SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 98-24

From Formal Specifications to Ada Programs

Peter Kearney and Luke Wildman

October 1998

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

To appear in *Proceedings of the* 22nd *Australasian Computer Science Conference '99*, Auckland, 1999, Springer-Verlag, Australian Computer Science Communications, Vol 21, No 1.

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

From Formal Specifications to Ada Programs

Peter Kearney and Luke Wildman

Abstract

This paper describes an approach to embedding the semantics of a target programming language within the Z-based formal specification and development language Sum. The approach enables formal tool-supported refinement from specifications to a Sum subset which is then translated to Ada.

Keywords: Formal specification, Sum Intermediate Language(IL), Refinement, Ada

1 Introduction

Formal, mathematically based software development methods provide a basis for achieving very high levels of assurance of the correctness of software. The Cogito system [9] is an integrated method and tool set supporting formal development from high-level requirements specification (expressed in Sum, an extended Z specification language) to production of executable code. Previous papers have described the system [10, 11] and its use in a complete formal development [2].

This paper focuses on how the semantics of target programming languages is represented in the system and translation of specifications to target programming languages achieved. The approach used is quite generic, but currently Ada is the only target language supported by the translation tools.

Our approach is to embed the semantics of a simple but useful programming language within the semantics of Sum. This 'Intermediate Language' (IL) subset of Sum is translatable to Ada. The embedding uses the 'predicative programming' paradigm for defining the semantics of IL statements as Sum predicates which allows integration with general Sum specification predicates. To our knowledge this is the first time the predicative programming approach has been used in a Z context. Other approaches to development from Z specifications typically translate to another language before detailed development, for example to the refinement calculus [6] or a mixed Ada/Z development language as in [5]. Our approach provides better integration of the development language semantics with the specification language semantics.

The next section gives brief background on Cogito and the Sum language. Section 3 discusses the translation of Sum module structures to Ada package structures. Section 4 outlines how data types corresponding to a subset of Ada data types are modeled in Sum and translated to Ada. Section 5 describes

how the semantics of imperative statements is embedded in the semantics of Sum using the 'programs are predicates' paradigm[3, 4, 7, 8]. This section also discusses issues relating to the translation to Ada statements. Section 6 provides some more extended discussion on issues relating to modular development and the treatment of input and output variables. Finally, Section 7 provides some conclusions and discussion of further work.

2 Background: Sum and development to code

The specification language Sum [10] extends Z with various facilities that provide more comprehensive support for specification, validation, reasoning, and refinement. In particular, Sum provides facilities for:

- modular and parameterised specifications,
- distinguished state machines within a module (via distinguished *State*, *init* and *Op*eration schemas), and
- explicit preconditions on schemas.

The Sum intermediate language (IL) is a subset of the full Sum specification and development language. It can be translated to a variety of imperative languages. Currently automatic translation to Ada is supported.

In development by refinement as supported by Cogito, an abstract specification is transformed step-wise into a form suitable for implementation and, at intermediate stages, development is represented by a mixture of abstract specification and IL code. (See for example the development described in [2]). Formal proof of refinement conditions is carried out using the Ergo proof tool [1]. The Cogito Ada translator supports the translation of such mixtures by translating IL components to Ada and more general Sum to comments in the Ada code. Sum specifications in 'pure' IL are translated to fully executable Ada code.

Figure 1 shows a Sum module for a simple symbol table implementation. This module is the result of the refinement process described in [2] and is in 'pure IL' suitable for translation to executable Ada.

3 Translation of Sum Modules

A Sum module is composed of a collection of declarations and may be used in other modules via *import*. All the declarations within an imported module can be accessed by qualifying their name with the name of the imported module. Modules can be parameterised with types, values, and relations. Parameters of a module must all be fully instantiated when a module is imported (that is, no 'partial parameterisation' is permitted). The Sum *import as* construct creates a new instance of a module with a new module identifier. This form of import must be used when importing parameterised modules and is also used when

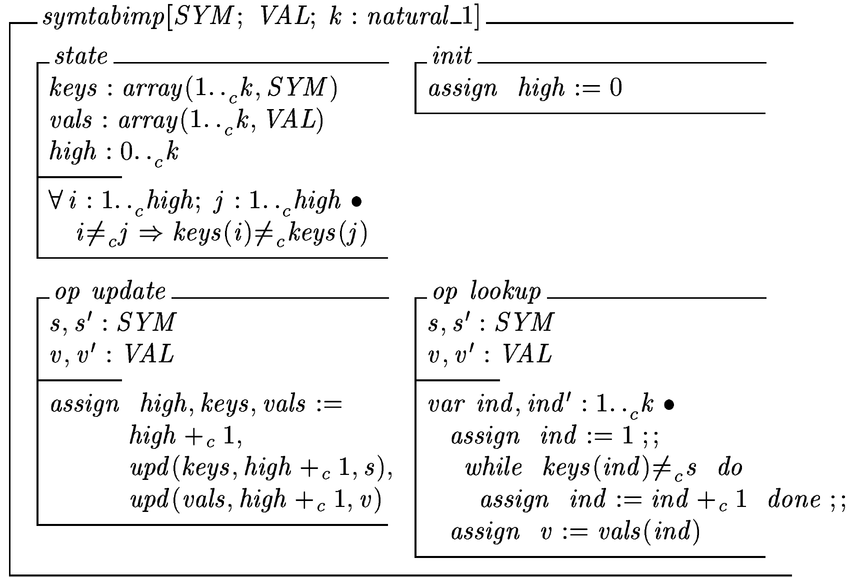

Figure 1: "symtabimp.sum"

importing non-parameterised modules that have a state component that we do not want to share with other importations of the module.

The sum module *symtabimp* (Figure 1) is translated to a generic Ada package with both a specification and a body (Figure 2). The translator generates generic Ada packages for both generic and non-generic Sum modules. Generic instantiation in Ada can then be used to implement Sum *import-as*. In addition, non-generic packages are generated for non-generic modules in order to support direct *import* in Sum. On translation to Ada, the module type parameters SYM and VAL become formal generic private types declared in the package specification. The constant parameter k becomes a generic formal object.

Figure 3 shows the importation of the module *symtabimp* as a new instance *sti*, using *import-as*. Figure 4 shows the translation of this to package instantiation in Ada. Figure 3 also illustrates the capability for Sum to be used to model the Ada libraries. As Sum supports generic submodules, the Ada library structure can be directly modeled in Sum allowing, for instance, reference to Ada IO in an IL specification.

Sum provides a *visible* declaration which makes entities in an imported module directly accessible without the need for a qualified name. It does not make the declarations in transitively imported modules visible. For example, it is used in Figure 3 to make the declarations within *Text_IO* directly visible. The Ada **use** clause broadly corresponds to the Sum *visible* declaration. However, Sum provides *linear visibility*. For example, suppose Sum modules A and B

both declare f, and both A and B are imported and made visible in module C. If B is made visible after A then linear visibility determines that $B.f$ is visible in C. In Ada, fully qualified names are required to resolve the resulting name clash.

Returning to the translation of the Sum module in Figure 1, note that the variables occurring in the *state* schema of the Sum module are translated to public variables in the package specification. The state invariant of the Sum module, guaranteed by the refinement to be preserved by each of the operations, is inserted as a comment to that effect. Section 6.1 below gives the rationale for translating state to public rather than private variables.

IL operation schemas are translated to procedures in the package body (see Figure 2). The special operation *init* becomes a procedure which is called by the package initialisation part. The only allowable IL statement for initialisation is a (multiple) assignment.

Parameters to IL operations are modeled by pre- and post-state variables using the conventional manner of representing state-change in Z. For example, the parameters to the operation *update* in Figure 1, *s* and *v*, occur both unprimed and primed. In translation, the primed variable is recognized as the post-state version of the unprimed variable, with the result that a single Ada **in out** parameter is generated corresponding to each primed, unprimed pair.

Note that the IL does not use Z-style input variables (decorated with '?') and output variables (decorated with '!') for operation parameters. The motivation for this is discussed in 6.2 below.

4 IL data types and their translation

The IL currently includes types which model the following Ada types: boolean, integer, character, string, enumeration types, subrange types, constrained array types and simple record types. The following subsections discuss the modeling of some of these types in Sum and their translation to Ada. The IL does not currently support any modeling of Ada real types, variant record types, access types or derived types.

4.1 Computational primitive types

To ensure accurate modeling of implementation and overflow behavior, the IL models operations applicable to Ada's boolean and integer types with specially defined 'computational' operators rather than Sum's general operators for *int* and *bool* types.

The types *integer*, *natural* and *natural_1* are defined as sub-types of the theoretical (infinite) *int* type and are represented by specific (machine-dependent) ranges of integers.

$$
\begin{array}{l}
min_int, max_int : int \\
integer == \{i : int \mid min_int \leq i \land i \leq max_int\}
\end{array}
$$

Package Specification

```
generic
  type SYM is private;
  type VAL is private;
  k: positive;
package gen_symtabimp is
  high : integer range 1 .. k;
  keys : array(1 .. k) of SYM;
  vals : array(1 .. k) of VAL;
  – forall i : 1 ..c high; j :1 ..c high @ i /=c j => keys(i) /=c keys(j)
  procedure init;
  procedure update(s : in out SYM; v:in out VAL);
  procedure lookup(s: in out SYM; v:in out VAL);
end gen_symtabimp;
```

Package Body

```
package body gen_symtabimp is
  procedure init is begin high := 0; end;
  procedure update(s: in out SYM;
                   v: in out VAL) is
  begin
    declare
      high_temp: integer range 0 .. k;
      keys_ind: integer range 1 .. k;
      keys_temp: SYM;
      vals_ind: integer range 1 .. k;
      vals_temp: VAL;
    begin
      high_temp := high + 1;
      keys_ind := high + 1;
      keys_temp := s;
      vals_ind := high + 1;
      vals_temp := v;
      high := high_temp;
      keys(keys_ind) := keys_temp;
      vals(vals_ind) := vals_temp;
    end;
  end;
  procedure lookup(s: in out SYM;
                   v: in out VAL) is
  begin
    declare
      ind: integer range 1 .. k;
    begin
      ind := 1;
      while (keys (ind) /= s) loop
        ind := ind + 1;
      end loop;
      v := vals (ind);
    end ;
  end;
begin
  init;
end gen_symtabimp;
```

Figure 2: (Generic) Symtab Implementation

Machine oriented arithmetic operators are defined so that results are only guaranteed to be defined when results are in range. (Thus they are partial operations on *int*, the theoretical integer type.) For example, $+_c$ is a computational operator specified as follows.

$$+_c : integer \times integer \nrightarrow integer$$

$$\forall\, x, y : integer \bullet (min_int \leq x + y \wedge x + y \leq max_int) => x +_c y = x + y$$

Note that the result of the computational addition is defined only when the (theoretical) sum is within bounds. Integer subranges are modeled by the $.._c$ operator. Other computational arithmetic (and boolean) operators may be characterised similarly.

```
symtabuser
  import Ada; visible Ada
  import Text_IO; visible Text_IO
  import Integer_IO(integer) as Int_IO
  Sym ::= enum(s1, s2, s3, s4)
  import symtabimp(Sym, integer, 3) as sti

  state
    sti.state

  op op1
    x, x' : integer
    sx, sx' : Sym
    y, y' : integer
    ----
    call (sti.update , (s => sx, v => x), ());;
    call (sti.lookup , (s => sx), (v => y));;
    call (Int_IO.Put , (Base => 10, Width => 13, Item => y), ())
```

Figure 3: “symtabuser.sum”

4.2 Enumeration types

In languages such as Ada and Pascal, when an enumeration type is introduced, a number of ordering operators are implicitly introduced. To support reasoning about these operators, a number of declarations and axioms are required. To allow compact representation of enumeration types Sum IL includes specific notation for enumeration types.

An enumerated type is written

$$E ::= enum(C_1, \ldots, C_n)$$

and it implicitly declares the type E consisting of the constants $C_1, \ldots C_n$ together with associated operators, *succ*, $<$, etc. For example, Figure 3 uses an enumerated type to define the type *Sym*. Figure 4 shows its translation to Ada.

An enumerated type is equivalent to the following definitions in Sum:

E ::= $C_1 \mid C_2 \mid \ldots \mid C_n$;
import enum(E) *as E_enum*;
visible E_enum;

where the module *enum* includes declarations for each of the required operators.

Package Specification

```
with Ada; with Text_IO;
with gen_symtabimp;
package symtabuser is
  use Ada; use Text_IO;
  package Int_IO is new Integer_IO(integer);
  type Sym is (s1, s2, s3, s4);
  package sti is new gen_symtabimp(Sym, integer, 3);
  procedure op1(x : in out integer; sx: in out Sym; y : in out integer);
end symtabuser;
```

Package Body

```
package body symtabuser is
  procedure op1(x : in out integer; sx : in out Sym; y : in out integer) is
  begin
    sti.update(s => sx, v => x);
    sti.lookup(s => sx, v => y);
    Int_IO.Put(Base => 10, Width => 13, Item => y);
  end;
end symtabuser;
```

Figure 4: Symtab User

enum $[T]$

$pos : T \rightarrow \mathbb{N}$	$_ = _ : T \times T \rightarrow \mathbb{B}$	$_ \neq _ : T \times T \rightarrow \mathbb{B}$
$val : \mathbb{N} \nrightarrow T$	$_ < _ : T \times T \rightarrow \mathbb{B}$	$_ > _ : T \times T \rightarrow \mathbb{B}$
$succ : T \nrightarrow T$	$_ \leq _ : T \times T \rightarrow \mathbb{B}$	$_ \geq _ : T \times T \rightarrow \mathbb{B}$

The corresponding Ergo theory includes axioms defining the behavior of the operators.

4.3 Arrays

Constrained (fixed-length) array types are represented in the IL by total functions from finite, discrete types to IL types.

An array type, written $array(indextype, rangetype)$ denotes the mathematical type $indextype \rightarrow rangetype$. Array access is represented by function application. Access out of bounds corresponds to function application off domain, and yields an undefined (error) value. Any such erroneous applications will be discovered during proof of validation and refinement conditions.

Component update is represented by a version of function override, written *upd*, which does not extend the domain of a function.

Note that only 'whole array' update is available in the IL. This provides a simple semantics for array update, enabling the simplification of validation and refinement proofs. However, to achieve greater efficiency, the Ada translator translates these whole array updates to suitable array component updating operations. Translation of assignment, including array assignment is discussed in

Section 5.7.

5 IL statements and their translation

5.1 Predicative programming and the semantics of statements

Sum utilises the usual Z convention for representing state change using primed and unprimed variable names. In general state change is represented by a predicate relating primed and unprimed variable pairs. Moving from Sum specifications to an imperative language requires mapping Sum state change predicates to statements implementing state change in the target imperative language.

The Sum intermediate language includes 'statements' which can be readily translated to corresponding Ada statements. The semantics of these statements is defined by predicates relating primed and unprimed variable pairs. This semantics provides a basis for refining general state change predicates of Sum to IL statements. (See [2] for more details of such a refinement).

For example, the meaning of the IL multiple assignment

$$assign\ x1, \ldots xn := E1, \ldots En$$

where $E1, \ldots En$ are independent of any dashed variables, is (somewhat informally expressed) that

1. $x1' = E1 \wedge \ldots \wedge xn' = En$, and
2. $changes_only(x1, .., xn)$, i.e., for all variable pairs Y, Y' in the current signature such that `Y` does not appear on the left of the assignment, $Y' = Y$.

The approach of specifying statement semantics by predicates ('predicative programming') has been investigated by a number of authors (see, for example, [3, 4, 7, 8].) In Cogito we use this approach to specify imperative language statement semantics in the same framework as that of Sum/Z schemas and predicates.

The semantics of a Sum schema (and its model in Ergo) is a pair $[\, Sig \mid A \,]$ of a *signature* and a *predicate*. Predicates are modeled as mappings from *situations* to *boolean* values. A *situation* maps identifiers to their values and types. Thus for each situation (dependent on the context in which a schema is used) the predicate part yields a truth value based on the current situation.

5.2 Program composition

As a preliminary step to defining the sequential (or program) composition of two predicates we define the relational composition of two predicates A and B, $A \mathbin{;} B$ to be true in a situation s if and only if

1. there exist two situations s_1 and s_2 such that s_1 binds the dashed variables in s and s_2 binds matching undashed variables to the same values and
2. A is true in the situation s overridden by s_1 and
3. B is true in the situation s overridden by s_2.

This relational composition, a predicate level version of the standard Z schema relational composition, is not directly suitable for representing program composition because $A \fatsemi B$ succeeds in relating input and output states if there is *some* intermediate state satisfying A's postcondition and B's precondition. However, refinement of A permits the range of possible results specified by A to be narrowed, thus potentially removing the successful path through $A \fatsemi B$. This is undesirable because we wish refinement of components to maintain success of the overall program.

For this reason, Sum uses a 'demonic' form of composition $A \;;;\; B$ which is true if the relational composition $A \fatsemi B$ is true and *every* final state of A satisfies B's precondition. Such a form of composition is also used in [8], for example.

5.3 If statements

An intermediate language 'if' statement predicate is defined by

$$ifc(A, B, C) = (A \wedge_z B) \vee_z (not_c(A) \wedge_z C)$$

Here $\wedge_z$ is a 'lifted' $\wedge$ operating on Sum predicates such that $A \wedge_z B$ is true in a situation s when A is true at s and B is true at s. The operator not_c is a lifted version of the 'computational' negation operator, which is defined so that if A is undefined, not_c evaluates to false. In that circumstance the whole of $ifc(A, B, C)$ will evaluate to false, which in the predicative programming paradigm represents a non-terminating computation.

5.4 Loops

To define a semantics for while loops, we use the standard approach of introducing a complete partial order and defining loops and other recursive constructs with respect to that ordering.

As a preliminary to defining the ordering, it is necessary to introduce the definition of refinement. We say that predicate A is refined by B relative to a signature sig ($A \sqsubseteq_{sig} B$) if for all situations compatible with sig,

1. the precondition of A implies the precondition of B (that is, the precondition may be weakened) and
2. When the precondition of A is true and B is true, A is also true (That is, the postcondition may be strengthened).

This is expressed formally as follows, where the notation $sig \Vdash P$ means that the predicate P is true in all situations type compatible with the signature sig.

$$A \sqsubseteq_{sig} B \mathrel{\widehat{=}} sig \Vdash (pre(A) \Rightarrow_z (pre(B) \wedge (B \Rightarrow_z A)))$$

We also define the ordering $A \leq_{sig} B$ as meaning that in all situations compatible with sig, A implies B, that is,

$$A \leq_{sig} B \mathrel{\widehat{=}} sig \Vdash (A \Rightarrow_z B)$$

The ordering defined by

$$A \sqsubseteq_{c,sig} B \mathrel{\widehat{=}} (A \leq_{sig} B) \wedge (A \sqsubseteq_{sig} B)$$

is a complete partial order and the basic programming constructs ;; and *if* (and *ifc*) are monotonic with respect to it. This ordering is also used by Nguyen [7]. The refinement order $\sqsubseteq_{sig}$ itself was suggested in [8] as an appropriate order in which to take least fixed points. However this ordering is unsuitable since a sequence of predicates ascending in the refinement order may not have a least upper bound. The simplest way to see this is through a simple example. Consider the chain of predicates $P_i, i = 0, 1, ...$ where

$$P_i = x' \in (\mathbb{N} - \{0, ..., i\})$$

This is ascending in the refinement order and each P_i has a precondition of *true*. (That is, in all initial states, a final state exists satisfying the predicate.) However, the limit to which this is tending is clearly $x \in \{\}$ which has a precondition of *false* and so is not a refinement of any P_i.

The order $\sqsubseteq_{c,sig}$ does not permit any reduction in nondeterminism and thus does not exhibit this problem. Reduction of non-determinism is not required in an *approximation* ordering where we wish to define sequences of approximations to a possibly non-deterministic fixed point.

We now define $\mu_{sig}\ X\ TERM$ to be the least fixed point (if it exists) with respect to $\sqsubseteq_{c,sig}$ of the operator defined by $\lambda\ X\ TERM$.

The IL *while* statement predicate is defined as follows.

$$while(P, Q) = \lambda\, s : Sitns(\mu_{sig(s)}\ X\ ifc_z(P, Q\ ;;\ X, skip)) \bullet s$$

Note that since *ifc* and ;; are monotonic with respect to $\sqsubseteq_{c,sig}$, the least fixed point exists. The semantics defined here provides an adequate basis for extending the IL to include recursive procedures but this has not currently been done.

5.5 Blocks

We define a local variable block $var\ k, k' : T \bullet B$ as in [8] to mean that

1. for all possible initial k's the precondition of B is true and
2. there exists k, k' such that B is true.

As an example, the *lookup* operation (Figure 1) introduces a new block with the IL *var*.

5.6 Procedure call

Consider the procedure `P` represented in Sum as follows.

$$
\begin{array}{|l}
\textit{op } P \\
\hline
x1, x1' : T11;\ ...;\ xm, xm' : T1m;\ v1, v1' : T21;\ ...;\ vn, vn' : T2n \\
\hline
... \\
\hline
\end{array}
$$

A call is represented as follows

$$\textbf{call}\ (P, (x1\ \ => E1, ..., xm => Em), (v1 => w1, ..., vn => wn))$$

where $E1, ..., Em$ are expressions to be bound to input parameters $x1, .., xm$ and $w1, ..., wn$ are variables to be bound to output parameters $v1, .., vn$.

The semantics of the call basically utilises renaming on the schema P. However, some elaborations on that basic idea are required for a number of reasons. Firstly, schema renaming does not permit the substitution of expressions for inputs. Secondly, we wish to correctly match the 'copy in copy out' semantics for variables. To deal with these issues, the semantics of call (shown below) introduces new local variables corresponding to the input and output parameters and P is 'called' with those new local variables substituted for P's inputs and outputs. Before the call, the local variables corresponding to inputs are initialised to the expressions $E1, ..., En$. After the call, the values of the local variables corresponding to outputs are copied out to the variables $w1, ..., wn$.

Another matter dealt with in the semantics of call is that the operation P does not necessarily completely describe the effect of P on the state in the *calling* environment. The semantics of call includes a predicate saying that only variables in the signature of P and the variables $w1, ..., wn$ can be changed by the call.

Finally, to ensure that aliasing problems do not arise, it is required that $w1, ..., wn$ are distinct and do not occur in the signature of P. This constraint ensures that the absence of aliasing is checked during refinement.

The semantics of the call is defined as follows in each situation s:

$$
\begin{array}{l}
(\textit{var}\ z11, z11' : T11;\ ...;\ z1m, z1m' : T1m; \\
\quad z21, z21' : T21;\ ...,;\ z2n, z2n' : T2n \bullet \\
\quad (z11, ..., z1m := E1, ..., Em\ ;; \\
\quad P\{z11/x1, ..., z1m/xm, z21'/v1', ..., z2n'/vn'\}\ ;; \\
\quad w1, ..., wn := z21, ..., z2n)) \bullet s \wedge \\
\textit{changes_only}(\textit{sig}(P) \cup \{w1, ..., wn\}, \textit{sig}(s))
\end{array}
$$

5.7 Translating IL statements

Translation of *while* and $\textit{if}_c$ constructs is straightforward. Here we briefly discuss translation of assignments, blocks and procedure calls.

5.7.1 Multiple assignment

To achieve the affect of the Sum multiple assignment in Ada, it is expanded in translation to a sequence of assignments to temporary variables. For example, the Sum multiple assignment in the *update* operation in Figure 1 has been expanded in the Ada procedure *update* in Figure 2.

Temporary variables are introduced in the translation in a local **declare** block. Their types are inferred from the declared types of the variables. As seen in the example, array updates are translated using temporary scalar variables of the array index and range types. This relatively simple treatment of translation of array updates is possible because the IL does not support array slices nor may repeated occurrences of the same array appear on the left-hand-side.

The current approach to the translation of multiple assignment is quite conservative. Subsequent analyses, such as data flow analysis, of the expanded assignments would allow further optimisation.

5.7.2 Blocks

Sum *var* blocks are translated to Ada **declare** blocks. For example, the *var* block in the *lookup* operation in Figure 1 is translated to an equivalent Ada **declare** block in Figure 2. The translator declares the variable *ind* representing the Sum modeling of a variable by *ind* and *ind'*.

5.7.3 Operation call

IL operation calls are translated to Ada procedure calls The operation parameters are translated to Ada named parameter associations. If the input expression is not a simple variable reference, a temporary variable must be used as the actual parameter to the **in out** formal parameter. For example, see the translations of Sum calls in Figure 3 to corresponding Ada calls in Figure 4.

6 Discussion

6.1 Modular development

In section 3 above we noted that module state is translated to public variables of the corresponding package. It might seem that translation to private variables would be more appropriate, in order to ensure that local state is not interfered with other than through the operations of the module, each of which assumes the state invariant and is required to maintain it.

However a model-oriented approach to specification (as used here and as adopted in e.g. Z and VDM) precludes complete hiding of the state, at least at early stages of specification and development, since the semantics of operations is specified in terms of the module state. Using modules must have access to that state for specification and reasoning purposes.

Note however that when a using module includes imported state into its own state it also imports an obligation to maintain the *state invariant* of the imported module. Formal development ensures that this obligation is satisfied, thus protecting the state of the imported module. Thus the formal development process itself ensures that the local state is protected. The simplest way for a developer to maintain the state invariant will be to access the imported module state only through its exported operations.

For example, module *symtabuser* (Figure 3) includes the state of *sti* into its own state allowing local operations to be developed referring to the state of *sti* directly. Through the process of development, the interface to *sti* is restricted to the operations *sti.lookup* and *sti.update*. At this point the inclusion of *sti.state* may be removed as the state invariant of the imported module is guaranteed by the operations.

6.2 Input and output variables

As mentioned in 3 above, we have avoided the use of '?' and '!' variables in the Sum IL.

Output ('!') variables essentially only exist in the 'final' state of a complete operation. In general Sum and Z specifications, the value of an output parameter is instantaneously chosen to make the predicate of the operation true. As there is no model of the output parameter in intermediate states, in pure IL an output variable can only usefully be set in a 'final assignment'. This severely restricts their usefulness in practical program development. Note that in contrast an Ada **out** parameter is essentially an uninitialized **in out** parameter. A programmer may use intermediate assignments to determine the final value of the **out** parameter.

The approach currently taken by Cogito is to simply require that variables of the form $x!$ in higher level Sum specifications be transformed to pairs x, x' before detailed algorithm refinement is undertaken. (This transformation is justified relative to the constraint that $x! = x'$). Such a pair x,x' appearing in an operation is translated to an Ada **in out** parameter. A similar approach is adopted for ? variables.

7 Conclusions and further work

We have shown how a simple but useful programming language can be semantically embedded in an extended Z-like language, and translated to Ada. Our approach provides a strong integration of the development language semantics with the specification language semantics and gives a foundation for refining from higher-level specifications to implementations.

In further work we would like to refine and extend the approach outlined here. We wish to investigate the introduction of private state into translated Ada packages. As explained above, the current approach is sound in an environment in which all users of a package are formally developed. However, in

an environment where some users are untrusted, interference with the package invariant is possible. Two points need to be noted. First, the state of an imported module must still be available at the semantic and theory level, in order to permit reasoning about called procedures. Restrictions on use affect only the permitted form of specification in the importing module. Second, we do not wish to introduce restrictions on usage of imported state too early in the development cycle. Doing so would rule out the ability to use the imported state in order to concisely specify an effect which could later be refined to procedure calls.

We also wish to the extend the current approach to procedure development and translation to allow development to procedures with **in** and **out** parameters. Our current approach of tranforming input-output variables to primed, unprimed pairs is sound but the ability to target more general procedure headers would be useful where the target interface is already defined for some reason.

References

[1] Holger Brecht, Anthony Bloesch, Ray Nickson, and Mark Utting. The Ergo 4.1 User Manual. Technical Report 96-31, Software Verification Research Centre, The University of Queensland, 4072, Australia, November 1996.

[2] Nicholas Hamilton, Dan Hazel, Peter Kearney, Owen Traynor and Luke Wildman. A Complete Formal Development using Cogito. In Computer Science '98: Proceedings of the 21st Australasian Computer Science Conference, February 1998. Springer Verlag.

[3] C.A.R. Hoare, 'Programs are predicates' in C.A.R. Hoare and J.C. Shepardson (eds.) *Mathematical Logic and Programming Languages*, Prentice-Hall, pp. 141-54 (1985).

[4] Eric Hehner, *A practical theory of programming* New York : Springer-Verlag, 1993.

[5] D.T. Jordan, C.J. Locke, J.A. McDermid, C.E. Parker, B.A. Sharp, and I. Toyn. Literate Formal Development of Ada from Z for Safety Critical Applications. In *SAFECOMP*, 1994.

[6] S. King, Z and the Refinement Calculus, in *VDM and Z – Formal Methods in Software Development*, Springer-Verlag, LNCS 428, 1990, 164-188.

[7] Thanh Tung Nguyen, A relational model of demonic nondeterministic programs, Foundations of Computer Science, Vol. 2, No. 2 (1991) 101-131.

[8] Sekerinski, E., A Calculus for Predicative Programming, in *Mathematics of Program Construction* , LNCS 669.

[9] O. Traynor, D. Hazel, P. Kearney, A. Martin, R. Nickson, and L. Wildman, The Cogito Development System, in *Algebraic Methodology and Software Technology*, LNCS 1349, 1997.

[10] O. Traynor, E. Kazmierczak, P. Kearney, L. Wang, and E. Karlsen, Extending Z with Modules, in *Australian Computer Science Communications*, Vol 17, No. 1, 1995.

[11] A. Bloesch, E. Kazmierczak, P. Kearney, J. Staples, O. Traynor, and M. Utting, A formal reasoning environment for Sum – A Z based specification language, in Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 1996.