mathsf{c}_2
\end{array}
$$

$$
\begin{array}{l}
\mathsf{upright\_cup\_thm}\ ?\vdash \\
\forall\ \mathsf{prog} : \mathsf{PROG};\ \mathsf{c}_1, \mathsf{c}_2 : \mathsf{PRED} \bullet \\
\qquad \mathsf{prog} \perp \mathsf{c}_1 \wedge \mathsf{prog} \perp \mathsf{c}_2 \Rightarrow \mathsf{prog} \perp \mathsf{c}_1 \cup \mathsf{c}_2
\end{array}
$$

4 The VC Generator

4.1 Pre-condition Calculation

The verification condition (VC) generators we are concerned with are essentially pre-condition calculators: given a desired post-condition *postc* and a program *prog*, the VCs amount to a pre-condition *prec*, such that whenever *prog* terminates after execution from an initial state satisfying *prec*, then *postc* holds in the terminal state. One then endeavours to show that *prec* is true, in practice by proving a set of syntactic predicates whose conjunction implies *prec*.

In this section we formalise a general notion of a pre-condition calculator and say what it means for such a thing to be sound. In the next section, we will specify a particular pre-condition calculator.

The simplest view of a pre-condition calculator would be a predicate transformer: a function that takes a program and a post-condition given as a predicate on the final state as its argument and returns a predicate that, we hope, gives a pre-condition which will guarantee achievement of the post-condition.

However, the predicate transformer idea is slightly simplistic: to deal with a post-condition such as $\mathsf{Y} = \mathsf{Y}_0/\mathsf{X}$, in which the post-condition depends on values in the before state, something more is needed, to reflect the dependency. (In other accounts, such as [6], auxiliary (non-program) variables are used to capture values at particular points in the program. These introduce a similar problem.)

One possible solution is to deal with post-conditions as predicates parametrised in some way. However, it seems more natural to deal with post-conditions as relations on states, rather than just states. The domains and ranges of these relations corresponds to appropriate initial, final, or intermediate program states, depending on the point in the program being considered. This gives us the following signature for a pre-condition calculator.

$$\mathsf{PREC\_CALC} \mathrel{\widehat{=}} \mathsf{PROG} \times \mathsf{POSTC} \rightarrow \mathsf{POSTC}$$

We can now state what it means for a pre-condition calculator to be sound. Soundness requires, that given any program, prog, and any target post-condition, after, the result of pre-condition calculation is a relation before such that (equipped with standard pre-conditions) the sequential composition before $\fatsemi$ semantics prog is a refinement of after.

$$\begin{array}{|l}
\mathsf{sound_prec_calc} : \mathbb{P}\mathsf{PREC_CALC} \\
\hline
\forall \mathsf{pc} : \mathsf{PREC_CALC} \bullet \\
\quad \mathsf{pc} \in \mathsf{sound_prec_calc} \\
\Leftrightarrow \quad (\forall \mathsf{prog} : \mathsf{PROG};\ \mathsf{before},\ \mathsf{after} : \mathsf{POSTC} \\
\quad \mid \quad \mathsf{before} = \mathsf{pc}(\mathsf{prog},\ \mathsf{after}) \\
\quad \bullet \quad (\mathrm{dom}\ \mathsf{before},\ \mathsf{after}) \sqsubseteq \\
\qquad\quad (\mathrm{dom}\ \mathsf{before},\ \mathsf{before} \fatsemi \mathsf{semantics}\ \mathsf{prog}))
\end{array}$$

So, for example, consider the case when prog is the following if-statement:

$$\mathsf{if}\ \mathsf{X} < \mathsf{Y}\ \mathsf{then}\ \mathsf{X} := \mathsf{Y}\ \mathsf{else}\ \mathsf{Y} := \mathsf{X}\ \mathsf{end}\ \mathsf{if}$$

For a sound pre-condition calculator, $before$ must be a state transformer such that the program:

$$\mathsf{before};\ \mathsf{if}\ \mathsf{X} < \mathsf{Y}\ \mathsf{then}\ \mathsf{X} := \mathsf{Y}\ \mathsf{else}\ \mathsf{Y} := \mathsf{X}\ \mathsf{end}\ \mathsf{if}$$

will have the overall effect of making the desired post-condition $after$ hold.

In the relation returned by a pre-condition calculator, the domain and the range of the relation have both been pulled back to the initial state. The desired pre-condition is extracted by intersecting with the identity relation. This gives the following simple consequence of soundness, if the program has a specification at the top level and if the calculated pre-condition contains the pre-condition of the specification, then the program satisfies the specification:

$$\begin{array}{l}
\mathsf{prec_calc_sat_thm}\ ?\vdash \\
\forall\ \mathsf{pc} : \mathsf{PREC_CALC};\ \mathsf{c_prec},\ \mathsf{s_prec} \colon \mathsf{PREC}; \\
\ \ \mathsf{s_postc} : \mathsf{POSTC};\ \mathsf{p} : \mathsf{PROG} \\
\mid \quad \mathsf{pc} \in \mathsf{sound_prec_calc} \\
\land \quad \mathsf{c_prec} = \\
\quad\ \ \mathrm{dom}\ (\mathsf{pc}(\mathsf{Spec}((\mathsf{s_prec},\ \mathsf{s_postc}),\ \mathsf{p}),\ \mathsf{s_postc}) \cap \\
\quad\ \ (\mathrm{id}\ \mathsf{STATE})) \\
\land \quad \mathsf{s_prec} \subseteq \mathsf{c_prec} \\
\bullet \quad \mathsf{p} \models (\mathsf{s_prec},\ \mathsf{s_postc})
\end{array}$$

Continuing the earlier example, if pc is a sound pre-condition calculator and the desired post-condition s_postc is $\mathsf{X} = \max\{\mathsf{X}_0,\ \mathsf{Y}_0\}$ the calculated pre-condition for the if-statement

$$\mathsf{if}\ \mathsf{X} < \mathsf{Y}\ \mathsf{then}\ \mathsf{X} := \mathsf{Y}\ \mathsf{else}\ \mathsf{Y} := \mathsf{X}\ \mathsf{end}\ \mathsf{if}$$

could be true, whereas if the desired post-condition is $\mathsf{Y} = \max\{\mathsf{X}_0,\ \mathsf{Y}_0\}$, then the calculated pre-condition must be no stronger than $\mathsf{X} = \mathsf{Y}$.

4.2 A Useful Pre-condition Calculator

We can easily exhibit a sound but not at all useful, pre-condition calculator, which simply returns the empty relation. The following theorem says that this is indeed sound.

$$\begin{array}{l}
\mathsf{trivial_prec_calc_sound_thm}\ ?\vdash \\
\forall\ \mathsf{pc} : \mathsf{PREC_CALC} \\
\mid \quad \forall \mathsf{prog} : \mathsf{PROG};\ \mathsf{postc} : \mathsf{POSTC} \bullet \\
\qquad \mathsf{pc}\ (\mathsf{prog},\ \mathsf{postc}) = \varnothing \\
\bullet \quad \mathsf{pc} \in \mathsf{sound_prec_calc}
\end{array}$$

In this section, we give a model of a pre-condition calculator that is more useful than this trivial example. Before giving the definition we need two preliminaries.

The first preliminary comprises a variant on the theme of range restriction and range anti-restriction, which are needed to deal with if-statements. The idea here is this: If R is a set of state transitions and T is some set of target after-states, then the usual range restriction $\mathsf{R} \rhd \mathsf{c}$ contains all before-states that *may* lead to states in T, whereas $\mathsf{R} \rhd_* \mathsf{c}$ contains only the before-states that *must* lead to states in T. $\mathsf{R} \unrhd_* \mathsf{c}$ bears the analogous relationship with $\mathsf{R} \unrhd \mathsf{c}$.

$$\begin{array}{|l}
[X, Y] \\
\hline
_ \rhd_* _ : (\mathsf{X} \leftrightarrow \mathsf{Y}) \times \mathbb{P}\mathsf{Y} \rightarrow (\mathsf{X} \leftrightarrow \mathsf{Y}); \\
_ \unrhd_* _ : (\mathsf{X} \leftrightarrow \mathsf{Y}) \times \mathbb{P}\mathsf{Y} \rightarrow (\mathsf{X} \leftrightarrow \mathsf{Y}) \\
\hline
\forall \mathsf{R} : \mathsf{X} \leftrightarrow \mathsf{Y};\ \mathsf{c} : \mathbb{P}\ \mathsf{Y} \bullet \\
\quad \mathsf{R} \rhd_* \mathsf{c} = \mathsf{R}^{\sim} \llparenthesis\ \mathsf{Y} \setminus \mathsf{c}\ \rrparenthesis \ntriangleleft \mathsf{R} \\
\land \quad \mathsf{R} \unrhd_* \mathsf{c} = \mathsf{R}^{\sim} \llparenthesis\ \mathsf{c}\ \rrparenthesis \ntriangleleft \mathsf{R}
\end{array}$$

The useful pre-condition calculator will use a heuristic to propose a specification for the body of a while-loop. Our remaining preliminary is a loose definition of this heuristic. The definition requires that if the body has been supplied with an explicit specification, then that specification should be used. The pre-condition calculator must therefore ensure soundness without relying on any properties of this heuristic. (A practical realisation of the pre-condition calculator might simply insist that the body of a while-loop be given with an explicit specification).

$$\begin{array}{|l}
\mathsf{guess_spec} : \mathsf{PROG} \rightarrow \mathsf{SPEC} \\
\hline
\forall \mathsf{s} : \mathsf{SPEC};\ \mathsf{p} : \mathsf{PROG} \bullet \mathsf{guess_spec}(\mathsf{Spec}(\mathsf{s},\ \mathsf{p})) = \mathsf{s}
\end{array}$$

With the preliminaries in place, we can now define the useful pre-condition calculator, which we think of as pulling a post-condition backwards through a program transforming it as we go. We give the Z first and then give a clause-by-clause commentary:

$$
\begin{array}{|l}
\mathsf{prec\_calc} : \mathsf{PREC\_CALC} \\
\hline
\forall \mathsf{t} : \mathsf{STATE\_TRANSFORMER};\ \mathsf{postc},\ \mathsf{postc}_1 : \mathsf{POSTC}; \\
\quad \mathsf{p}_1,\ \mathsf{p}_2 : \mathsf{PROG};\ \mathsf{c} : \mathsf{PRED};\ \mathsf{prec}_1 : \mathsf{PREC}; \\
\quad \mathsf{body\_prec} : \mathsf{PRED};\ \mathsf{body\_postc} : \mathsf{POSTC} \bullet \\
\qquad \mathsf{prec\_calc}\ (\mathsf{Atom}\ \mathsf{t},\ \mathsf{postc}) = \\
\qquad\qquad \{\mathsf{s},\ \mathsf{s}' : \mathsf{STATE} \mid \mathsf{t}(\!|\{\mathsf{s}'\}|\!) \subseteq \mathsf{postc}(\!|\{\mathsf{s}\}|\!)\} \\
\land \quad \mathsf{prec\_calc}\ (\mathsf{Seq}(\mathsf{p}_1,\ \mathsf{p}_2),\ \mathsf{postc}) = \\
\qquad\qquad \mathsf{prec\_calc}(\mathsf{p}_1,\ \mathsf{prec\_calc}(\mathsf{p}_2,\ \mathsf{postc})) \\
\land \quad \mathsf{prec\_calc}\ (\mathsf{If}(\mathsf{c},\ \mathsf{p}_1,\ \mathsf{p}_2),\ \mathsf{postc}) = \\
\qquad\qquad (\mathsf{prec\_calc}(\mathsf{p}_1,\ \mathsf{postc}) \rhd_* \mathsf{c}) \cup \\
\qquad\qquad (\mathsf{prec\_calc}(\mathsf{p}_2,\ \mathsf{postc}) \ntriangleright_* \mathsf{c}) \\
\land \quad ((\mathsf{body\_prec},\ \mathsf{body\_postc}) = \mathsf{guess\_spec}\ \mathsf{p}_1 \\
\Rightarrow \quad \mathsf{prec\_calc}\ (\mathsf{While}(\mathsf{c},\ \mathsf{p}_1),\ \mathsf{postc}) = \\
\qquad\qquad \mathsf{postc} \ntriangleright \mathsf{c} \cup \\
\qquad\qquad \{\ \mathsf{ss}' : \mathsf{dom}\ \mathsf{postc} \times \mathsf{c} \\
\qquad\qquad \mid \mathsf{c} \subseteq \mathsf{body\_prec} \\
\qquad\qquad \land\ \mathsf{body\_prec} \subseteq \\
\qquad\qquad\qquad \mathsf{dom}\ (\mathsf{prec\_calc}(\mathsf{p}_1,\ \mathsf{body\_postc}) \cap \\
\qquad\qquad\qquad (\mathsf{id}\ \mathsf{STATE})) \\
\qquad\qquad \land\ \mathsf{dom}\ \mathsf{postc} \times (\mathsf{body\_postc}(\!|\mathsf{body\_prec}|\!) \setminus \mathsf{c}) \subseteq \\
\qquad\qquad\qquad \mathsf{postc}\ \}) \\
\land \quad \mathsf{prec\_calc}\ (\mathsf{Spec}((\mathsf{prec}_1,\ \mathsf{postc}_1),\ \mathsf{p}_1),\ \mathsf{postc}) = \\
\qquad\qquad \{\ \mathsf{s} : \mathsf{STATE};\ \mathsf{s}' : \mathsf{STATE} \\
\qquad\qquad \mid \mathsf{postc}_1(\!|\{\mathsf{s}'\}|\!) \subseteq \mathsf{postc}(\!|\{\mathsf{s}\}|\!)\ \} \rhd \\
\qquad\qquad (\mathsf{prec}_1 \cap \mathsf{dom}(\mathsf{prec\_calc}(\mathsf{p}_1,\ \mathsf{postc}_1) \cap \\
\qquad\qquad (\mathsf{id}\ \mathsf{STATE})))
\end{array}
$$

In the definition, the 5 clauses deal with the various syntactic categories as follows:

- A post-condition is pulled back through an atomic statement, by calculating the set of pairs (s, s') such that the response of the atom on s' is a response permitted by the post-condition on s.

- A post-condition is pulled back through the sequential composition of p_1 and p_2 in the obvious way: pull it back through p_2 and then pull the result back through p_1.

- A post-condition is pulled back through an if-then-else statement by pulling it back through the then- and else-parts of the statement. The overall result is then the union of these intermediate results after discarding all transitions which do not unambiguously belong to the if-part or the else-part.

- A post-condition is pulled back through a while loop by applying the heuristic to guess a specification for the body of the loop. The overall result is formed as a union of two parts.

  The first part of the union corresponds to states where the body of the loop is never execute and is just the appropriate restriction of the original post-condition.

  The second part corresponds to states where the body of the loop is executed at least once and is given as a set comprehension below. The set comprehension is empty unless three conditions are satisfied: *(i)* the condition of the while-loop must denote a set of states that are included in the guessed pre-condition; *(ii)* the pre-condition of the body must denote a set of states that satisfy the pre-condition resulting from pulling the guessed post-condition back through the body; and *(iii)* for each state satisfying the guessed pre-condition, the set of all states allowed by the guessed post-condition in response to this state which do not satisfy the loop condition must be contained in every possible response of the original post-condition.

- A post-condition is pulled back through a specification statement in much the same way as it is pulled back through an atomic statement treating the post-condition of the specification statement in the same way as the state transformer of the atom. The result is then filtered to remove all state transitions which do not unambiguously satisfy both the pre-condition of the specification statement and the pre-condition calculated from the body of the specification statement.

4.3 Implementation Considerations

Before formalising theorems about the useful pre-condition calculator, some remarks about how something like it is realised in a practical system are in order.

An implementation can represent the post-condition being transformed as (the conceptual conjunction of) a finite set of syntactic predicates $\mathcal{P}_i(\vec{x_0}, \vec{x})$, where $\vec{x}$ represents some list of program variables and $\vec{x_0}$ represents a list of program variables decorated to distinguish them as initial variables (i.e., they refer to the before-state of the code being analysed). The pre-condition calculator will operate by syntactic transformations on these predicates which hold the initial variables fixed but may make substitutions to $\vec{x}$. At the beginning of the calculation, $\vec{x}$ refers to the final state of the program, and as the calculation works backwards through the code, the execution state referred to by $\vec{x}$ moves backwards in step.

The most primitive state-changing operation will be the atomic statements that represent program language assignments. Given an assignment, $v := e$, the requirements of the above formal definition are precisely met by substituting e for v in the $\mathcal{P}_i(\vec{x_0}, \vec{x})$. Here we are tacitly assuming that program variables and expressions have some well-defined representation as logical variables and expressions in the logical system in use.

Procedure calls are the other common form of atomic statements and as already discussed these can be treated much as specification statements (with empty bodies).

If the programming language has them, then other forms of atomic statements can be dealt with in an *ad hoc* way as their semantics dictates. For example, many programming languages have a null statement form, which corresponds to the identity operation on the set of predicates. An atomic statement that aborted execution could be dealt with by delivering an empty set of predicates, or equivalently, the single predicate *true*, (see example pre-condition calculations at the end of this section).

Sequential composition can be handled exactly as in the formal definition: the set of predicates calculated for the second statement is just passed in as the target post-condition for the first statement.

If-then-else statements cause sets of predicates to be combined. Each $\mathcal{P}_i(\vec{x_0}, \vec{x})$ resulting from analysing the then-part of the conditional with condition c would contribute $c \Rightarrow \mathcal{P}_i(\vec{x_0}, \vec{x})$ to the result. Similarly, each $\mathcal{P}_j(\vec{x_0}, \vec{x})$ resulting from the analysis of the else-part would contribute $\neg c \Rightarrow \mathcal{P}_j(\vec{x_0}, \vec{x})$.

While-loops are handled by logical transformations that mimic the various parts of the set comprehension in the formal definition above. There are various possible approaches, some of which necessitate a more complex representation of the post-condition involving quantifiers, rather than a flat conjunction of quantifier-free formulae. The Compliance Notation avoids this complexity by generating what are called side conditions, universally closed conjectures that have to be proved to justify the correctness of the main calculation. From a user's perspective the end result of the whole process is just a set of verification conditions (VCs) that have to be proved and these side conditions just get added to the final set of VCs. For example, in a loop with condition c, if the post-condition *postc* in the formal definition above is represented by the set of syntactic predicates $\mathcal{A}_j$, a side condition of the form $\mathcal{P}_i(\vec{x_0}, \vec{x}) \wedge c \Rightarrow \mathcal{A}_j$ is generated for each $\mathcal{P}_i(\vec{x_0}, \vec{x})$ in the representation of *body_postc*. This corresponds to the requirements of the last conjunct in the set comprehension above.

Like while-loops, if full generality is to be achieved, specification statements require a more complex representation using quantifiers. Again, the Compliance Notation adopts the simpler approach of generating side conditions, if necessary. For example, side conditions of the form $\mathcal{P}_i(\vec{x_0}, \vec{x}) \wedge c \Rightarrow \mathcal{A}_j$ will be generated for each $\mathcal{P}_i(\vec{x_0}, \vec{x})$ in the representation of what is called postc_1 above and for each $\mathcal{A}_j$ in the representation of postc. This corresponds to the predicate of the set comprehension above.

4.4 Properties of the Pre-condition Calculator

We now return to the formal work and present a number of conjectures all of which have been proved with ProofPower. We conjecture that the useful pre-condition calculator is sound. Note that since soundness is defined quite directly in terms of the notion of refinement and of the programming language semantics, this is strong evidence that the definition of the pre-condition calculator is strong enough.

```
prec_calc_sound_thm ?⊢ prec_calc ∈ sound_prec_calc
```

The following conjecture gives a useful property of our useful pre-condition calculator, which turns out to be a simple consequence of its soundness.

```
prec_calc_dom_thm ?⊢
∀ prog: PROG; postc : POSTC
•    dom (prec_calc(prog, postc) ∩ (id STATE)) ∩
     dom (semantics prog)
              ⊆ dom postc
```

We also conjecture that a program is upright in every state in the pre-condition produced by the pre-condition calculator, i.e., no execution of the program can cause the pre-condition of any specification in the program to be violated in those states. Taken together with the soundness conjecture, this shows that a VC generator based on the pre-condition calculator does indeed guarantee program correctness as discussed in section 3.2 above.

```
prec_calc_upright_thm ?⊢
∀ prog: PROG; postc, postc′ : POSTC
|    postc = prec_calc(prog, postc′)
•    prog ⊥ ran postc
```

Finally, we give some evidence that the useful pre-condition calculator really is useful by exhibiting some simple examples for which it returns something more interesting than an empty relation.

The first block of examples covers various forms of atom.

```
prec_calc_atom_egs_thm ?⊢
∀ null, chaos, stop : PROG; postc : POSTC
|    null = Atom (id STATE)
∧    chaos = Atom (STATE × STATE)
∧    stop = Atom ∅
•    prec_calc(null, postc) = postc
∧    prec_calc(chaos, postc) =
          {s, s′: STATE | postc⦇{s}⦈ = STATE}
∧    prec_calc(stop, postc) = STATE × STATE
```

The second block gives at least one example of each of the compound syntactic categories.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{prec\_calc\_compound\_egs\_thm}\ ?\vdash} \\
\multicolumn{2}{l}{\forall\ \mathsf{null, chaos, stop, p, spec\_null : PROG;}} \\
 & \mathsf{postc : POSTC;\ c : PRED} \\
| & \mathsf{null = Atom\ (id\ STATE)} \\
\wedge & \mathsf{chaos = Atom\ (STATE \times STATE)} \\
\wedge & \mathsf{stop = Atom\ \varnothing} \\
\wedge & \mathsf{spec\_null = Spec((STATE,\ id\ STATE),\ null)} \\
\bullet & \mathsf{prec\_calc(If(c,\ null,\ stop),\ postc) =} \\
 & \mathsf{postc \rhd_* c \cup (STATE \times STATE) \unrhd_* c} \\
\wedge & \mathsf{prec\_calc(Seq(p,\ null),\ postc) = prec\_calc(p,\ postc)} \\
\wedge & \mathsf{prec\_calc(Seq(null,\ p),\ postc) = prec\_calc(p,\ postc)} \\
\wedge & \mathsf{prec\_calc(While(STATE,\ spec\_null),\ postc) =} \\
 & \mathsf{dom\ postc \times STATE} \\
\wedge & \mathsf{prec\_calc\ (spec\_null,\ postc) = postc}
\end{array}
$$

These calculational results give evidence that the definition of the useful pre-condition calculator is not too strong and there is some value in them: it was only when I tried to prove them that I realised that I had mistakenly written:

$$\mathsf{\{s'\} \lhd t \subseteq \{s\} \lhd postc}$$

instead of

$$\mathsf{t(\!|\{s'\}|\!) \subseteq postc(\!|\{s\}|\!)}$$

in the equation for the semantic category Atom. The pre-condition calculator is sound and guarantees uprightness with this mistake, but very far from useful (since the mistaken predicate prohibits the state from changing if $\mathsf{\{s'\} \lhd t}$ is not empty).

5 Some Statistics

As the reader can see the specifications and the statements of the theorems as given here occupy only about 8 or 9 pages even when heavily leavened with narrative. The corresponding parts of the account in the case study document [1] is actually about 10 pages, but that is mainly due to differences in layout. The case study document also includes 11 pages of automatically generated listings and an index. When the narrative is stripped out the specification is about 200 lines of Z text. The proof script comprise about 380 lines of Standard ML code. The original development of both specifications and proofs took about 5 days.

As has been mentioned, the material was subsequently revised to use a generic theory of specification and refinement. This adaptation was very simple and only took an hour or two (the only material affected is that in section 2 above, which becomes a little shorter). The generic theory of refinement in [2] is very similar in size to the present specification and the proof script is also closely comparable both in size and in development time.

Correcting the omission in the definition of uprightness mentioned in section 3.2 presented some evidence that the specification and proof infrastructure was reasonably robust: it took only about 30 minutes to develop the handful of additional lines of proof script to show that the useful pre-condition calculator is correct relatively to the stronger notion of uprightness. Moreover, once the specification was amended the proof work was purely mechanical: no significant intellectual effort was required.

6 Conclusions

A formal specification in Z of a simple imperative programming language equipped with specification annotations has been presented together with an abstract formal design of a verification condition generator for that language. A number of theorems have been stated which together show that the verification condition generator is sound with respect to the programming language semantics and guarantees a property we have called *uprightness* which expresses formally part of the intensionality of the specification annotations. Machine-checked proofs of these theorems were prepared using ProofPower. The work was of some benefit in gaining a better understanding of a real-world program verification system (the Z/Ada Compliance Tool) and in enhancing the capabilities of that system.

By choosing a suitable level of abstraction, it was a relatively small task to conduct fully machine-checked proofs of all the conjectures. This helped give a deeper understanding and revealed some flaws. It is noteworthy that even such a familiar thing as a Floyd-Hoare logic for a simple programming language contains pitfalls for the unwary and some surprises even for the expert.

It is becoming apparent that the ideas behind Floyd-Hoare logic have application beyond the sphere of functional correctness of imperative programming languages. Potential applications include resource management in programming [4] and, going further afield, continuous systems expressed diagrammatically [5]. There are theoretical indications that the Floyd-Hoare approach applies to a very wide class of systems indeed [9].

Some experimentation has begun in applying the approach of the present paper to other systems (including one of the two main general types identified in [9]) with continuous systems as a goal. In these less familiar applications, a fully rigorous approach to the metatheory promises to give important insights and to help build sound tools for specification and verification in applications.

Acknowledgments

I am grateful to the referees and to the participants at the Z 2006 meeting for their very helpful comments.

References

[1] R. Arthan. On correctness of imperative programs — precondition calculation. Working Paper WRK071, Lemma 1 Ltd., 2002. `http://www.lemma-one.com`.

[2] R. Arthan. On refinement calculus and partial correctness. Working Paper WRK069, Lemma 1 Ltd., 2002. `http://www.lemma-one.com`.

[3] R. Arthan and R. Jones. Z in HOL in ProofPower. *BCS FACS FACTS*, 2005-1.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270. ACM Press, 2005.

[5] R. Boulton, R. Hardy, and U. Martin. A hoare logic for single-input single-output continuous-time control systems. In O. Maler and A. Pnueli, editors, *Hybrid Systems: Computation and Control*, pages 113–125. Springer-Verlag, 2003.

[6] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 841–993. Elsevier, Amsterdam, 1990.

[7] M. J. Gordon. *Programming Language Theory and its Implementation*. Prentice/Hall International, 1988.

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

[9] U. Martin, E. A. Mathiesen, and P. Oliva. Hoare logic in the abstract. *To appear*, 2006.

[10] C. Morgan. *Programming from Specifications*. Prentice/Hall International, 1990.

[11] C. O'Halloran. Model based code verification. In *ICFEM*, pages 16–25, 2003.

[12] C. O'Halloran, R. D. Arthan, and D. King. Using a formal specification contractually. *Formal Asp. Comput.*, 9(4):349–358, 1997.

[13] C. O'Halloran and A. Smith. Don't verify, abstract! In *ASE*, pages 53–62, 1998.

[14] C. T. Sennett. Demonstrating the compliance of ada programs with z specifications. In R.Shaw, editor, *5th Refinement Workshop*, Workshops in Computing, pages 88–118. Springer-Verlag/BCS-FACS, 1992.

[15] J. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.