SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 97-46

A Complete Formal Development using Cogito

Nicholas Hamilton   Dan Hazel
Peter Kearney   Owen Traynor
Luke Wildman

December 1997

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

To appear in *Proceedings of the* 21st *Australasian Computer Science Conference*, Perth, 1998, Springer-Verlag, Transactions of the CSA.

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`

A Complete Formal Development using Cogito

Nicholas Hamilton Dan Hazel Peter Kearney
Owen Traynor Luke Wildman

Abstract

This paper gives an overview of a case study in fully formal development from high level specification to executable code using the Cogito formal development system. The case study demonstrates a significant advance in integrated support for fully formal development.

Keywords Formal Methods, Formal Specification, Algorithm Refinement, Data Refinement.

1 Introduction

Formal, mathematically based software development methods provide a basis for achieving very high levels of assurance of the correctness of software. However, while formal techniques have been used quite a lot for formal specification and to a lesser extent for formal code verification, there are relatively few demonstrations of fully formal machine checked developments from high level specification to executable code.

This paper presents such a development for a small symbol table package, using the Cogito formal development system [BKKT95]. Cogito's formal development language is based on the well known specification language Z [Wor92] with extensions for modular specification and development to code (see Section 2.1 for more details).

The development presented here comprises formal high-level specification, formal validation of the specification, data refinement of high level data structures to low level data structures, algorithm refinement to imperative statements and translation to Ada code.[1] All steps in the development are machine assisted and verification requirements have been machine checked using the Ergo theorem prover [UW95, UNT96].

This case study demonstrates a significant advance in support for fully formal development from Z to executable code. More detailed comparisons with related work are given in Section 5.

[1] Ada'83 is targeted, but the subset of Ada which the system produces is common to Ada'83 and Ada'95.

2 Background: The Cogito System

2.1 The Cogito Methodology

The Cogito system consists of methods and tools for the formal development of sequential systems from high level specification to executable code.

Cogito's formal development language, called Sum, is based on the well known specification language Z [Wor92], with soundness assured by a mathematical semantics in set theory. Sum extends Z with facilities for: encapsulation, parameterisation and instantiation, partitioning specifications, defining state machines, combining state machines, stating preconditions explicitly, and expressing implementations. These extensions enable Sum to support the entire development process, whereas Z has focussed on the specification phase of development.

Development begins with a high level specification expressed as one or more Sum modules. Specifications are formally *validated* to show the consistency of the specification by proving certain standard proof obligations. Expected properties of the specification may also be proved as a form of validation. *Animation* can also be used to 'test' the results of an abstract specification.

Development proceeds through one or more *data refinement* steps, in which abstract data structures used in the initial specification are converted to data structures suitable for implementation.

Next, *algorithm refinement* steps are undertaken to convert specifications expressed as pre and post conditions into procedural programs. This process proceeds step-wise, finally producing an implementation expressed in a subset of Sum called the *intermediate programming language* (see Section 2.2).

Finally, the resulting intermediate language program can be automatically translated to Ada.

The case study described here illustrates all these development steps, except for the use of animation.[2]

2.2 The Sum Intermediate Language

Fully formal development in Cogito is targeted to an intermediate programming language (IL), defined as a subset of Sum, which can be translated to a variety of imperative languages. Currently, Cogito supports automatic translation to Ada.

Space limitations preclude a full discussion of IL types here. We give a brief review of types actually used in the case study presented in this paper.

The computational types *integer*, *natural*, and *natural_1* are defined as subtypes of the theoretical (infinite) *int* type and are represented by a specific (machine dependent) range of integers.

Machine oriented arithmetic operators are defined so that results are only guaranteed to be defined when results are within range. (Thus they are partial

[2]The animator is a relatively recent addition to the toolkit. The current case study has been animated using the tool. For more details on the animator see [HST97].

operations on *int*, the theoretical integer type). These operators are generally denoted using a c subscript. For example computational addition is denoted $+_c$.

Similarly, the computational boolean type and operators are available.

Constrained array types are represented in the IL by total functions from finite, discrete types to IL types: the type $array(I, R)$ of arrays with index type I and range R is defined to be $I \to R$. Array update is represented by a restricted version of function override which does not extend its domain (since this is not possible for arrays). The expression $upd(a, i, v)$ denotes the function (array) a updated at index i to the value v.

Procedural constructs are modelled in the intermediate language as predicates relating before (unprimed) and after (primed) states. For example, the IL has a multiple assignment operator :=. The meaning of the multiple assignment

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

where $E1, \ldots En$ are independent of any primed variables, is that

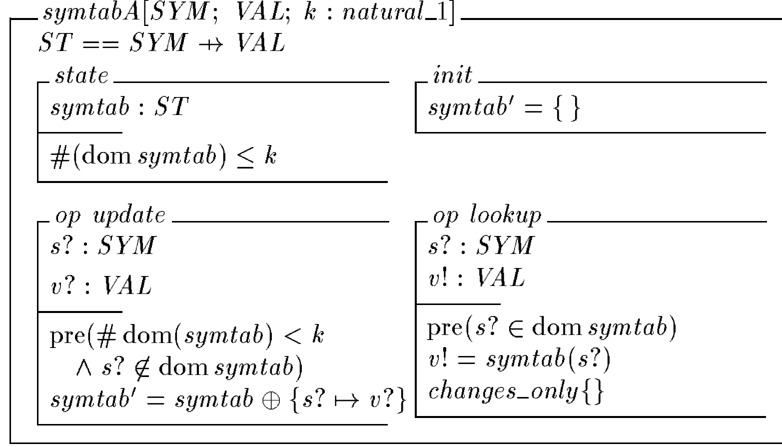
1. $x1' = E1 \wedge \ldots xn' = En$
2. For all variable pairs Y, Y' in the current signature such that Y does not appear on the left of the assignment, $Y' = Y$.

The sequencing operator ;; is defined using quantification over intermediate states. The *while* construct is defined by a fixed point construction over predicates in the Sum semantics.

As in various refinement calculi (e.g., [Mor95]), Sum intermediate language constructs can be intermixed with more general 'non-IL' Sum specification constructs.

3 The Case Study

We begin with an abstract specification *SymtabA* which describes a simple symbol table as a map (ST) from symbols (SYM) to values (VAL). The state invariant is that the number of maps should not exceed the predetermined value k. The table is initialised to the empty set in the *init* schema. There are then two operation schemas: *update* which given the appropriate precondition adds to the symbol table; and *lookup* which given an appropriate symbol finds the corresponding value. In Cogito, operation schemas (denoted by *op schemaname*) implicitly include the *state* schema and a primed copy of the *state* schema. The notation *changes_only* A asserts that for every state variable x which is not in the set of state variables A, $x = x'$.



3.1 Validation Proof Obligations

A schema is modelled in Ergo as a pair $[\, Sig \mid A \,]$ of a *signature* and a *predicate*. Predicates are modelled as mappings from *situations* to *boolean*. A *situation* maps identifiers to their values and types. Thus for each situation (dependent on the context in which the schema is used) the predicate part yields a truth value based on the current situation. Based on this modeling, the schema operators $\wedge_s$, $\vee_s$, etc. are supported, as well as the operator pre_s which calculates the precondition of a schema. In addition, the *schema_derives* ($\Rrightarrow_s$) operator can be used to specify proof requirements in the schema language. $S1 \Rrightarrow_s S2$ is true if and only if $S1$ and $S2$ are type compatible and in all situations which satisfy the predicate of $S1$ the predicate of $S2$ is also satisfied. See [BKK+96] for more details of the modeling of Sum in Ergo.

As part of the Cogito methodology it is usual, as a form of validation, to discharge two types of proof obligation for the initialisation and operation schemas. These are generated automatically by Ergo when loading theories generated from Sum theories, and then discharged using the Ergo interactive theorem prover.

These proof obligations are

- The stated precondition *Pre* of schema *Op* implies the calculated precondition. This is expressed in Ergo as: $(Pre \wedge_s State) \Rrightarrow_s \text{pre}_s(Op)$
- The schema is satisfiable, i.e., there exist values for all of the variables that satisfy the typing conditions and predicate part of the schema.

For instance, in the *SymtabA* specification the precondition proof obligation for the *lookup* schema amounts to showing that for all values of *symtab* and $s?$, if $s?$ is in the domain of *symtab* (as required by the explicit precondition), then there exist values of the correct type of $symtab'$ and $v!$ such that $s?$ is in the domain of *symtab*, $v! = symtab(s?)$ and the state variables are unchanged (i.e., $symtab = symtab'$).

Each of the specifications *SymtabA*, *SymtabC* and *SymtabD* were validated in this manner using Ergo. Proof lengths range from 1 to 250 proof steps entered by the user. The actual number of proof steps was often several thousand, but this number is reduced using automated tactics.

3.2 Data Refinement

The next development step is to data refine the specification to *SymtabC* in which the abstract function *ST* is replaced by the more array like constructs *keys* and *vals* to record the symbol table.

$symtabC[SYM;\ VAL;\ k : natural_1]$

state

$$keys : 1 \mathrel{..} k \rightarrow SYM$$
$$vals : 1 \mathrel{..} k \rightarrow VAL$$
$$high : 0 \mathrel{..} k$$

$$\forall\, i, j : 1 \mathrel{..} high \bullet i \neq j \Rightarrow keys(i) \neq keys(j)$$

init

$$high' = 0$$

op update

$$s? : SYM$$
$$v? : VAL$$

$$\mathrm{pre}(high < k \wedge \forall\, i : 1 \mathrel{..} high \bullet s? \neq keys(i))$$
$$high' = high + 1$$
$$keys' = keys \oplus \{high + 1 \mapsto s?\}$$
$$vals' = vals \oplus \{high + 1 \mapsto v?\}$$

op lookup

$$s? : SYM$$
$$v! : VAL$$

$$\mathrm{pre}(\exists\, i : 1 \mathrel{..} high \bullet keys(i) = s?)$$
$$\exists\, i : 1 \mathrel{..} high \bullet (keys(i) = s? \wedge v! = vals(i))$$
$$changes_only\{\}$$

In data refinement a *representation relation* defines the formal relationship between the abstract and concrete states. For instance, in the data refinement between *SymtabA* and *SymtabC* the representation relation is expressed in the *SymtabAC* module.

$symtabAC[SYM;\ VAL;\ k : natural_1]$

import symtabA(SYM, VAL, k) as Abs
import symtabC(SYM, VAL, k) as Conc

symtabReln

$$Abs.state$$
$$Conc.state$$

$$\mathrm{dom}\, Abs.symtab = \{ks : SYM \mid \exists\, i : 1 \mathrel{..} Conc.high \bullet Conc.keys(i) = ks\}$$
$$\forall\, i : 1 \mathrel{..} Conc.high \bullet Abs.symtab(Conc.keys(i)) = Conc.vals(i)$$

updateReln
$Abs.update$
$Conc.update$

$Abs.s? = Conc.s?$
$Abs.v? = Conc.v?$

lookupReln
$Abs.lookup$
$Conc.lookup$

$Abs.s? = Conc.s?$
$Abs.v! = Conc.v!$

(– – *data_refine*(*symtabAC_Abs*, *symtabAC_Conc*,
[*symtabAC_Abs.init* ∼ *symtabAC_Conc.init*,
symtabAC_Abs.update ∼ *symtabAC_Conc.update*,
symtabAC_Abs.lookup ∼ *symtabAC_Conc.lookup*],
symtabReln, [*empty_schema*, *updateReln*, *lookupReln*]).
– –)

In this module *SymtabA* and *SymtabC* are imported as *Abs* and *Conc* respectively. The relationship between the state variables in *Abs* and *Conc* is then defined in the schema *symtabReln*. The *data_refine* directive specifies the data refinement relationship beteen *SymtabA* and *SymtabC* and is used by the translator to create proof obligations.

To verify that a specification refines another, a number of types of proof obligation are generated using the representation relation module. Formally, we say operation OpA is data refined to operation OpC where OpC and OpA are related by the relation R, when the following conditions hold:

- Precondition Weakened: $\mathrm{pre}_s(OpA) \wedge_s R \Rrightarrow_s \mathrm{pre}_s(OpC)$,
- Postcondition Strengthened:
  $\mathrm{pre}_s(OpA) \wedge_s R \wedge_s OpC \Rrightarrow_s \exists_s(A.state', OpA \wedge_s R')$,
- Initialisation: $C.init \Rrightarrow_s \exists_s(A.state', A.init \wedge_s R')$.

Automated tactics in Ergo discharge substantial parts of each of the obligations. The lengths of the proofs for the *SymtabA* to *SymtabC* were typically less than 100 lines, though the precondition proof for the *lookup* schema proved relatively difficult and took 500 lines.

SymtabC is then data refined to *SymtabD* in which the abstract types such as $1\,..\,k$ are replaced by computational types $1\,..\,_c k$ in preparation for algorithm refinement. The refinement relation *SymtabCD* (not shown here) is expressed as a specification similar to *SymtabAC*.

symtabD[*SYM*; *VAL*; *k* : *natural_1*]

state
$keys : array(1\,..\,_c k, SYM)$
$vals : array(1\,..\,_c k, VAL)$
$high : 0\,..\,_c k$

$\forall\, i, j : 1\,..\,_c high \bullet$
$\quad i \neq_c j \Rightarrow keys(i) \neq_c keys(j)$

init
$high' = 0$

op update

$$\begin{array}{l} s, s' : SYM \\ v, v' : VAL \\ \hline \mathrm{pre}(high < k \land \\ \quad \forall\, i : 1 \,..\, high \bullet s \neq keys(i)) \\ high' = high + 1 \\ keys' = keys \oplus \{high + 1 \mapsto s\} \\ vals' = vals \oplus \{high + 1 \mapsto v\} \\ changes_only\{keys, vals, high\} \end{array}$$

op lookup

$$\begin{array}{l} s, s' : SYM \\ v, v' : VAL \\ \hline \mathrm{pre}(\exists\, i : 1 \,..\, high \bullet keys(i) = s) \\ \exists\, i : 1 \,..\, high \bullet \\ \quad (keys(i) = s \land v' = vals(i)) \\ changes_only\{v\} \end{array}$$

3.3 Algorithm Refinement

The final refinement of *SymtabD* to IL is given in Module *SymtabDaref*.

symtabDaref$[SYM;\ VAL;\ k : natural_1]$

state

$$\begin{array}{l} keys : array(1 \,..{}_c\, k, SYM) \\ vals : array(1 \,..{}_c\, k, VAL) \\ high : 0 \,..{}_c\, k \\ \hline \forall\, i : 1 \,..{}_c\, high;\ j : 1 \,..{}_c\, high \bullet \\ \quad i \neq_c j \Rightarrow keys(i) \neq_c keys(j) \end{array}$$

init

$$\begin{array}{l} high' : 0 \,..{}_c\, k \\ keys' : array(1 \,..{}_c\, k, SYM) \\ vals' : array(1 \,..{}_c\, k, VAL) \\ \hline assign \quad high := 0 \end{array}$$

update

$$\begin{array}{l} high, high' : 0 \,..{}_c\, k \\ keys, keys' : array(1 \,..{}_c\, k, SYM) \\ s, s' : SYM \\ v, v' : VAL \\ vals, vals' : array(1 \,..{}_c\, k, VAL) \\ \hline assign \quad high, keys, vals := \\ \qquad high +_c 1, \\ \qquad upd(keys, high +_c 1, s), \\ \qquad upd(vals, high +_c 1, v) \end{array}$$

lookup

$$\begin{array}{l} high, high' : 0 \,..{}_c\, k \\ keys, keys' : array(1 \,..{}_c\, k, SYM) \\ s, s' : SYM \\ v, v' : VAL \\ vals, vals' : array(1 \,..{}_c\, k, VAL) \\ \hline var\ ind : 1 \,..{}_c\, k;\ ind' : 1 \,..{}_c\, k \bullet \\ \quad assign \quad ind := 1 \ ;; \\ \quad\ while \quad keys(ind) \neq_c s \quad do \\ \qquad assign \quad ind := ind +_c 1 \quad done\ ;; \\ \quad assign \quad v := vals(ind) \end{array}$$

In algorithm refinement, the predicate in the body of an operation schema is refined to a predicate defined by IL constructs. Refinement of predicates is in the context of the signature of the operation being refined. The term $A \sqsubseteq_{Sig} B$ means that the predicate A is refined to the predicate B in the context of the signature Sig.

The following is the definition of refinement of the predicate A to the predicate B in the context of signature Sig.

axiom gen_ref_intro ===

$$
\begin{aligned}
&\text{pre}_s([\, Sig \mid A \,]) \Rrightarrow_s \text{pre}_s([\, Sig \mid B \,]) \wedge_s \\
&\text{pre}_s([\, Sig \mid A \,]) \wedge_s [\, Sig \mid B \,] \Rrightarrow_s [\, Sig \mid A \,] \\
&\qquad \Rightarrow A \sqsubseteq_{Sig} B
\end{aligned}
$$

We discuss here the algorithm refinement of the *lookup* schema. This involves the construction of a *while* loop with the appropriate initialisation and loop invariant.

```
                 pre( exists i : 1 .. high @ keys(i) = s )
                 and exists i : 1 .. high  @ keys(i) = s and v' = vals(s)
                 and changes_only{v}
               ____________________________________________________
                                     ⊑
      var ind, ind' : 1 ..c k @
                 pre( exists i : 1 .. high @ keys(i) = s )
                 and exists i : 1 .. high  @ keys(i) = s and v' = vals(s)
                 and changes_only{v}
               ____________________________________________________
                                     ⊑
    var ind, ind' : 1 ..c k @
       keys'(ind') = s' and ind' < high ' and changes_only{ind}  ;;  assign v := vals(ind)
       ________________________________________________________      _________________
                          ⊑
var ind, ind' : 1 ..c k @
                      exists y : ind .. high @ keys(y) = s
                      and exists y : ind' .. high' @ keys'(y) = s'
   assign ind := 1 ;; and changes_only{ind} and notc(keys'(s') /=c s' ;; assign v := vals(ind)
   _______________    ________________________________________        __________________
                                     ⊑

                      while keys(ind) /=c s do
                         exists y : ind .. high @ keys(y) = s
var ind, ind' : 1 ..c k @  and exists y : ind' .. high' @ keys'(y) = s'
                         and changes_only{ind}
                         and high' - ind' < high - ind
   assign ind := 1 ;;    and 0 <= high' - ind'  done      ;;  assign v := vals(ind)
   _______________    _________________________________       __________________
                                  ⊑

var ind, ind' : 1 ..c k @
                          while keys(ind) /=c s do
      assign ind := 1 ;;    assign ind := ind +c 1  done  ;;  assign v := vals(ind)
      _______________     ______________________________      __________________
```

Figure 1: Algorithm Refinement of the Lookup Schema Predicate

In Figure 1 the steps of the algorithm refinement of the *lookup* schema are shown (note that Sum's machine-readable syntax is used). The top of the figure begins with the body of the *SymtabD lookup* schema.

The first step is to introduce the local variables *ind* and *ind'*. These will be used as our counter in the while loop to be introduced. This step uses the

following rule that can be derived from *gen_ref_intro* in the specific instance of variable introduction :
axiom loc_var_intro $===$ $disjoint_sig(Sig, D) \Rightarrow F \sqsubseteq_{Sig} var(D, F)$

To use this rule and introduce a new variable we need to show that the variable D is disjoint from the signature Sig (expressed by the proposition $disjoint_sig(Sig, D)$).

The body within the scope of the declarations of ind and ind' is then refined to the sequential composition $S2$;; $assign\ v := vals(ind)$ where $S2 = keys'(ind')$ and $ind' < high'$ and $changes_only\{ind\}$.

The following rule allows such an introduction of program sequencing, refining the predicate A to B;; C relative to the signature Sig.
axiom seq_intro $===$

$$\begin{array}{l} \mathrm{pre}_s([\,Sig \mid A\,]) \Rrightarrow_s \mathrm{pre}_s([\,Sig \mid B\,]) \wedge_s \\ \mathrm{pre}_s([\,Sig \mid A\,]) \wedge_s [\,Sig \mid B\,] \Rrightarrow_s schema_prime(\mathrm{pre}_s([\,Sig \mid C\,])) \wedge_s \\ \mathrm{pre}_s([\,Sig \mid A\,]) \wedge_s [\,Sig \mid B\,] \,;;\, [\,Sig \mid C\,] \Rrightarrow_s [\,Sig \mid A\,] \\ \quad \Rightarrow A \sqsubseteq_{Sig} B \,;;\, C \end{array}$$

The next couple of steps are to refine $S2$ to an initialised while loop. This involves establishing an initialisation: $assign\ \ ind := 1$, a loop invariant: $\exists\, i : ind..high \bullet keys(i) = s$, a guard for the loop : $keys(ind) \neq s$, and a bound decreasing function: $high - ind$, and applying a while introduction rule to obtain the final result.

The proof obligations generated for the above typically took less than 100 lines each. There is plenty of room for improvement as further tactics for reasoning about intermediate language types are developed.

4 Translation to Ada

The translation from the intermediate programming language module *symtabDaref* to a safe subset of Ada is achieved using the Sum translator.

The *symtabDaref* module becomes a generic package with both a specification and a package body. The translator generates generic packages for both generic and non-generic modules. Generic instantiation in Ada can then be used to support *import-as* in Sum. Additional non-generic packages are generated for non-generic modules to support direct *import* in Sum. The Ada package specification contains declarations visible to any importing package.

```
generic
    type SYM is private;
    type VAL is private;
    k: positive;
package gen_symtabDaref 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_symtabDaref;
```

The state invariant of the IL module, now preserved by each of the operations, is inserted as a comment for future reference.

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

In the other operations, variables other than the state variables (primed and unprimed) become “in out” procedure parameters. As Sum input and output variables have been data-refined to primed and unprimed counterparts these also become “in out” parameters.

```
package body gen_symtabDaref 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 ;
    begin
        init;
    end;
end gen_symtabDaref;
```

In translating the whole array assignments of the IL such as those occurring in the *update* operation, one could either invent a function to implement the *upd* function directly or perform the indexed array assignments in line. As the use of a function to do array assignment may introduce inefficient memory usage we have chosen the latter. This is possible because the IL does not support array slices nor may repeated occurrences of the same array appear on the left-hand-side.

In translating the array update when used in a multiple assignment, temporary scalar variables *keys_temp* and *vals_temp* of the array index and range types must be introduced. The current approach to the translation of multiple assignment is quite conservative. Subsequent analyses, such as data flow analysis, of the expanded assignments could allow further optimisation.

In the *lookup* operation, the translator declares the variable *ind* representing the Sum modeling of a variable by *ind* and *ind'*. The translations of the IL *while* loop, and *sequential composition* are straightforward.

5 Discussion and Conclusions

The only other work on tool-supported formal development from Z of which we are aware is the University of York work on the CADiZ Z proof system [TM95] and the Zeta [JLM+94] refinement method. The Cogito support described here provides a better semantic integration of the implementation-level constructs with the specification level semantics than that provided by Zeta, since in Cogito intermediate programming language constructs are defined as part of the Sum specification language whereas Zeta utilises a mixture of Ada and Z for formal development.

Furthermore, Zeta does not utilise any modularisation constructs at the specification level so that the production of Ada packages is ad-hoc when compared to Cogito, where an implementation level module structure can be defined as part of the formal development and used as a basis for a well defined translation from Sum modules to Ada packages.

The B specification language [Abr96] is not Z or Z-like, but the B method supports model-oriented specification and development similar to that supported in Cogito. The Cogito system and the B-system are broadly comparable, both supporting modular specification and refinement. In some respects the Cogito module structuring facilities are more flexible than those in B and, since Cogito extends Z, a broader range of specification structuring mechanisms

are available (such as the schema calculus). For a more detailed comparison see [Wat97].

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[BKK+96] A. Bloesch, E. Kazmierczak, P. Kearney, O. Traynor J. Staples, and M. Utting. A formal reasoning environment for Sum - a Z based specification language. *Australian Computer Science Communications*, 18(1):149–158, 1996.

[BKKT95] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: A Methodology and System for Formal Development. *International Journal of Software Engineering*, 4(3), 1995.

[HST97] Dan Hazel, Paul Strooper, and Owen Traynor. Possum: An animator for the sum specification language. *Software Verification Research Centre Technical Report, 97 – 10, The University of Queensland*, 1997.

[JLM+94] 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.

[Mor95] C. Morgan. *Programming from Specifications.* Prentice Hall, 2nd edition, 1995.

[TM95] I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software—Practice & Experience*, 25(3):305–330, March 1995.

[UNT96] M. Utting, R. Nickson, and O. Traynor. Theory structuring in Ergo 4.1. In *Computing: The Australasian Theory Symposium*, Melbourne, Australia, January 1996.

[UW95] M. Utting and K. Whitwell. The Ergo interactive theorem prover v4.0. *Software Verification Research Centre Technical Report, 94 – 14, The University of Queensland*, 1995.

[Wat97] G. Watson. A comparison of modularity in B and Cogito. In S. Reeves L. Groves, editor, *Formal Methods Pacific*, pages 263–286. Springer-Verlag, 1997.

[Wor92] J.B. Wordsworth. *Software Development using Z.* Addison-Wesley, 1992.