ProofPower

Compliance Tool—Language Description

PPTex-2.9.1w2.rda.110727

Information on the current status of ProofPower is available on the World-Wide Web, at URL:

`http://www.lemma-one.demon.co.uk/ProofPower/index.html`

This document is published by:

Lemma 1 Ltd.
2nd Floor
31A Chain Street
Reading
Berkshire
UK
RG1 2HX
e-mail: `pp@lemma-one.com`

PPTex-2.9.1w2.rda.110727 13:54:16 27/07/2011

CONTENTS

List of Tables

ABOUT THIS PUBLICATION

0.1 Purpose

This document describes the syntax and semantics of the Compliance Notation as supported by the Compliance Tool supplied as an extension to ProofPower.

0.2 Readership

This document provides reference material intended to be read by users of the Compliance Tool.

0.3 Related Publications

A bibliography is given on page 97 of this document.

- An overview of the of the Compliance notation can be found in the DRA document:
  A commentary on the Specification of the Compliance Notation for SPARK and Z [4].
- The formal specification of the Compliance notation may be found in the DRA document:
  Specification of the Compliance Notation for SPARK and Z (3 volumes) [3].
- The SPARK subset of Ada is described in the book:
  High Integrity Ada — The Spark Approach [1].
- The use of the Compliance Tool is described in:
  Compliance Tool — User Guide [8].
- A description of ProofPower may be found in:
  ProofPower Software and Services *[6]*,
  which also contains a list of other ProofPower documentation.
- The Ada language supported by the Compliance Tool is a subset of Ada '83. Ada '83 is defined in the following book (referred to as ALRM in this document):
  The Annotated Ada Reference Manual[2]

0.4 Prerequisites

It is assumed that the reader is familiar with Ada and Z.

0.5 Acknowledgements

Lemma 1 Ltd. gratefully acknowledges its debt to the many researchers (both academic and industrial) who have provided intellectual capital on which ICL and then Lemma 1 have drawn in the development of ProofPower and the Compliance Tool.

Program Validation Limited (PVL) designed the SPARK subset of Ada, on which the Compliance Notation subset of Ada was originally based. We are indebted to the Defence Research Agency, Malvern (DRA), now called QinetiQ, who designed the Compliance Notation and sponsored the development of SPARK and of the Compliance Tool.

INTRODUCTION

The Compliance Notation allows Ada programs to be presented in a literate programming style in which the order of presentation of program fragments is chosen by the writer rather than fixed by the Ada syntax rules. A program presented in the Compliance Notation may be interspersed with formal specifications of the program's behaviour written in the specification language Z. If the program conforms to the rules given in this language description, a Z document can be produced automatically from its presentation using the Compliance Notation. This Z document contains Z paragraphs representing the Ada types, functions and constants defined in the Ada program, together with conjectures, known as verification conditions (VCs), whose proof constitutes a correctness proof for the program against its specification.

The level of mathematical rigour in a Compliance Notation script is under the user's control. At one extreme, no formal material at all need be included; at the other extreme, every subprogram can be formally specified and verified. Most practical uses of the Compliance Notation will lie between these extremes.

The notation includes a feature allowing the syntactic and semantic rules that support formal modelling to be bypassed completely so that Ada features, such as tasks, which are outside the scope of the formal treatment, can be used.

The Compliance Notation is supported by the Compliance Tool, an extension to ProofPower. In addition to syntax-checking, type-checking and document preparation functions, the Compliance Tool supports extraction of the Ada program from a Compliance Notation script and the generation of the Z document. All the facilities of ProofPower are available for proving VCs. These facilities are augmented with a range of theorems and proof procedures which are customised for VC proofs. The use of the Compliance Tool is described in *Compliance Tool — User Guide* [8].

This document describes the syntax and semantics of the Compliance Notation. The description of the syntax follows the structure of of the Ada Language Reference Manual, [2], referred to as ALRM in the rest of this document. Some features of Ada'95 as defined in [7] are supported in the Compliance Notation, notably use type clauses and child packages, but ALRM is the primary reference.

The description of the semantics of the Compliance Notaion is informal, but is based on the formal specifications of the Compliance Notation prepared by the Defence Research Agency [3].

COMPLIANCE NOTATION SYNTAX

2.1 Introduction

In this chapter, the syntax of the Compliance Notation is described. The Compliance Notation as described here uses the ProofPower-Z dialect of Z. The syntax of ProofPower-Z is described in ProofPower *Description Manual* [5].

Sections 2.2 to 2.14 give the bulk of the description following the structure of chapters 2 to 14 of ALRM.

BNF syntax in this document is given using the same notation as in ALRM, except that reserved words are shown in double quotation marks rather than in a bold font. Bold font is used for non-terminal symbols at their point of introduction. An index including these symbols is given at the end of this document.

Section 2.15 describes the *web clauses* which comprise the outermost level of the notation (the level at which Compliance Notation constructs are interleaved with narrative text in a script).

In the description of the syntax, the following terminology is used:

1. A construct *is not handled formally* if the Compliance Notation allows the syntax for the construct in parts of the script which do not have a formal specification, but does not support formal reasoning about the construct.
2. A construct is *not supported* if the Compliance Notation does not allow the syntax for the construct.

Unsupported constructs can be included in a script by means of the arbitrary Ada replacement facility (see 2.15).

2.2 Lexical Elements

2.2.1 Character Set

The character set for the Compliance Notation is the ISO seven-bit coded character set augmented with symbols required by the Z notation and certain special symbols described in section 2.2.2 below.

2.2.2 Lexical Elements, Separators, and Delimiters

The lexical elements of the Compliance Notation are the union of those for Ada as described in [2] and those for ProofPower-Z as described in ProofPower *Description Manual* [5] augmented with a small list of symbols with special significance.

Symbol	Name
Ⓢ*CN*	Compliance Notation start symbol
■	Compliance Notation end symbol
$\sqsubseteq$	Statement refinement symbol
$!\sqsubseteq$	Statement replacement symbol
$\equiv$	Declaration replacement symbol
$!\equiv$	Arbitrary Ada replacement symbol
$\langle$	Left k-slot symbol
$\rangle$	Right k-slot symbol
Δ	Specification statement symbol
Γ	Assertion statement symbol
Ξ	Function specification statement symbol

Table 2.1: Special Symbols

The special symbols are listed in table 2.1:

The Compliance Notation start and end symbols are used like the analogous symbols for ProofPower-Z to delimit the Compliance Notation parts of a document (see ProofPower *Description Manual* [5]). The other symbols are used within the Compliance Notation proper. Several of these symbols are also used in Z; the grammar of the Compliance Notation is such that the extent of a Z expression or Z predicate within a valid Compliance Notation construct can always be unambiguously determined.

2.2.3 Identifiers

The rules for identifiers are as in ALRM. The Compliance Notation reserved words are listed in section 2.2.9 below.

While Ada is not case-sensitive, Z is: when an Ada name is translated into a Z name it is translated into upper case.

2.2.4 Numeric Literals

Integer literals are as in ALRM.

Based real literals are not handled formally.

2.2.5 Character Literals

Character literals are as in ALRM.

2.2.6 String Literals

String literals are as in ALRM.

2.2.7 Comments

Ada comments are normally not passed on to the Ada program generated from a Compliance Notation script. For compatibility with the SPARK examiner, comments beginning with $--\#$ are optionally

passed on to the Ada program in those syntactic positions where the SPARK examiner allows or requires annotations, as described in [1]. Comments of this form are only allowed in these syntactic positions when the option is enabled. If this option is disabled, comments of this form are ignored.

Comments may also be passed in to the Ada program using the arbitrary Ada replacement facility.

Wherever a SPARK annotation is permitted, a k-slot (see section 2.3.9) may be used to defer provision of the actual text of the annotation. In the case of the assert annotation, which appears as part of a sequence of statements, use of a k-slot produces a construct which cannot be handled formally (see section 2.5.1). To handle a deferred annotation formally, a specification statement with an empty frame and *true* for the pre- and post-condition can be used; this specification statement can then be refined (see section 2.14) by the assert annotation, which is equivalent to a null statement for the purposes of VC generation.

2.2.8 Pragmas

Pragmas in the Compliance Notation are supported in the following places:

1. at any place where a declaration or a statement would be allowed;
2. in a declarative part;
3. immediately after a context clause;
4. where a compilation unit would be allowed.

Pragmas are just treated as data to be passed in to the Ada program generated from a Compliance Notation script and have no effect on the Z document.

2.2.9 Reserved Words

In addition to the reserved words of the ALRM, the Compliance Notation has the following keywords comprising a "$" character immediately followed by an identifier. The keywords are not case-sensitive.

$*auxiliary*
$*block*
$*by*
$*con*
$*deferred*
$*implement*
$*implicit*
$*nothing*
$*references*
$*till*
$*using*

2.2.10 Allowable Replacements of Characters

The replacement characters are not supported.

2.3 Declarations and Types

2.3.1 Declarations

Only the following forms of basic declaration are supported:

basic_declaration ::=
object_declaration
| *number_declaration*
| *type_declaration*
| *subtype_declaration*
| *subprogram_declaration*
| *package_declaration*
| *renaming_declaration*
| *deferred_constant_declaration*

A package declaration standing as a basic declaration inside a subprogram or another package declaration cannot be handled formally; package declarations can only be handled formally when they are used as library units (i.e., at the top level).

2.3.2 Objects and Named Numbers

object_declaration ::=
constant_declaration
| *variable_declaration*

constant_declaration ::=
identifier_list ":" "*constant*" *subtype_indication* ":=" *expression* ";"

variable_declaration ::=
identifier_list ":" *subtype_indication* [":=" *expression*] ";"

number_declaration ::=
identifier_list ":" "*constant*" ":=" *expression* ";"

identifier_list ::= *identifier* {"," *identifier*}

Object declarations involving anonymous array subtypes are not supported.

2.3.3 Types and Subtypes

2.3.3.1 Type Declarations

type_declaration ::= *full_type_declaration*
| *private_type_declaration*

full_type_declaration ::= "*type*" *identifier* [*discriminant_part*] "*is*" *type_definition* ";"

type_definition ::= *enumeration_type_definition*

| *integer_type_definition*
| *real_type_definition*
| *array_type_definition*
| *record_type_definition*

2.3.3.2 Subtype Declarations

subtype_declaration ::= "*subtype*" *identifier* "*is*" *subtype_indication* ";"

subtype_indication ::= *type_mark* [*constraint*]

type_mark ::= *name*

constraint ::= *range_constraint*
| *floating_point_constraint*
| *fixed_point_constraint*
| *index_constraint*
| *discriminant_constraint*

Note that a subtype indication need not include a constraint.

2.3.4 Derived Types

Derived types are not supported.

2.3.5 Scalar Types

range_constraint ::= "*range*" *range*

range ::= *range_attribute*
| *simple_expression* .. *simple_expression*

A range constraint must not denote a null range when used within a type definition. A VC may be generated asserting that the range is not null if it cannot be determined whether or not the range is null (e.g., if the range constraint appeals to compiler-dependent constants such as $INTEGER'LAST$).

2.3.5.1 Enumeration Types

enumeration_type_definition ::= "(" *identifier* {"," *identifier*} ")"

Character literals are not supported as enumeration literals.

2.3.5.2 Character Types

The predefined type *CHARACTER* may be used; however, user-defined character types are not supported (see the restriction in section 2.3.5.1 above).

2.3.5.3 Boolean Types

No restrictions apply to this section of ALRM.

2.3.5.4 Integer Types

integer_type_definition ::=
signed_integer_type_definition | *modular_type_definition*

signed_integer_type_definition ::= *range_constraint*

modular_type_definition ::= "*mod*" *expression*

No restrictions apply to this section of ALRM.

2.3.5.5 Operations of Discrete Types

All predefined attributes of discrete types can be handled formally. See section 3.1.8 below for more information.

2.3.5.6 Real Types

real_type_definition ::=
floating_point_constraint | *fixed_point_constraint*

2.3.5.7 Floating Point Types

floating_point_constraint ::=
floating_accuracy_definition [*range_constraint*]

floating_accuracy_definition ::=
"*digits*" *simple_expression*

2.3.5.8 Operations of Floating Point Types

All predefined attributes of floating point types can be handled formally. See section 3.1.8 below for more information.

2.3.5.9 Fixed Point Types

fixed_point_constraint ::=
fixed_accuracy_definition [*range_constraint*]

fixed_accuracy_definition ::=
"*delta*" *simple_expression*

2.3.5.10 Operations of Fixed Point Types

All predefined attributes of fixed point types can be handled formally. See section 3.1.8 below for more information.

2.3.6 Array Types

array_type_definition ::=
unconstrained_array_definition | *constrained_array_definition*

unconstrained_array_definition ::=
"*array*" "(" *index_subtype_definition* {"," *index_subtype_definition*} ")" "*of*"
type_mark

constrained_array_definition ::=
"*array*" *index_constraint* "*of*" *type_mark*

index_subtype_definition ::= *type_mark* "*range*" "<>"

index_constraint ::= "(" *discrete_range* {"," *discrete_range*} ")"

discrete_range ::= *subtype_indication* | *range*

2.3.6.1 Index Constraints and Discrete Ranges

A discrete range in an index constraint that is not given as a type mark and which does not contain a discriminant is treated as if transformed into one that is given as a type mark. This is done by treating the declaration as if it were preceded by a declaration of a type with an automatically generated type mark whose Z representation is equal to the required discrete range. The discrete range is then treated as if it had been written using the automatically generated type mark. See section 3.1.15.

An index constraint including a discriminant is only allowed in a record component declaration. The translation of these constraints is described in section 3.2.6.3 below.

2.3.6.2 Operations of Array Types

All predefined attributes of array types can be handled formally. See section 3.1.8 below for more information. In the case of the following attributes used with an argument, it must be possible to evaluate the argument statically — see section 2.4.9 for information on static expressions.

$A'FIRST(N)$
$A'LAST(N)$
$A'RANGE(N)$
$A'LENGTH(N)$

2.3.6.3 The Type String

The type *STRING* is treated precisely as if defined by the following unconstrained array type definition (see section 3.2.2.2):

type STRING is array (POSITIVE range <>) of CHARACTER

The catenation and ordering operators for string types are handled formally as is concatenation of a character and a string or a string and a character.

2.3.7 Record Types

record_type_definition ::=
"*record*"
component_list
"*end*" "*record*"

component_list ::=
component_declaration {*component_declaration*}

component_declaration ::=
identifier_list ":" *type_mark* ";"

2.3.7.1 Discriminants

discriminant_part ::= "(" *discriminant_specification* {";" *discriminant_specification*}

discriminant_specification ::= "*identifier_list* ":" *type_mark*

2.3.7.2 Discriminant Constraints

discriminant_constraint ::= *actual_parameter_part*

2.3.7.3 Variant Parts

Variant parts are not supported. The syntax for *choice* is given in section 2.4.3.

2.3.7.4 Operations of Record Types

The attributes for record types can be handled formally. See section 3.1.8 for more information.

2.3.8 Access Types

Access types are not supported.

2.3.9 Declarative Parts

declarative_part ::=
{*basic_declarative_item*} {*later_declarative_item*}

basic_declarative_item ::=
basic_declaration | *representation_clause* | *use_clause* | *k_slot* | *using_declaration*

later_declarative_item ::= *body*
| *subprogram_declaration* | *package_declaration* | *k_slot*

body ::= *proper_body* | *body_stub*

proper_body = *subprogram_body* | *package_body*

A k-slot (short for Knuth-slot) serves in several places in the syntax and denotes a placeholder for an Ada construct. The Ada constructs which can be deferred in this way are a compilation, a basic declarative item, a statement, a visible part or a private part.

Using declarations are only allowed in the declarative part of a package body. See section 2.7.1 for more information on using declarations. See

k_slot ::= "⟨" *commentary* "⟩" [*tag*]

tag ::= "(" *digit*{*digit*} ")" | *identifier*

Here **commentary** stands for an arbitrary sequence of characters not including "⟩". The optional tag is used to identify the k-slot when the Ada construct whose place it is holding is provided in a refinement or replacement step. If the tag is omitted, then an anonymous tag is implicitly introduced to identify the k-slot. The next refinement or replacement step in the script that also omits the tag will be treated as if it referred to this anonymous tag.

2.4 Names and Expressions

2.4.1 Names

name ::=
simple_name
| *indexed_component*
| *selected_component*

prefix ::= *name* | *function_call*

Use of an operator symbol as a name is not supported.

Slices are not supported.

Character literals and attributes are taken as primaries rather than names.

2.4.1.1 Indexed Components

indexed_component ::= *prefix* "(" *expression* {"," *expression*} ")"

2.4.1.2 Slices

Slices are not supported.

2.4.1.3 Selected Components

selected_component ::= *prefix* "." *selector*

Other rules imply that the only supported forms of selected components denote a record component or an entity declared in the visible part of a package.

2.4.1.4 Attributes

attribute ::= *prefix* "'" *attribute_designator*
| *character_literal* "'" *attribute_designator*

Only the first form of attribute can be handled formally.

2.4.2 Literals

Literals are as in ALRM except that the literal *null* and based real literals are not handled formally.

2.4.3 Aggregates

aggregate ::=
"(" *component_associations* ["," "*others*" "=>" *expression*] ")"
| "(" "*others*" "=>" *expression* ")"

component_associations ::=
named_association {"," *named_association*}
| *positional_association* {"," *positional_association*}

named_association ::=
choice {"|" *choice*} "=>" *expression*

positional_association ::= *expression*

choice ::=
simple_expression
| *discrete_range*
| "*others*"
| *simple_name*

A mixture of named and positional component associations is not supported in an aggregate (although an *others* choice is permitted when positional component associations are used).

To be handled formally, an aggregate must appear either as the operand of a qualified expression or as a component in another aggregate or in a variable or constant declaration. The restriction means that the Ada type of the aggregate is available to guide its translation into Z. In the case of a subaggregates, the types are given by the range attributes of the multidimensional array type that qualifies the outermost enclosing aggegrate.

2.4.3.1 Record Aggregates

To be handled formally, a record aggregate must appear either as the operand of a qualified expression or as the initial value in a variable or constant declaration, so that the Ada type is available to guide the translation into Z.

A record aggregate with an *others* choice cannot be handled formally.

2.4.3.2 Array Aggregates

To be handled formally, an array aggregate must appear either as the operand of a qualified expression or as a subaggregate (i.e., inside another aggregate as part of a multidimensional aggregate) or as the initial value in a variable or constant declaration. The restriction means that the Ada type of the aggregate is available to guide its translation into Z. In the case of subaggregates, the types are given by the range attributes of the multidimensional array type that qualifies the outermost enclosing aggegrate.

2.4.4 Expressions

expression ::=
relation {"*and*" *relation*} | *relation* {"*and*" "*then*" *relation*}
| *relation* {"*or*" *relation*} | *relation* {"*or*" "*else*" *relation*}
| *relation* {"*xor*" *relation*}

relation ::=
simple_expression [*relational_operator simple_expression*]
| *simple_expression* ["*not*"] "*in*" *range*
| *simple_expression* ["*not*"] "*in*" *type_mark*

simple_expression ::=
[*unary_adding_operator*] *term* {*binary_adding_operator term*}

term ::=
factor {*multiplying_operator factor*}

factor ::= *primary* ["∗∗" *primary*] | "*abs*" *primary* | "*not*" *primary*

primary ::=
numeric_literal | *aggregate* | *string_literal*
| *name* | *function_call* | *type_conversion*
| *qualified_expression* | "(" *expression* ")" | *attribute*
| *auxiliary_expression*

auxiliary_expression ::= "⟦" *z_expression* "⟧"

The *null* expression and allocators are not supported.

See section 2.4.2 for restrictions on literals.

Logical operators on boolean arrays are supported and handled formally.

z_expression stands for the construct called *Expr* in ProofPower *Description Manual* [5].

2.4.5 Operators and Expression Evaluation

The syntax for the six classes of operator is exactly as in ALRM except that the catenation operator & is not supported.

2.4.5.1 Logical Operators and Short-circuit Control Forms

The short-circuit control forms *and then* and *or else* are treated formally as synonymous with *and* and *or* respectively.

2.4.5.2 Relational Operators and Membership Test

2.4.5.3 Binary Adding Operators

Addition and subtraction are only handled formally for integer types.

Catenation of one array element with another to produce a two-element array is not handled formally. Catenation of arrays with arrays and of arrays with array elements is handled formally.

2.4.5.4 Unary Adding Operators

The unary adding operators are only handled formally for integer types.

2.4.5.5 Multiplying Operators

The multiplying operators are only handled formally for integer types.

2.4.5.6 Highest Precedence Operators

The absolute value operation is only handled formally for integer types.

2.4.5.7 Accuracy of Operations with Real Operands

Ada real types are represented in Z as subsets of the type of real numbers. The arithmetic operators on real numbers in Ada are translated into corresponding operators in Z. These operators will therefore have the semantics defined for them in the Z toolkit. This will typically be the semantics of the field of real numbers of pure mathematics. The translation into Z is then an idealisation of the Ada semantics and it is the user's responsibility to deal with issues of numeric analysis by formulating pre- and post-conditions appropriately.

2.4.6 Type Conversions

type_conversion ::= *type_mark* "(" *expression* ")"

A type conversion appearing as an actual parameter whose corresponding formal parameter has mode *out* or *in out* cannot be handled formally.

A type conversion can only be handled formally if the operand and target types are integer or real types. Type conversions from integer to real and from real to integer are supported.

2.4.7 Qualified Expressions

qualified_expression ::=
type_mark "'" "(" *expression* ")"
| *type_mark* "'" *aggregate*

2.4.8 Allocators

Allocators are not supported.

2.4.9 Static Expressions and Static Subtypes

Not all static expressions can be completely evaluated when a Compliance Notation script is checked. For example, the value of an attribute such as *INTEGER'FIRST* depends on the compiler being used. This sometimes results in a VC being generated (see sections 2.3.5 and 3.3.19).

2.4.10 Universal Expressions

Universal expressions of integer and real types are handled formally.

2.5 Statements

2.5.1 Simple and Compound Statements — Sequences of Statements

sequence_of_statements ::= *statement* {*statement*}

The statement forms comprises the optionally labelled, simple and compound statement forms of ALRM augmented with specification statements, assertion statements and k-slots.

statement ::=
simple_statement
| *compound_statement*
| *label statement*
| [*logical_constant_declaration*] *specification_statement* [*tag*]
| *assertion_statement*
| *k_slot_statement*

All the statement forms other than those associated with tasks and machine code insertions are supported:

simple_statement ::=
null_statement
| *assignment_statement*
| *procedure_call_statement*
| *exit_statement*
| *return_statement*
| *goto_statement*

compound_statement ::=
if_statement
| *case_statement*
| *loop_statement*
| *block_statement*

A specification statement gives a formal specification of a sequence of statements to be given later in the script (in a refinement step or a replacement step, see section 2.15). The formal specification comprises a list of variables called the frame and Z predicates called the pre-condition and the post-condition. A specification statement may be preceded by an optional Z declaration defining variables known as logical constants.

logical_constant_declaration ::= "$*con*" *z_declaration* "•"

specification_statement ::=
"Δ" *frame* "[" [*pre_condition* ","] *post_condition* "]"
| "Δ" *frame* "{" *pre_condition* "}"

frame ::= [*z_identifier* {"," *z_identifier*}]

pre_condition ::= *z_predicate*

post_condition ::= *z_predicate*

Here the symbols **z_declaration**, **z_identifier** and **z_predicate** stand for the constructs referred to in ProofPower *Description Manual* [5] as *Decl*, *Id* and *Pred* respectively.

Either, but not both, the pre-condition or the post-condition may be omitted. Braces are used instead of square brackets when the post-condition that has been omitted. The omitted predicate is taken to be *true*.

The semantics of specification statements are discussed in section 3.3. Z variable names ending in a subscript 0, e,g., MY_VAR_0, are called initial variables. Initial variables are allowed to appear free in post-conditions, but may not appear free in pre-conditions.

The Z declaration in a logical constant declaration must not comprise any schemas-as-declarations. The specification statement following a logical constant declaration must have a pre-condition, and that pre-condition must have the form: $X_1 = E_1 \wedge X_2 = E_2 \wedge ... X_k = E_k \wedge A$ or $X_1 = E_1 \wedge X_2 = E_2 \wedge ... X_k = E_k$, where X_1, X_2, ... X_k stand for the variables declared by the Z declaration. The variables in the Z declaration may appear in any order, but the defining equations in the pre-condition must satisfy a rule of definition-before-use. I.e., No X_i may appear free in E_1; only X_1 may appear free in E_2, only $X_1$and X_2 may appear free in E_3, and so on.

Anonymous tags are introduced for specification statements appearing as statements without a tag in the same way as for k-slots (see section 2.3.9).

assertion_statement ::=
"*Γ*" "{" *pre_condition* "}"

An assertion statement cannot be refined and has no effect on the translation of a script into Ada. An assertion statement otherwise has the same formal semantics as the specification statement obtained by replacing the leading *Γ* by a *Δ*.

kslot_statement ::= *k_slot*

A k-slot used as a statement stands as a placeholder for a sequence of statements to be given later in the script without any formal specification. A k-slot used as a statement acts as a break in the chain of formal development and cannot be handled formally.

label ::= "<<" *simple_name* ">>"

Labels have no significance in the Compliance Notation (since `goto` statements are not handled formally).

null_statement ::= "*null*" ";" | "$*nothing*" ";"

The statement `$nothing` is semantically equivalent to `null` but causes no code to be generated in the Ada program. It may be used, for example, when formally specifying an `if` statement whose `else` part has been omitted in the Ada program: the `else` part can be given in the Compliance Notation script as a specification statement which is later refined by `$nothing`.

2.5.2 Assignment Statement

assignment_statement ::= *name* ":=" *expression* ";"

2.5.3 If Statements

if_statement ::=
"*if*" *condition* "*then*"
sequence_of_statements
{ "*elsif*" *condition* "*then*"
sequence_of_statements }
["*else*"
sequence_of_statements]

condition ::= *expression*

2.5.4 Case Statements

case_statement ::=
"*case*" *expression is*
case_statement_alternative
{ *case_statement_alternative* }
"*end*" "*case*" ";"

case_statement_alternative ::=
"*when*" *choice* { "|" *choice* } =>
sequence_of_statements

See section 2.4.3 for the syntax of *choice*.

2.5.5 Loop Statements

loop_statement ::=
[*block_name*:]
[*iteration_scheme*]
[*till_predicate*] "*loop*"
sequence_of_statements
"*end*" "*loop*" [*block_name*] ";"

block_name ::=
simple_name
| "$*block*" *simple_name*

iteration_scheme ::=
"*while*" *condition*
| "*for*" *loop_parameter_specification*

loop_parameter_specification ::=
identifier "*in*" ["*reverse*"] *discrete_range*

till_predicate ::=
"$*till*" *auxiliary_expression*

See section 2.4.4 for the syntax of auxiliary expression.

To be handled formally, the sequence of statements comprising the body of a loop statement must comprise a single specification statement possibly with a tag (see section 2.5.1).

A loop which is anonymous in Ada may be given a name in the Compliance Notation using the keyword $*block*.

2.5.6 Block Statements

block_statement ::=
[*block_name*:]
["*declare*"
declarative_part]
"*begin*"
sequence_of_statements
"*end*" [*block_name*] ";"

To be handled formally, a block statement must appear as the only statement in the sequence of statements on the right-hand side of a refinement step or a replacement step (see section 2.15).

A block which is anonymous in Ada may be given a name in the Compliance Notation using the keyword $*block*.

2.5.7 Exit Statements

exit_statement ::=
"*exit*" [*loop_name*] ["*when*" *condition*] ";"

loop_name ::= *identifier*

To be handled formally, the loop being exited by an exit statement must have a *till* predicate.

2.5.8 Return Statements

return_statement ::=
"*return*" [*expression*] ";"

2.5.9 Goto Statements

goto_statement ::=
"*goto*" *simple_name* ";"

Goto statements cannot be handled formally.

2.6 Subprograms

2.6.1 Subprogram Declarations

subprogram_declaration ::= ["$*implicit*"] *subprogram_specification* ";"

subprogram_specification ::=
informal_subprogram_specification
| *formal_subprogram_specification*

informal_subprogram_specification ::=
"*procedure*" *identifier* [*formal_part*]
| "*function*" *designator* [*formal_part*] "*return*" *type_mark*

formal_subprogram_specification ::=
"*procedure*" *identifier* [*formal_part*]
procedure_specification_statement
| "*function*" *designator* [*formal_part*] "*return*" *type_mark*
function_specification_statement

designator ::= *identifier*

formal_part ::=
"(" *parameter_specification* { ";" *parameter_specification* } ")"

parameter_specification ::=
identifier_list : *mode* *type_mark* [":=" *expression*]

mode ::= ["*in*"] | "*in*" "*out*" | "*out*"

procedure_specification_statement ::=
"Δ" *frame* ["Ξ" *global_dependencies*] "[" [*pre_condition* ","] *post_condition* "]"
| "Δ" *frame* ["Ξ" *global_dependencies*] "{" *pre_condition* "}"

function_specification_statement ::=
"Ξ" *global_dependencies* "[" [*pre_condition* ","] *post_condition* "]"
| "Ξ" *global_dependencies* "{" *pre_condition* "}"

global_dependencies ::= [*z_identifier* {"," *z_identifier*}]

A subprogram declaration preceded by the Compliance Notation keyword $*implicit* allows a formal subprogram declared in a package specification to be used in a package body before its body is introduced. The subprogram declaration is not included in the Ada program.

Use of an operator symbol as a function designator is not supported for user-defined functions.

A subprogram, procedure or function is said to be a *formal* subprogram, procedure or function if it has a specification statement and is said to be informal otherwise. Subprogram calls can only be handled formally for formal subprograms.

A formal function may not have side effects — the frame of its specification statement is implicitly empty.

A formal subprogram may read the values of variables declared outside the subprogram. Such variables must be identified in the global dependencies list of the subprogram specification statement and must be in scope at the point that the subprogram is declared. These can either be program variables or auxiliary variables. If auxiliary variables are used, they must be declared in a package other than the package (if any) containing the subprogram specification. The variables need not actually be global: they might be local to a package or subprogram containing the function specification.

In a parameter specification, if a default expression is given, the default expression can only be handled formally if it contains no variables.

2.6.2 Formal Parameter Modes

Certain rules are applied when a formal procedure call is processed to ensure that parameter aliasing does not compromise the soundness of the VCs generated for the call. The rules also make the soundness of the VCs independent of the parameter passing mechanism used by the Ada compiler.

The rules are defined using the notion of an *entire variable* of an actual parameter of mode `in out` or `out`. The entire variable is what is obtained by removing all array indexes and record component selectors from the parameter. For example, the entire variable of the actual parameter `A(I).DAY` is `A`.

1. A variable in the frame or global dependencies list of the procedure must not appear as the entire variable in an actual parameter of mode `in out` or `out`.
2. A variable in the frame of the procedure must not appear anywhere in any expression in the actual parameter list.
3. The entire variable of an actual parameter of mode `in out` or `out` must not occur anywhere else in any expression in the parameter list.

The last rule disallows an `in out` or `out` mode actual parameter such as `B(B(I))` in which `B` appears both as the entire variable and as part of the index expression.

2.6.3 Subprogram Bodies

subprogram_body ::=
["$*deferred*"]
subprogram_specification "*is*"
[*declarative_part*]
"*begin*"
sequence_of_statements
"*end*" [*designator*] ";"

Exception handlers are not supported.

A subprogram body may be preceded by the keyword $*deferred* is called a deferred subprogram. In a deferred subprogram, the declarative part must contain only k-slots and the sequence of statements must comprise a single k-slot. The generation of Z corresponding to the declarative parts and the sequence of statements of a deferred subprogram is deferred until these k-slots are expanded.

This

2.6.4 Subprogram Calls

procedure_call_statement ::=
name [*actual_parameter_part*]";"

function_call ::=
name [*actual_parameter_part*]

actual_parameter_part ::=
"(" *named_parameter* {"," *named_parameter*} ")"
| "(" *positional_parameter* {"," *positional_parameter*}} ")"

name_parameter ::= *formal_parameter* "=>" *expression*

formal_parameter ::= *simple_name*

positional_parameter ::= *expression*

Positional and named parameter association may not be mixed within one subprogram call.

2.6.5 Parameter and Result Type Profile — Overloading of Subprograms

Overloading of subprogram names cannot be handled formally.

2.6.6 Overloading of Operators

No overloading of operator symbols is supported except for the limited form of overloading of operator symbols using renaming declarations (see section 2.8.5).

2.7 Packages

2.7.1 Package Structure

package_declaration ::= *package_specification* ";"

package_specification ::=
"*package*" *defining_program_unit_name* "*is*"
visible_part
["*private*"
private_part
"*end*" [[*parent_unit_name*"."]*simple_name*]

defining_program_unit_name ::= [*parent_unit_name*"."]*simple_name*

parent_unit_name ::= *name*

visible_part ::= { *package_declaration* }

package_declaration ::=
{ *basic_declarative_item*
| *subprogram_declaration*
| *auxiliary_declaration*
| *renaming_declaration* }

private_part ::= { *package_declaration* }

auxiliary_declaration ::= "$*auxiliary*" *z_declaration* ";"

Here *z_declaration* stands for the first alternative for the construct called *BasicDecl* in ProofPower *Description Manual* [5], i.e., a list of Z names followed by a colon followed by a Z expression.

A variable introduced by an auxiliary declaration ia a Z variable referred to as an **auxiliary variable**. Auxiliary variables are used in specification statements in a package specification as abstractions of all or part of the state of the package body. Together with using declarations, auxiliary variables support data refinement.

package_body ::=
"*package*" "*body*" *defining_program_unit_name* "*is*"
declarative_part
["*begin*"
sequence_of_statements]
"*end*" [[*parent_unit_name*"."]*simple_name*] ";"

using_declaration ::=
"$*using*" *simple_declaration* { *simple_declaration* }
"$*implement*" *z_identifier* "$*by*" *invariant* ";"

simple_declaration ::=
object_declaration
| *type_declaration*
| *subtype_declaration*

invariant ::= *z_predicate*

Here **z_identifier** and **z_predicate** stand for the constructs called *Id* and *Pred* respectively in ProofPower *Description Manual* [5].

The visible part declarations after a using declaration may not be k-slots or auxiliary variable declarations.

Auxiliary declarations and using declarations together support data refinement. A using declaration relates the values taken by one or more variables in the package body with the value of a Z variable introduced in an auxiliary declaration in the package specification. If using declarations are given the package body must have a sequence of statements.

2.7.2 Package Specifications and Declarations

If a procedure in a package specification has a specification statement, then it is this specification statement which is used in the generation of VCs for calls of the procedure outside the package body.

When a package is named in a with clause, Z global variables are introduced representing the types, constants and functions in the package. The Z variable names are derived from the Ada name by converting it to upper case and prefixing the result with the package name (also converted to upper case) and an 'o'. For example a function *fnc* define in package *pck* would give rise to the Z global variable *FNCoPCK* defined in an axiomatic description capturing the signature and the formal specification of the function (if any).

2.7.3 Package Bodies

If a subprogram has a specification statement both in the package specification and in the package body, then VCs are generated to ensure that the specification statement in the package body refines that in the package specification.

If a subprogram has a specification statement in the package body, then it is this specification statement which is used in the generation of VCs for calls of the subprogram inside the package body. The subprogram will be treated as an informal procedure if it is used before its implementation within the package body.

When a package body is introduced, Z global variables are automatically bought into scope representing the types and constants, *but not the functions* in the package (as stated in the package specification). The names of these global variables are *not* prefixed with the package name. Global variables for any functions in the package are introduced as the function implementations are processed.

A Z global variable representing a function in the package is introduced at the point where the implementation of the function is provided. Thus calls of the function will not be handled formally within the package body, if they appear before the implementation of the function.

2.7.4 Private Type and Deferred Constant Declarations

private_type_declaration ::=
"*type*" *identifier* "*is* ["*limited*"] "*private*" ";"

deferred_constant_declaration ::=
identifier_list : "*constant*" *type_mark* ";"

The Z global variables corresponding to private type declarations and deferred constant declarations are introduced using the information in the actual declarations in the private part of the package specification.

2.8 Visibility Rules

The visibility rules for the Compliance Notation may be described in terms of the Ada visibility rules given the method for extracting the Ada program from a literate script. The Ada program is extracted from a literate script as follows:

1. Any auxiliary declarations are removed;
2. Any using declarations are replaced by their constituent simple declarations;
3. Any till predicates are removed;
4. Any references clauses are removed;
5. Any specification statements not occurring as statements are removed;
6. Any assertion statements are removed;
7. Any subprogram declarations preceded by the keyword $*implicit* are removed;
8. The constituent web clauses of the script are scanned in order and k-slots and specification statements in each web clause are expanded in turn;
   (a) A k-slot is expanded by replacing its text by the text on the right hand side of the corresponding replacement step or arbitrary replacement step and then, in the case of an ordinary replacement step, recursively expanding any k-slots or specifications in the resulting text;
   (b) A specification statement occurring as a statement is expanded by replacing its text by the text on the right hand side of the corresponding refinement step, replacement step or arbitrary replacement step and then recursively expanding any k-slots or specifications in the resulting text;
9. All statements of the form "`nothing;`" are removed and any else parts, others parts or package initialisation statements which become empty as a result are also removed.
10. Any procedure or function specification statements are removed.

The phrase "current Ada program" or "current Ada program for *C*" is used below to refer to the result of carrying out the above steps on an initial fragment of the literate script, up to the point at which some construct, *C*, appears.

If any k-slots or specification statements remain unexpanded after the above steps, then the script is incomplete.

2.8.1 Declarative Region

The declarative region in which a Compliance Notation construct occurs is determined by the declarative region it occupies in the current Ada program.

2.8.2 Scope of Declarations

Within one compilation unit, the scope of a declaration includes any Compliance Notation construct, C, which would be in the scope of the declaration in the current Ada program for C.

A compilation unit U_1 may only refer to entities defined in another compilation unit, U_2, under the following circumstances: *(a)*, U_1 is the package body implementing the package specification in U_2 and the entity is a type, constant or variable, or, *(b)*, U_2 gives the package specification of a package identified in the context clause of U_2. In case *(a)*, the entity must be referred to by its simple name, without a package name prefix.

2.8.3 Visibility

No overloading of names or operator symbols is allowed except for the limited form of overloading of operator symbols using renaming declarations (see section 2.8.5).

Declaring a name in an inner scope which is already declared in an outer scope is not handled formally.

2.8.4 Use Clauses

use_clause ::= *use_package_clause* | *use_type_clause*

use_package_clause ::= "*use*" *identifier* {"," *identifier*} ";"

use_type_clause ::= "*use*" "*type*" *identifier* {"," *identifier*} ";"

A use package clause makes the simple names declared in one or more packages available for use in the current declarative region. Occurrences of the simple names are converted into the corresponding expanded names when handled formally.

A package mentioned in a use clause must have been processed formally and must not contain any unexpanded k-slots. A package may not be mentioned more than once in the use clauses occurring in each declarative region.

The syntax of Ada '95 use type clauses is supported. Use type clauses have no semantic effect in the Compliance Notation (since the Z representations of the predefined operators for any type are always directly available and may not be redefined).

2.8.5 Renaming Declarations

renaming_declaration ::=
object_renaming
| *package_renaming*
| *operator_symbol_renaming*
| *subprogram_renaming*

object_renaming ::= *identifer* ":" *type_mark renames name semi*

package_renaming ::= *package identifier* "*renames*" *name semi*

operator_symbol_renaming ::=
"*function*" *operator_symbol formal_part* "*return*" *type_mark*
"*renames*" *simple_name* "." *operator_symbol* ";"

subprogram_renaming ::=
informal_subprogram_specification
"*renames*" *simple_name* "." *operator_symbol* ";"
| *informal_subprogram_specification* "*renames*" *name* ";"

In an operator symbol renaming, the two operator symbols must be the same. This form of renaming declaration is intended for operators such as *pck*."+" introduced implicitly when a numeric type is declared in package *pck*.

If the object in an object renaming is a constant, a Z global variable is introduced and defined to be equal to the renamed constant. If the object is a variable, the new name for the variable may be used as a synonym for the old name in Ada and Z expressions in the scope of the renaming declaration.

A package renaming, `package new renames old`, is considered to be equivalent to the declaration of a package named `new` containing a sequence of renaming declarations and subtype declarations. Each declaration in the package `new` corresponds to a name declared in the package old. If a name in the package `old` denotes a type, the corresponding declaration in `new` is a subtype declaration, e.g., `subtype ty is old.ty`. If the name is an object or subprogram, then the corresponding declaration is a renaming declaration, e.g., `x : float renames old.x`.

In a subprogram renaming, if the subprogram being renamed is a formal procedure, then the formal specification for the new procedure name is taken from that for the old one. If the subprogram being renamed is a function or a predefined operator, a Z global variable for the new function name is introduced and defined to be equal to the Z global variable corresponding to the renamed function or predefined operator.

2.8.6 The Package Standard

Compliance Notation scripts are processed in an environment which includes the declarations in the package *STANDARD* with the following restrictions:

1. The enumeration literals of the type *CHARACTER* are not provided.
2. The package *ASCII* is not provided.
3. The only predefined operators for the type *STRING* are =, &, and /=.

4. The type *DURATION* is not provided.

2.8.7 The Context of Overload Resolution

No overloading of names is supported except for the limited form of overloading of predefined operators using renaming declarations (see section 2.8.5). The question of overload resolution does not arise for predefined operators, since the underlying semantics of these operators is built into the notation.

2.9 Tasks

Tasks are not supported.

2.10 Program Structure and Compilation Issues

2.10.1 Compilation Units — Library Units

compilation ::= *k_slot* | *compilation_unit* {*compilation_unit*}

compilation_unit ::= *context_clause library_unit* | *context_clause secondary_unit*

library_unit ::= [*private*] *package_declaration* | *subprogram_body*

secondary_unit ::= *library_unit_body* | *subunit*

library_unit_body ::= *package_body*

The only declarations allowed as compilation units are package declarations.

Generic packages are not supported.

2.10.1.1 Context Clauses — With Clauses

context_clause ::= { *with_clause* | *references_clause* | *use_clause* } ;

with_clause ::= "*with*" *simple_name* {"," *simple_name*} ";"

references_clause ::= "$*references*" *simple_name* {"," *simple_name*} ";"

A with clause must refer to a package not a library subprogram.

To allow use of packages for purely informal purposes, the names in a with clause need not all identify packages whose specifications have been provided. The use of entities declared in such packages is not handled formally.

A context clause may also include a references clause. A references clause is used to identify packages that are needed to specify the semantics of a compilation unit. This allows such packages to be identified without including them in a with clause.

2.10.2 Subunits of Compilation Units

body_stub ::=
subprogram_specification "*is*" "*separate*" ";"
| "*package*" "*body*" *simple_name* "*is*" "*separate*"

subunit ::=
"*separate*" "(" *name* ")" *proper_body*

Tasks are not supported.

2.11 Exceptions

Exceptions are not supported.

2.12 Generic Units

Generic units are not supported.

2.13 Representation Clauses and Implementation-Dependent Features

2.13.1 Representation Clauses

representation_clause ::= *type_representation_clause* | *address_clause*

type_representation_clause ::= *length_clause*
| *enumeration_representation_clause* | *record_representation_clause*

2.13.2 Length Clauses

length_clause ::= "*for*" *attribute* "*use*" *simple_expression* ";"

2.13.3 Enumeration Representation Clauses

enumeration_representation_clause ::= "*for*" *simple_name* "*use*" *aggregate* ";"

2.13.4 Record Representation Clauses

record_representation_clause ::=
"*for*" *simple_name* "*use*"
"*record*" [*alignment_clause*]
{ *component_clause* }
"*end*" "*record*" ";"

alignment_clause ::= "*at*" "*mod*" *simple_expression* ";"

component_clause ::= *name* "*at*" *simple_expression* "*range*" *range* ";"

2.13.5 Address Clauses

address_clause ::= "*for*" *simple_name* "*use*" "*at*" *simple_expression* ";"

2.13.6 Change of Representation

Note that because derived types are not supported the approach taken in the example in section 13.6 of ALRM would not be supported by the Compliance notation.

2.13.7 The Package System

The Compliance Notation does not provide the package *SYSTEM*.

2.13.8 Machine Code Insertion

Machine code insertions are not supported.

2.13.9 Interface to Other Languages

The interface pragma is supported (as are any other pragmas meeting the restrictions of section 2.2.8).

2.13.10 Unchecked Programming

The generic library subprograms for unchecked storage deallocation and unchecked type conversions are not supported.

2.14 Input-Output

The predefined packages for input-output are not provided.

2.15 Web Clauses and Compliance Notation Scripts

web_clause ::=
z_paragraph
| Ⓢ*CN*
compilation
■
| Ⓢ*CN*
refinement_step
■
| Ⓢ*CN*
replacement_step
■
| Ⓢ*CN*
arbitrary_replacement_step

■

refinement_step ::=
[*tag*] "⊑" *sequence_of_statements*

replacement_step ::=
[*tag*] "!⊑" *sequence_of_statements*
| [*tag*] "≡" *compilation*
| [*tag*] "≡" *private_part*
| [*tag*] "≡" *visible_part*
| [*tag*] "≡" *declarative_part*

arbitrary_replacement_step ::=
[*tag*] "!≡" *lexical_elements*

If the tag is omitted in any of the above constructs, the refinement, replacement or arbitrary replacement applies to the immediately preceding k-slot or specification statement without an explicit tag.

The kind of refinement or replacement must be appropriate to the corresponding k-slot or specification statement. For example, it is not allowed to refine a declaration k-slot or to replace a specification statement with a declaration. This restriction does not apply to an arbitrary replacement step.

A replacement step which introduces a declaration into a declarative part has the implicit effect of widening the frames of any specification statements in the scope of that declarative part to include the variables declared on the right-hand side of the replacement step. Thus, local variables introduced in this way after a specification statement become available for use in the refinement of that specification statement.

Here, **lexical_elements** stands for an arbitrary sequence of the lexical elements of Ada. This means that string and character quotation characters must be properly balanced in an arbitrary replacement step. No other restrictions apply to the right-hand side of an arbitrary replacement step, which is copied verbatim, with all format effectors preserved, into the Ada program when the replacement is expanded.

compliance_notation_application ::= *compliance_notation_script*
{*compliance_notation_script*}

compliance_notation_script ::= *web_clause*
{*web_clause*}

A compliance notation application is made of one or more scripts each of which is a sequence of web clauses. In practice, the scripts are interleaved with ProofPower metalanguage commands to identify the scripts (see section 3.5 and *Compliance Tool — User Guide* [8]).

A script may contain at most one Ada compilation unit.

The compliance notation requires any formal dependencies between scripts to be expressed in a linear sequence.

COMPLIANCE NOTATION SEMANTICS

The semantics of the Compliance Notation may be understood in terms of the translation of one or more Compliance Notation script into *(a)* Ada source code, and *(b)* one or more Z documents. The translation into Ada source code is described in section 2.8 above. The translation of a script into a Z document is described in this chapter.

The main purpose of the Z document is to provide a set of Z conjectures, the verification conditions, or VCs, whose truth entails that the formally specified parts of the Ada program satisfy their specifications. The Z document also contains any Z paragraphs which appear in the script together with Z paragraphs that are automatically generated from some of the Ada declarations in the script. These Z paragraphs together with the extensions to the Z library described in chapter 4 provide the vocabulary in terms of which the VCs are couched.

In sections 3.1 to 3.5 below, the generation of the Z document is described under the following headings:

Expressions Ada expressions are translated into Z expressions appearing in VCs or in generated Z paragraphs according to the context of use.

Declarations Type, constant and function declarations are translated into Z paragraphs.

VC Generation VCs are generated for refinement steps and for certain other constructs.

Domain Conditions Optionally, VCs may be generated with additional hypotheses justifying the application of partial functions.

Program Structure The top level structure of the literate scripts gives rise to the creation of one or more ProofPower theories.

3.1 Translation of Expressions

In this section the translation of Ada expressions into Z is described. The examples used depend on the following definitions:

type **DAY** *is* (**MON**, **TUE**, **WED**, **THU**, **FRI**, **SAT**, **SUN**);
subtype **INDEX** *is INTEGER range 2 .. 5*;
type **ARR** *is array* (*INDEX*) *of DAY*;
type **REC** *is record START* : *DAY*; *FINISH* : *DAY*; *end record*;
type **ARR2** *is array* (*INDEX*, *DAY*) *of REC*;
function **MAX** (*x*, *y*: *DAY*) *return DAY* Ξ [*true*];
function **CONST** *return INTEGER* Ξ [$CONST = 1001$];
function **SUMXY** *return INTEGER* Ξ X, Y [$SUMXY(X, Y) = X + Y$];
function **XPLUSY** (*Y* : *INTEGER*) *return INTEGER* Ξ X [$XPLUSY(X)(Y) = X + Y$];
ArrVar : **ARR**;
Arr2Var : **ARR2**;

$RecVar$: **REC**;

Sections 3.1.1 to 3.1.13 below discuss the translation of the various forms of Ada expression. Many of these translations depend on the some extensions to the Z library which support predefined types and their operators and attributes. These extensions are described in chapter 4.

3.1.1 Literals

Null literals and based real literals cannot be handled formally and do not have a translation.

Integer literals both based and decimal are translated into Z decimal literals. For example, both *35* and *2#0010_0011#* are translated as *35*.

Real literals are translated into expressions involving the operator $(\_e\_)$. For example, *3.14159* is translated as $314159\ e\ {\sim}5$. The translation, $i\ e\ j$ of a real literal is always normalised so that i is not divisible by 10.

An enumeration literal is translated in the same way as an identifier (see below).

3.1.2 Identifiers

A simple name is translated into Z as a variable (global or local) by converting all letters into upper case. For example, both *Var* and *var* are translated as *VAR*.

An identifier prefixed by a package name (which is strictly speaking a selected component in Ada terminology) is translated into a Z variable by converting all letters into upper case and replacing each '.' by '*o*'. For example, *Pack.object* is translated as *PACKoOBJECT*.

Attribute names are translated into Z global variables whose names are formed by converting letters to upper case and replacing the prime with a '*v*'. For example, $ARR'LENGTH$ is translated as *ARRvLENGTH*.

3.1.3 Record Aggregates

A record aggregate using either named or positional association is translated into a Z binding display.

For example both $REC'(WED,\ SUN)$ and $REC'(START{=>}WED,\ FINISH{=>}SUN)$ are translated as $(FINISH \mathrel{\widehat=} SUN,\ START \mathrel{\widehat=} WED)$.

3.1.4 Array Aggregates

A positional array aggregate with no *others* part is translated as a Z sequence display composed with a numerical shift operator. For example,

$$ARR'(SAT,\ SUN,\ MON,\ TUE)$$

is translated as

$$succ^{\,1\ -\ ARRvFIRST}_{\ 3}\ \langle SAT,\ SUN,\ MON,\ TUE\rangle.$$

A named array aggregate with no *others* part is translated as the union of functions with a singleton range, and with a domain formed by representing the aggregate choice as a union of singleton sets or integer intervals: For example,

$$ARR'(MON \mathinner{..} THU \mid FRI \Rightarrow WED,\ SAT \mid SUN \Rightarrow TUE);$$

is translated as

$$((MON \mathinner{..} THU) \cup \{FRI\} \times \{WED\}) \cup (\{SAT,\ SUN\} \times \{TUE\}).$$

An *others* part is represented by a total function on the index set of the array with a singleton range. This is overridden with the translation of the rest of the aggregate (if any) For example,

$$ARR'(others \Rightarrow MON)$$

is translated as

$$(ARRvRANGE \times \{MON\}),$$

and

$$ARR'(SAT,\ SUN,\ others \Rightarrow MON)$$

is translated as

$$(ARRvRANGE \times \{MON\}) \oplus (succ\ ^{1\ -\ ARRvFIRST}_{9}\ \langle SAT,\ SUN\rangle).$$

For $n \geq 2$, an n-dimensional array aggregate is translated by first applying the above translation as if it were an array of arrays of arrays of arrays ... (where each array is one-dimensional). This is done using the attributes of the n-dimensional array type in place of the qualifying type marks for the subaggegrates. The result is then converted to the right type using one of the functions *array_agg2*, *array_agg3*, and so on, defined in the extensions to the Z library described in chapter 4. For example, the aggregate:

$$\begin{array}{ll} ARR2'(& 2 \Rightarrow (\\ & MON \Rightarrow REC'(MON,\ TUE), \\ & others \Rightarrow REC'(TUE,\ WED)), \\ others \Rightarrow (& \\ & others \Rightarrow REC'(SAT,\ SUN))); \end{array}$$

is translated as:

$$\begin{array}{l} ((ARR2vRANGEv1 \times \{ARR2vRANGEv2 \times \{(FINISH \mathrel{\widehat=} SUN,\ START \mathrel{\widehat=} SAT)\}\}) \\ \quad \oplus (\{2\} \times \{(ARR2vRANGEv2 \times \{(FINISH \mathrel{\widehat=} WED,\ START \mathrel{\widehat=} TUE)\}) \\ \quad\quad \oplus (\{MON\} \times \{(FINISH \mathrel{\widehat=} TUE,\ START \mathrel{\widehat=} MON)\})\})) \end{array}$$

3.1.5 Unary Expressions

Unary plus is simply discarded in the translation. I.e., $+Exp$ is translated in the same way as Exp.

Other unary expressions are translated into an application of a Z function representing the operator to the translation of the operand. The Z functions representing the operators are either taken from the Z library or are defined in the theory *cn* described in chapter 4.

3.1.6 Binary Expressions

Binary expressions are translated into an application of an infix Z function representing the operator to the translated operands. The Z functions representing the operators are either taken from the Z library or are defined in the theory *cn* described in chapter 4. For example, $not(abs\ (-2) = (1\ +\ y))$ is translated as $not\ (abs\ (\sim\ 2)\ eq\ (1\ +\ Y))$.

3.1.7 Membership

Membership of a range is treated in the same way as the binary expressions and is supported by the operators *mem* and *notmem* in the theory *cn* described in chapter 4.

3.1.8 Attributes

The names of attributes are translated into Z identifiers as follows: function calls or other complex expressions in the prefix are replaced by the translation of their type; occurrences of the attribute $P'BASE$ are replaced by the Z identifier for the base type of P; the result of these replacements is then translated as an Ada name in the usual way; the prime character before the attribute designator is translated as a lower-case 'v', the attribute designator is translated into upper-case; finally, if the attribute has an argument, then the argument is statically evaluated to give an integer, say N and then 'vN' is appended to the Z identifier. For example, $arr'range$ and $arr'length(2)$ are translated as *ARRvRANGE* and *ARRvLENGTHv2* respectively. An attribute with an argument that cannot be statically evaluated cannot be handled formally.

The Compliance Notation divides the predefined attributes into three categories, A, B and C. The Z paragraphs that support category A attributes are generated automatically when a type declaration that introduces the attribute is processed. Examples of these Z paragraphs for the various sorts of type declaration are given in section 3.2 below.

The Z paragraphs that support category B attributes are generated if necessary when a use of the attribute is encountered. For example, when the expression $float'small$ is processed, if the Z global variable *FLOATvSMALL* is not in scope, then the following axiomatic description is generated

$$FLOATvSMALL : \mathbb{R}$$

Attributes in category C are only supported formally if they are preceded in the script by a Z paragraph introducing the global variable that corresponds to the attribute. (Note that this usage is also acceptable for category B attributes, but would cause a Z type-checking error for a category A attribute).

The category A and B attributes are listed in the following table. All other attributes except $P'BASE$ are in category C. The attribute $P'BASE$ is a special case as described in the rule for translating attribute names above.

Attribute	Type	Category
P'DELTA	Real	A
P'DIGITS	Integer	A
P'FIRST	P	A
P'LAST	P	A
P'LENGTH	Integer	A
P'POS	Function from Integer to P	A
P'PRED	Function from P'BASE to P'BASE	A
P'SUCC	Function from P'BASE to P'BASE	A
P'VAL	Function from P to Integer	A
P'RANGE	Range (of elements of P)	A
P'AFT	Integer	B
P'EMAX	Integer	B
P'EPSILON	Real	B
P'FORE	Integer	B
P'LARGE	Real	B
P'MACHINE_EMAX	Integer	B
P'MACHINE_EMIN	Integer	B
P'MACHINE_MANTISSA	Integer	B
P'MACHINE_OVERFLOWS	Boolean	B
P'MACHINE_RADIX	Integer	B
P'MACHINE_ROUNDS	Boolean	B
P'MANTISSA	Integer	B
P'SAFE_EMAX	Integer	B
P'SAFE_LARGE	Real	B
P'SAFE_SMALL	Real	B
P'SIZE	Integer	B
P'SMALL	Real	B

3.1.9 Indexed Components

An indexed component with a one-dimensional index is translated into a Z function application of the function representing the array to the translation of the index expression. For example, *ArrVar*(*3*) is translated as *ARRVAR 3*.

For $n \geq 2$, an indexed component with an n-dimensional into a Z function application of the function representing the array to the Z tuple whose components are given by translating the n index expressions. For example, *Arr2Var*(*3*, *MON*) is translated as *ARR2VAR* (*3*, *MON*).

(Note: the above description does not apply to an indexed component appearing as the left-hand side of an assignment statement. Such an assignment is effectively treated as an assignment of an aggregate value to the whole array, see section 3.3.2.)

3.1.10 Selected Components

Selection of a component from a record is translated into Z component selection. For example, *RecVar.START* is translated as *RECVAR.START*. (Note: the above description does not apply to a selected component appearing as the left-hand side of an assignment statement. Such an assignment is effectively treated as an assignment of an aggregate value to the whole record, see section 3.3.2.)

3.1.11 Function Calls

A function call using either positional or named argument association translates into a Z function application. For example, both $MAX(y{=>}THU,\ x{=>}FRI)$ and $MAX(FRI,\ THU)$ are translated as $MAX(FRI,\ THU)$. A call of a function with neither global dependencies or arguments is translated as a Z global variable, for example $CONST$ is translated as $CONST$. If a function has global dependencies, then these are formed into a tuple (if there is more than one), and become an extra argument to the Z function call. For example, the function calls $SUMXY$ and $XPLUSY(42)$ are translated as $SUMXY(X,\ Y)$ and $PLUSX\ X\ 42$ respectively.

3.1.12 Qualified Expressions

A qualified expression is translated in the same way as its operand. For example, $NATURAL'(4)$ is translated as 4.

3.1.13 Type Conversions

A type conversion is translated in the same way as its operand provided both the operand and the result both have integer types or both have real types. For example, $NATURAL(4)$ is translated as 4.

A type conversion from an integer type to a real type or from a real type to an integer type is translated into an application of an appropriate numeric conversion function (*real_to_integer* or *integer_to_real*). For example, $FLOAT(4)$ is translated as *integer_to_real* 4.

Type conversions involving types other than integer or real types cannot be handled formally and are not translated.

3.1.14 Array Sliding

Section 5.2.1 of ALRM describes an implicit conversion that may apply to expressions whose type is a subtype of an unconstrained array type. This implicit conversion is referred to in this document as *sliding*. It may apply to the left-hand side of an assignment statement and to *in* mode actual parameters in function or procedure calls. When the sliding conversion is applicable, the expression is translated as *slide(e, r)* where e is the translation of the unconverted expression and r is a Z identifier denoting the range of the converted array.

The sliding conversion also applies in ALRM to *in* and *in out* mode parameters. When it does apply, a verification condition is generated, see section refVCG.

3.1.15 Subtype Indications and Discrete Ranges

Object declarations, discrete ranges and component declarations involving subtypes which are not type marks are translated as if subtype declarations for the subtypes had been introduced. The Z identifiers for these type marks are automatically generated so as to be distinct from Z identifiers representing names in the Ada program. For example, the type declaration:

type sti_eg is array(*index*) *of integer range 1 .. 10*;

is translated into the following Z paragraphs:

1. A definition of an automatically generated name for the anonymous subtype of `integer`:

   Informal Z
   $$INTEGERs1 \mathrel{\widehat=} 1 \mathinner{..} 10$$

2. Definitions of the attributes of the anonymous subtype:

   Informal Z
   $$INTEGERs1vFIRST \mathrel{\widehat=} 1$$

   ...etc.

3. A definition for the array type `sti_eg`:

   Informal Z
   $$STI\_EG \mathrel{\widehat=} INDEX \rightarrow INTEGERs1$$

4. Definitions for the attributes of the array type:

   Informal Z
   $$STI\_EGvFIRST \mathrel{\widehat=} INDEXvFIRST$$

   ...etc.

Discrete ranges that do not include a type mark are translated using the Z name *universal_discrete* in place of the type mark.

3.2 Translation of Declarations

3.2.1 Enumeration Types

An enumeration type is represented in Z as a range of integers starting at *0*. Global variables are introduced corresponding to the type name, the enumeration literals and the supported attributes of the type.

For example, the enumeration type declaration:

type ENUM_TYPE is (*LIT1*, *LIT2*, *LIT3*);

is translated into a sequence of abbreviation definitions as follows:

$$LIT1 \mathrel{\widehat=} 0$$
$$LIT2 \mathrel{\widehat=} 1$$
$$LIT3 \mathrel{\widehat=} 2$$
$$ENUM\_TYPE \mathrel{\widehat=} LIT1 \mathinner{..} LIT3$$
$$ENUM\_TYPEvFIRST \mathrel{\widehat=} LIT1$$
$$ENUM\_TYPEvLAST \mathrel{\widehat=} LIT3$$
$$ENUM\_TYPEvSUCC \mathrel{\widehat=} (ENUM\_TYPE \setminus \{ENUM\_TYPEvLAST\}) \lhd succ$$
$$ENUM\_TYPEvPRED \mathrel{\widehat=} ENUM\_TYPEvSUCC^{\sim}$$
$$ENUM\_TYPEvPOS \mathrel{\widehat=} id\ ENUM\_TYPE$$
$$ENUM\_TYPEvVAL \mathrel{\widehat=} ENUM\_TYPEvPOS^{\sim}$$

3.2.2 Array Types

3.2.2.1 Constrained Array Types

A constrained array type is represented as a set of total functions. Z global variables are introduced corresponding to the type name and the supported attributes of the type. The attributes are only supported for one-dimensional arrays.

For example, the one-dimensional array type:

$$type\ ARRAY\_TYPE\ is\ array\ (INDEX\_TYPE)\ of\ ELEMENT\_TYPE;$$

is translated into a sequence of abbreviation definitions as follows:

$$ARRAY\_TYPE \mathrel{\widehat=} INDEX\_TYPE \rightarrow ELEMENT\_TYPE$$
$$ARRAY\_TYPEvFIRST \mathrel{\widehat=} INDEX\_TYPEvFIRST$$
$$ARRAY\_TYPEvLAST \mathrel{\widehat=} INDEX\_TYPEvLAST$$
$$ARRAY\_TYPEvLENGTH \mathrel{\widehat=} \#\ INDEX\_TYPE$$
$$ARRAY\_TYPEvRANGE \mathrel{\widehat=} INDEX\_TYPE$$
$$ARRAY\_TYPEvFIRSTv1 \mathrel{\widehat=} INDEX\_TYPEvFIRST$$
$$ARRAY\_TYPEvLASTv1 \mathrel{\widehat=} INDEX\_TYPEvLAST$$
$$ARRAY\_TYPEvLENGTHv1 \mathrel{\widehat=} \#\ INDEX\_TYPE$$
$$ARRAY\_TYPEvRANGEv1 \mathrel{\widehat=} INDEX\_TYPE$$

A 2-dimensional array type:

$$type\ ARRAY\_TYPE2\ is$$
$$array\ (INDEX\_TYPE1,\ INDEX\_TYPE2)\ of\ ELEMENT\_TYPE;$$

is translated into the sequence of abbreviation definitions:

$$ARRAY\_TYPE2 \mathrel{\widehat=} INDEX\_TYPE1 \times INDEX\_TYPE2 \rightarrow ELEMENT\_TYPE$$
$$ARRAY\_TYPE2vFIRST \mathrel{\widehat=} INDEX\_TYPE1vFIRST$$
$$ARRAY\_TYPE2vLAST \mathrel{\widehat=} INDEX\_TYPE1vLAST$$
$$ARRAY\_TYPE2vLENGTH \mathrel{\widehat=} \#\ INDEX\_TYPE1$$
$$ARRAY\_TYPE2vRANGE \mathrel{\widehat=} INDEX\_TYPE1$$
$$ARRAY\_TYPE2vFIRSTv1 \mathrel{\widehat=} INDEX\_TYPE1vFIRST$$
$$ARRAY\_TYPE2vLASTv1 \mathrel{\widehat=} INDEX\_TYPE1vLAST$$
$$ARRAY\_TYPE2vLENGTHv1 \mathrel{\widehat=} \#\ INDEX\_TYPE1$$
$$ARRAY\_TYPE2vRANGEv1 \mathrel{\widehat=} INDEX\_TYPE1$$
$$ARRAY\_TYPE2vFIRSTv2 \mathrel{\widehat=} INDEX\_TYPE2vFIRST$$
$$ARRAY\_TYPE2vLASTv2 \mathrel{\widehat=} INDEX\_TYPE2vLAST$$
$$ARRAY\_TYPE2vLENGTHv2 \mathrel{\widehat=} \#\ INDEX\_TYPE2$$
$$ARRAY\_TYPE2vRANGEv2 \mathrel{\widehat=} INDEX\_TYPE2{=}TEX$$

3.2.2.2 Unconstrained Array Types

An unconstrained array type is represented as a set of partial functions. For example, the unconstrained array type:

type ARRAY_TYPE3 is array (INDEX_TYPE range <>) of ELEMENT_TYPE;

is translated as:

Informal Z

$$ARRAY\_TYPE3 : \mathbb{P}\ (INDEX\_TYPE \nrightarrow ELEMENT\_TYPE)$$

3.2.3 Record Types

A record type is represented as a schema type. To support the generation of VCs for assignments to components of records, record update functions are also introduced. For example, the record type:

type date is record d : DAY; m : MONTH; y : YEAR; end record;

is translated as follows:

Informal Z

$$DATE \mathrel{\widehat=} [D : DAY;\ M : MONTH;\ Y : YEAR]$$

Informal Z

$[g1, g2, g3]$

$$DATEuD : [D : g1;\ W : g2;\ Y : g3] \times g1 \rightarrow [D : g1;\ W : g2;\ Y : g3];$$
$$DATEuM : [D : g1;\ W : g2;\ Y : g3] \times g2 \rightarrow [D : g1;\ W : g2;\ Y : g3];$$
$$DATEuY : [D : g1;\ W : g2;\ Y : g3] \times g3 \rightarrow [D : g1;\ W : g2;\ Y : g3]$$

$$\forall\ r : [D : g1;\ W : g2;\ Y : g3];\ x1 : g1;\ x2 : g2;\ x3 : g3$$
$$\bullet \quad DATEuD\ (r,\ x1) = (D \mathrel{\widehat=} x1,\ W \mathrel{\widehat=} r.W,\ Y \mathrel{\widehat=} r.Y)$$
$$\wedge \quad DATEuW\ (r,\ x2) = (D \mathrel{\widehat=} r.D,\ W \mathrel{\widehat=} x2,\ Y \mathrel{\widehat=} r.Y)$$
$$\wedge \quad DATEuY\ (r,\ x3) = (D \mathrel{\widehat=} r.D,\ W \mathrel{\widehat=} r.W,\ Y \mathrel{\widehat=} x3)$$

A record type with a discriminant part is translated similarly using a predicate in the schema to express the formal discriminant constraints. For example, the record type:

type buffer(size : integer) is record data : string (1 .. size); end record;

is translated into the following schema (followed by the record update functions for the two components of the schema).

Informal Z

$$BUFFER \mathrel{\widehat=}$$
$$[SIZE : INTEGER;\ DATA : STRING$$
$$\mid DATA \in \{array : STRING \mid dom\ array = 1\ ..\ SIZE\}]$$

3.2.4 Integer Types

A signed integer type declaration is represented as a range of integers. Z global variables are introduced corresponding to the type name and the supported attributes of the type. The attributes that are functions have Z signatures but no defining properties (since the defining properties are compiler-dependent as regards arithmetic overflow).

For example, the integer type declaration:

$$type\ INTEGER\_TYPE\ is\ range\ -1000\ ..\ 1000;$$

is translated as the sequence of abbreviation definitions: Z global variables are introduced corresponding to the type name and the supported attributes of the type.

$$INTEGER\_TYPE \mathrel{\widehat=} \sim 1000\ ..\ 1000$$
$$INTEGER\_TYPEvFIRST \mathrel{\widehat=} \sim 1000$$
$$INTEGER\_TYPEvLAST \mathrel{\widehat=} 1000$$

followed by the sequence of axiomatic descriptions:

$$INTEGER\_TYPEvSUCC : \mathbb{Z} \pfun \mathbb{Z}$$
$$INTEGER\_TYPEvPRED : \mathbb{Z} \pfun \mathbb{Z}$$
$$INTEGER\_TYPEvPOS : \mathbb{Z} \pfun \mathbb{Z}$$
$$INTEGER\_TYPEvVAL : \mathbb{Z} \pfun \mathbb{Z}$$

A modular type is represented by a free type with one constructor representing the Val attribute of the type. The domain of the constructor is the set of integers representable in the modular type. The free type definition is preceded by an abbreviation definition for the Modulus attribute and followed by an axiom description for the function which maps the integers onto the modular type and abbreviation definitions for the other attributes of the type.

For example, the modular type declaration:

$$type\ mod5\ is\ mod\ 5;$$

is translated into the following sequence of Z paragraphs:

$$MOD5vMODULUS \mathrel{\widehat=} 5$$
$$MOD5 ::= MOD5vVAL\ (0\ ..\ MOD5vMODULUS - 1)$$

$$MOD5\_of\_int : \mathbb{Z} \fun MOD5$$

$$\forall i:\mathbb{Z} \bullet MOD5\_of\_int\ i = MOD5vVAL\ (i\ mod\ MOD5vMODULUS)$$

$$MOD5vFIRST \mathrel{\widehat=} MOD5vVAL\ 0$$
$$MOD5vLAST \mathrel{\widehat=} MOD5vVAL\ (MOD5vMODULUS - 1)$$
$$MOD5vPOS \mathrel{\widehat=} MOD5vVAL^{\sim}$$
$$MOD5vSUCC \mathrel{\widehat=} MOD5vPOS \fatsemi succ_{\mathbb{Z}}^{\mathbb{Z}} \fatsemi MOD5\_of\_int$$
$$MOD5vPRED \mathrel{\widehat=} MOD5vSUCC^{\sim}$$

3.2.5 Real Types

A real type (floating point or fixed point) is represented as a subset of the set of real numbers in Z. The category A attributes for the type are defined when the type is declared.

For example, the fixed point type declaration:

$$type\ FIX\ is\ delta\ 0.1\ range\ 5.1\ ..\ 10.0;$$

is translated as

$$FIX \mathrel{\widehat=} 51\ e\ {\sim}1\ ..\ 10\ e\ 0$$
$$FIXvDELTA \mathrel{\widehat=} 1\ e\ {\sim}1$$
$$FIXvFIRST \mathrel{\widehat=} 51\ e\ {\sim}1$$
$$FIXvLAST \mathrel{\widehat=} 1\ e\ 1$$

The floating point type declaration:

$$type\ FLOAT1\ is\ digits\ 7\ range\ -1.0\ ..\ 1.0;$$

is translated as

$$FLOAT1 \mathrel{\widehat=} {\sim}_R\ (1\ e\ 0)\ ..{}_R\ 1\ e\ 0$$
$$FLOAT1vDIGITS \mathrel{\widehat=} 7$$
$$FLOAT1vFIRST \mathrel{\widehat=} {\sim}_R\ 1\ e\ 0$$
$$FLOAT1vLAST \mathrel{\widehat=} 1\ e\ 0$$

A floating point type declaration with no range such as

$$type\ FLOAT2\ is\ digits\ 6;$$

is translated as

$$FLOAT2vFIRST : \mathbb{R}$$
$$FLOAT2vLAST : \mathbb{R}$$
$$FLOAT2 \mathrel{\widehat=} FLOAT2vFIRST\ ..\ FLOAT2vLAST$$
$$FLOAT2vDIGITS \mathrel{\widehat=} 6$$

3.2.6 Subtypes

3.2.6.1 Integer Subtypes

A signed integer subtype is represented as a range of integers. Z global variables are introduced corresponding to the type name and the supported attributes of the type. The attributes are defined to be equal to the corresponding attributes of the base type.

For example, the subtype declaration:

$$subtype\ INTEGER\_SUBTYPE\ is\ INTEGER\_TYPE\ range\ -10\ ..\ 10;$$

is translated as the sequence of abbreviation definitions:

$$
\begin{array}{|l}
INTEGER\_SUBTYPE \mathrel{\widehat=} \sim 10 \mathinner{..} 10 \\
INTEGER\_SUBTYPEvFIRST \mathrel{\widehat=} \sim 10 \\
INTEGER\_SUBTYPEvLAST \mathrel{\widehat=} 10 \\
INTEGER\_SUBTYPEvSUCC \mathrel{\widehat=} INTEGER\_TYPEvSUCC \\
INTEGER\_SUBTYPEvPRED \mathrel{\widehat=} INTEGER\_TYPEvPRED \\
INTEGER\_SUBTYPEvPOS \mathrel{\widehat=} INTEGER\_TYPEvPOS \\
INTEGER\_SUBTYPEvVAL \mathrel{\widehat=} INTEGER\_TYPEvVAL
\end{array}
$$

A modular subtype is represented as a subset of the free type representing the base type. The First and Last attributes are derived from the range in the subtype definition and the other attributes are defined equal to the corresponding attributes of the supertype. For example, the subtype declaration:

subtype ttf is mod5 range 2 .. 4;

is translated as the following sequence of abbreviation definitions:

$$
\begin{array}{l}
TTF \mathrel{\widehat=} MOD5vVAL\ (\!|2 \mathinner{..} 4|\!) \\
TTFvFIRST \mathrel{\widehat=} MOD5vVAL\ 2 \\
TTFvLAST \mathrel{\widehat=} MOD5vVAL\ 4 \\
TTFvMODULUS \mathrel{\widehat=} MOD5vMODULUS \\
TTFvPOS \mathrel{\widehat=} MOD5vPOS \\
TTFvSUCC \mathrel{\widehat=} MOD5vSUCC \\
TTFvPRED \mathrel{\widehat=} MOD5vPRED \\
TTFvVAL \mathrel{\widehat=} MOD5vVAL
\end{array}
$$

3.2.6.2 Array Subtypes

An array subtype, i.e. a subtype with an index constraint, is represented as an appropriate subset of the set of functions which represents the base type. Z global variables are introduced corresponding to the type name and the supported attributes of the type.

For example, the subtype declaration:

subtype ARRAY_SUBTYPE is ARRAY_TYPE3(INDEX_TYPE);

is translated as:

$$
\begin{array}{|l}
ARRAY\_SUBTYPE \mathrel{\widehat=} \{array : ARRAY\_TYPE3 \mid dom\ array = INDEX\_TYPE\} \\
ARRAY\_SUBTYPEvFIRST \mathrel{\widehat=} INDEX\_TYPEvFIRST \\
ARRAY\_SUBTYPEvLAST \mathrel{\widehat=} INDEX\_TYPEvLAST \\
ARRAY\_SUBTYPEvLENGTH \mathrel{\widehat=} \#\ INDEX\_TYPE \\
ARRAY\_SUBTYPEvRANGE \mathrel{\widehat=} INDEX\_TYPE \\
ARRAY\_SUBTYPEvFIRSTv1 \mathrel{\widehat=} INDEX\_TYPEvFIRST \\
ARRAY\_SUBTYPEvLASTv1 \mathrel{\widehat=} INDEX\_TYPEvLAST \\
ARRAY\_SUBTYPEvLENGTHv1 \mathrel{\widehat=} \#\ INDEX\_TYPE \\
ARRAY\_SUBTYPEvRANGEv1 \mathrel{\widehat=} INDEX\_TYPE
\end{array}
$$

For a multi-dimensional array, the domain in the first abbreviation definition is given by an appropriate cartesian product and then abbreviation definitions giving attributes for each dimension are introduced.

If a discrete range in an index constraint is not just a type mark, a Z declaration are introduced for the range as described section 3.1.15.

3.2.6.3 Record Subtypes

A record subtype, i.e., a subtype with a discriminant constraint is translated into a schema definition with the declaration given by the base type and with a predicate expressing the actual discriminant constraints. For example, the record subtype:

subtype line_buffer is buffer(*80*);

is translated as:

Informal Z

$$LINE\_BUFFER \mathrel{\widehat=} [BUFFER \mid SIZE = 80]$$

3.2.6.4 Other Subtypes

Unconstrained subtypes have type attributes introduced for each of the type attributes of the base type of the subtype. Otherwise, subtypes of forms other than those discussed in section 3.2.6.1 and 3.2.6.2 above are represented in a similar way to real types as described in section 3.2.5. If a subtype involves expressions that cannot be handled formally, then a given set paragraph is generated to represent the type (and the predefined operators on the type other than equality will not be handled formally).

3.2.7 Constant Declarations

An Ada constant is represented in Z as a global variable. The form of definition for this global variable depends on whether the defining expression for the constant can be handled formally.

Provided *EXPRESSION* can be handled formally, the constant declaration:

$$CONST\_NAME : constant\ TYPE\_NAME := EXPRESSION$$

is translated as the axiomatic description:

Informal Z

$$CONST\_NAME : TYPE\_NAME$$

$$CONST\_NAME = EXPRESSION$$

If *EXPRESSION* cannot be handled formally, the constant declaration above would be translated as the axiomatic description:

Informal Z

$$CONST\_NAME : TYPE\_NAME$$

A constant declaration defining more than one constant is treated as the equivalent sequence of single-constant declarations.

3.2.8 Function Specifications

An Ada function is represented in Z as a global variable defined via an axiomatic description. The form of the axiomatic description depends on whether or not the function is a formal function (i.e., whether or not the function specification includes a function specification statement).

An informal function (i.e., a function without a function specification statement) is translated into a member of the given set *Informal_Function*. Informal functions have no defining property and are only translated into Z to prevent their use in expressions in the formal parts of a Compliance Notation script. For example, a function specified as:

function INF_FUN (A : in INTEGER) return INTEGER;

is translated into Z as:

Informal Z

$$INF\_FUN : Informal\_Function$$

$$true$$

A formal function is translated into a global variable whose type is derived from the global dependencies, formal parameters and return type of the function and whose defining property is derived from the pre- and post-conditions in the function specification statement. For example, the functions specified as follows:

function FORM_FUN_00 return RTYPE
Ξ [$FORM\_FUN\_00 = 100$]
function FORM_FUN_01 (A : PTYPE) return RTYPE
Ξ [$FORM\_FUN\_01\ A = A * A$]
function FORM_FUN_10 return RTYPE
$\Xi\ G$ [$G > 0$, $FORM\_FUN\_10\ G = G$]
function FORM_FUN_23 (A, B, C : PTYPE) return RTYPE
$\Xi\ G, H$ [$H > 0$, $FORM\_FUN\_23\ (G, H)\ (A, B, C) = G + H + A + B + C$]

are translated into the following axiomatic descriptions.

Informal Z

$$FORM\_FUN\_00 : RTYPE$$

$$true \Rightarrow FORM\_FUN\_00 = 100$$

Informal Z

$$FORM\_FUN\_01 : PTYPE \rightarrow RTYPE$$

$$\forall\ A : PTYPE \bullet true \Rightarrow FORM\_FUN\_01\ A = A * A$$

Informal Z

$$FORM\_FUN\_10 : GTYPE \rightarrow RTYPE$$

$$\forall\ G : GTYPE \bullet G > 0 \Rightarrow FORM\_FUN\_10\ G = G$$

Informal Z

$$FORM\_FUN\_23 : GTYPE \times HTYPE \rightarrow PTYPE \times PTYPE \times PTYPE \rightarrow RTYPE$$

$$\forall\ G : GTYPE;\ H : HTYPE$$
$$\bullet\ \forall\ A,\ B,\ C : PTYPE$$
$$\bullet\ H > 0 \Rightarrow FORM\_FUN\_23\ (G,\ H)\ (A,\ B,\ C) = G + H + A + B + C$$

3.3 VC Generation

A number of language constructs give rise to verification conditions (VCs). These constructs are as follows.

1. A refinement step gives rise to VCs demanding that the sequence of statements on the right-hand side of the refinement symbol correctly implements the specification statement being refined.
2. A subprogram body with a formal subprogram specification gives rise to VCs demanding that the sequence of statements in the body correctly implements the specification statement in the subprogram specification.
3. If a subprogram specification in a package body and the corresponding subprogram specification in the package declaration are both formal, then VCs are generated demanding that the specification statement in the package body correctly implements the one in the package declaration.
4. If a subprogram specification in a subunit and the corresponding subprogram specification in the corresponding body stub are both formal, then VCs are generated demanding that the specification statement in the subunit correctly implements the one in the body stub.
5. A package body which implements a package containing using declarations gives rise to VCs demanding that the package initialisation statements establishes the invariants given in the using declarations.
6. If it cannot be determined whether a range in a type definition is non-empty, a VC is generated asserting that the type is non-empty; this VC always has the form $\vdash\ TYPENAME \neq \varnothing$.
7. In a call of a procedure with a formal parameter whose mode is `out` or `in out` and whose type is a subtype of an unconstrained array type, a VC is generated asserting that the range of the array passed as the actual parameter is the same as the range of the subtype specified in the procedure declaration. That is to say, the VC asserts that the sliding conversion is not needed (see section 3.1.14 above).

The first four forms of VC generation in the above list are concerned with one or more specification statements. A specification statement is a general means for making assertions about program state changes. The specification statement with frame variables A, B, C ...:

$$\Delta\ A,\ B,\ C\ ...\ [PRE\_CONDITION,\ POST\_CONDITION]$$

effectively denotes the set of all pairs (S_1, S_2) where: each S_i is a program state; the pre-condition holds in S_1; the post-condition holds in S_2; and S_1 and S_2 only differ with respect to the values

of the frame variables A, B, C ... (but see also the remarks about replacement steps which declare new local variables in section 2.15, and the discussion of domain conditions in section 3.4). In the post-condition the initial values of the frame variables may be referred to by adding a subscript *0* to the name. For example, the specification statement:

$$\Delta\ A\ [true,\ A = A_0 + 1]$$

asserts that the integer variable A should be increased by *1*.

The Compliance Notation VC generation algorithm is based on the notion of *partial correctness*. If all VCs of a refinement step can be proved, then at run-time, the refining code either meets its specification or fails to terminate or raises an exception. The soundness of the algorithm requires that the script conform to the rules in chapter 2 above and that the Ada program generated from the script conforms to the rules of the ALRM.

In generating the VCs that give the conditions under which a sequence of statements refine a specification statement, ertain statement forms must be restricted. The statement forms in question are:

- specification statements that refer to initial variables
- logical constant statements
- procedure calls, where the specification statement for the procedure refers to initial variables
- for-loops whose bounds are not static expressions

Such statements are restricted to appearing in positions where no code with side effects could be executed before them in the sequence of statements. This condition is formulated syntactically as follows: the above four statement forms may only appear in positions in the sequence of statements which are *suitable* in the following sense:

- the first statement in a sequence is suitable;
- the first statement in a branch of a case statement which is itself in a suitable position is also suitable;
- the first statement in a branch of an if statement which is itself in a suitable position is also suitable.

The form of VCs generated for refinement steps involving the various forms of statement are described, by means of symbolic examples, in sections 3.3.1 to 3.3.14 below. Sections 3.3.15 to 3.3.19 below give symbolic examples for the other kinds of VC generation.

3.3.1 Null Statement

Compliance Notation Script: Null Statement

Compliance Notation

procedure vc_null
is
  X : *INTEGER*;
begin
 Δ X [*PRE* X, *POST* X] (*111*)
end vc_null;

Compliance Notation

(*111*) $\sqsubseteq$ *NULL*;

Generated VCs

vc111_1 $\forall$ X : *INTEGER* | *PRE* X $\bullet$ *POST* X

Notes

A null statement can only achieve its specification if the pre-condition already implies the post-condition so that no action needs to be taken.

3.3.2 Assignment Statement

Compliance Notation Script: Assignment: Case 1

Compliance Notation

procedure vc_assignment1
is
  LHS, *RHS* : *INTEGER*;
begin
 Δ LHS [*PRE* (*LHS*, *RHS*), *POST* (*LHS*, *RHS*, LHS_0)] (*211*)
end vc_assignment1;

Compliance Notation

(*211*) ⊑ *LHS* := *RHS*;

Generated VCs

vc211_1 ∀ *LHS*, *RHS* : *INTEGER*
| *PRE* (*LHS*, *RHS*)
• *POST* (*RHS*, *RHS*, *LHS*)

Compliance Notation Script: Assignment: Case 2

Compliance Notation

procedure vc_assignment2
is
  type DAY is range 1 .. 31;
  type MONTH is range 1 .. 12;
  type YEAR is range −10000 .. +10000;
  type DATE is record D : DAY; *M : MONTH*; *Y : YEAR*; *end record*;
  LHS : DATE;
  RHS : DAY;
begin
 Δ *LHS* [*PRE* (*LHS*, *RHS*), *POST* (*LHS*, *RHS*, LHS_0)] (*221*)
end vc_assignment2;

Compliance Notation

(*221*) $\sqsubseteq$ *LHS.D* := *RHS*;

Generated VCs

vc221_1

$\forall$ *LHS* : *DATE*; *RHS* : *DAY*
 | *PRE* (*LHS*, *RHS*)
 • *POST* (*DATEuD* (*LHS*, *RHS*), *RHS*, *LHS*)

Notes

Assignment to a record component results in VC containing a call to the record update function for that component (see section 3.2.3).

Compliance Notation Script: Assignment: Case 3

Compliance Notation

procedure vc_assignment3
is
  type TRIAD is (ONE, TWO, THREE);
  type ARRAY3 is array (TRIAD) of INTEGER;
  LHS : ARRAY3;
  RHS : INTEGER;
  INDEX : TRIAD;
begin
 Δ *LHS* [*PRE (LHS, RHS), POST (LHS, RHS,* LHS_0*)*] (*231*)
end vc_assignment3;

Compliance Notation

(*231*) $\sqsubseteq$ *LHS(INDEX)* := *RHS;*

Generated VCs

vc231_1

$$\forall\ LHS : ARRAY3;\ RHS : INTEGER;\ INDEX : TRIAD \mid PRE\ (LHS, RHS) \bullet POST\ (LHS \oplus \{INDEX \mapsto RHS\}, RHS, LHS)$$

Notes

Assignment to an array element results in a singleton override on the Z function representing the array in the VC.

Compliance Notation Script: Assignment: Case 4

Compliance Notation

```
procedure vc_assignment4
is
   type REC_XY is record X : INTEGER; Y : INTEGER; end record;
   type DYAD is (EINS, ZWEI);
   type ARRAY2_REC is array (DYAD) of REC_XY;
   LHS : ARRAY2_REC;
   RHS : INTEGER;
   INDEX : DYAD;
begin
 Δ LHS [ PRE (LHS, RHS), POST (LHS, RHS, LHS_0) ]                    (241)
end vc_assignment4;
```

Compliance Notation

$(241) \sqsubseteq LHS(INDEX).X := RHS;$

Generated VCs

vc241_1

$$\forall\ LHS : ARRAY2\_REC;\ INDEX : DYAD;\ RHS : INTEGER$$
$$\mid PRE\ (LHS,\ RHS)$$
$$\bullet\ POST\ (LHS \oplus \{INDEX \mapsto REC\_XYuX\ (LHS\ INDEX,\ RHS)\},$$
$$RHS,$$
$$LHS)$$

Notes

Assignments with more complex right hand-sides produce a mixture of record update operations and singleton overrides in the VC.

Compliance Notation Script: Assignment: Case 5

Compliance Notation

```
procedure vc_assignment5
is
   type DYAD is (EINS, ZWEI);
   type ARRAY2_INT is array (DYAD) of INTEGER;
   type REC_AB is record A : ARRAY2_INT; B : ARRAY2_INT; end record;
   LHS : REC_AB;
   RHS : INTEGER;
   INDEX : DYAD;
begin
 Δ LHS [ PRE (LHS, RHS), POST (LHS, RHS, LHS_0) ]                    (251)
end vc_assignment5;
```

Compliance Notation

$(251) \sqsubseteq LHS.A(INDEX) := RHS;$

Generated VCs

vc251_1

$$\begin{array}{l} \forall\ INDEX : DYAD;\ RHS : INTEGER;\ LHS : REC\_AB \\ \quad \mid PRE\ (LHS,\ RHS) \\ \quad \bullet\ POST\ (REC\_ABuA\ (LHS,\ LHS.A \oplus \{INDEX \mapsto RHS\}), \\ \qquad RHS, \\ \qquad LHS) \end{array}$$

Notes

Assignments with more complex right hand-sides produce a mixture of binding displays and singleton overrides in the VC.

3.3.3 Specification Statement

Compliance Notation Script: Specification Statement

Compliance Notation

procedure vc_specification
is
  X : *INTEGER*;
begin
 Δ X [*PRE* X, *POST* (X, X_0)] (*311*)
end vc_specification;

Compliance Notation

(*311*) $\sqsubseteq$ Δ X [*PRE1* X, *POST1* (X, X_0)] (*312*)

Generated VCs

vc311_1 $\forall$ X : *INTEGER* | *PRE* X • *PRE1* X

vc311_2 $\forall$ X, X_0 : *INTEGER*
| *PRE* X_0 $\wedge$ *POST1* (X, X_0)
• *POST* (X, X_0)

Notes

Note that *PRE* rather than *PRE1* appears in the second VC. Since the first VC requires *PRE* to be at least as strong *PRE1* this is no loss, and could be a gain.

3.3.4 Semicolon

Compliance Notation Script: Sequence of Statements

Compliance Notation

procedure vc_sequence_of_statements
is
  X : *INTEGER*;
begin
 Δ X [*PRE* X, *POST* (X, X_0)] (*411*)
end vc_sequence_of_statements;

Compliance Notation

(*411*) $\sqsubseteq$ Δ X [*PRE1* X, *POST1* (X, X_0)] (*412*)
Δ X [*PRE2* X, *POST2* X] (*413*)

Generated VCs

vc411_1 $\forall$ X : *INTEGER* | *PRE* X • *PRE1* X
vc411_2 $\forall$ X, X_0 : *INTEGER* | *PRE* X_0 $\wedge$ *POST1* (X, X_0) • *PRE2* X
vc411_3 $\forall$ X, X_0 : *INTEGER* | *PRE* X_0 $\wedge$ *POST2* X • *POST* (X, X_0)

Notes

It is not permitted to refer to initial values of variables in the second statement.

The intermediate post-condition *POST1* only as appears as part of the assumptions under which the second VC requires us to prove *PRE2*. If *POST1* or some part of *POST1* is needed in *POST*, then it must be included in *POST2* and *PRE2*.

3.3.5 If Statement

Compliance Notation Script: If Statement

Compliance Notation

```
with FUNS;
 procedure vc_if
 is
    X : INTEGER;
 begin
  Δ X [ PRE X, POST X ]          (511)
 end vc_if;
```

Compliance Notation

```
(511) ⊑      if FUNS.TEST(X)
             then   Δ X [ PRE1 X, POST1 X ] (512)
             else   Δ X [ PRE2 X, POST2 X ] (513)
             end if;
```

Generated VCs

vc511_1 $\forall\ X : INTEGER \mid PRE\ X \wedge FUNSoTEST\ X = TRUE \bullet PRE1\ X$

vc511_2 $\forall\ X : INTEGER \mid PRE\ X \wedge FUNSoTEST\ X = FALSE \bullet PRE2\ X$

vc511_3 $\forall\ X, X_0 : INTEGER \mid PRE\ X_0 \wedge POST1\ X \bullet POST\ X$

vc511_4 $\forall\ X, X_0 : INTEGER \mid PRE\ X_0 \wedge POST2\ X \bullet POST\ X$

Notes

Initial variables are not allowed in the arms of the if statement.

If the *else* part is omitted then *POST* appears in the second VC in place of *PRE2* and the fourth VC does not appear.

3.3.6 Case Statement

Compliance Notation Script: Case Statement

Compliance Notation

```
with FUNS;
 procedure vc_case
 is
    type T_CASE is range 1 .. 9;
    X : T_CASE;
 begin
  Δ X [ PRE X, POST X ]          (611)
 end vc_case;
```

Compliance Notation

```
(611) ⊑      case T_CASE(FUNS.COMPUTE(X)) is
             when 1 =>                 Δ X [ PRE1 X, POST1 X ] (612)
             when 5 .. 7 =>            Δ X [ PRE2 X, POST2 X ] (613)
             when 2 .. 4 | 8 | 9  =>   Δ X [ PRE3 X, POST3 X ] (614)
             end case;
```

Generated VCs

vc611_1 $\forall\ X : T\_CASE \mid PRE\ X \wedge FUNSoCOMPUTE\ X \in \{1\} \bullet PRE1\ X$

vc611_2 $\forall\ X : T\_CASE \mid PRE\ X \wedge FUNSoCOMPUTE\ X \in 5\ ..\ 7 \bullet PRE2\ X$

vc611_3 $\forall\ X : T\_CASE \mid PRE\ X \wedge FUNSoCOMPUTE\ X \in (2\ ..\ 4) \cup \{8, 9\} \bullet PRE3\ X$

vc611_4 $\forall\ X, X_0 : T\_CASE \mid PRE\ X_0 \wedge POST1\ X \bullet POST\ X$

vc611_5 $\forall\ X, X_0 : T\_CASE \mid PRE\ X_0 \wedge POST2\ X \bullet POST\ X$

vc611_6 $\forall\ X, X_0 : T\_CASE \mid PRE\ X_0 \wedge POST3\ X \bullet POST\ X$

Notes

Initial variables are not allowed in the specification statements in the arms of the case statement.

If the fact that the case expression is in one of the ranges is needed in the corresponding arm, then that needs to be stated in the pre-condition for that arm. (E.g. if we need to know that the expression is *1* in the first arm here, we need to say that in *PRE1*).

3.3.7 Undecorated Loop Statement

Compliance Notation Script: Undecorated Loop Statement

Compliance Notation

```
procedure vc_undecorated_loop
is
   X : INTEGER;
begin
 Δ X [ PRE X, POST X ]          (711)
end vc_undecorated_loop;
```

Compliance Notation

```
(711) ⊑      $till ⟦ TILL X ⟧
             loop
                Δ X [ PRE1 X, POST1 X ] (712)
             end loop;
```

Generated VCs

vc711_1 $\forall\ X : INTEGER \mid PRE\ X \bullet PRE1\ X$

vc711_2 $\forall\ X, X_0 : INTEGER \mid PRE\ X_0 \wedge POST1\ X \bullet PRE1\ X$

vc711_3 $\forall\ X, X_0 : INTEGER \mid PRE\ X_0 \wedge TILL\ X \bullet POST\ X$

Notes

If the till predicate is omitted, then the last VC is not produced (and there is no formally handled way of leaving the loop, since an *exit* statement is not allowed in a loop with no till predicate).

3.3.8 While Loop Statement

Compliance Notation Script: While Loop Statement

Compliance Notation

with FUNS;
procedure vc_while_loop
is
  X : *INTEGER*;
begin
 Δ X [*PRE* X, *POST* X] (*811*)
end vc_while_loop;

Compliance Notation

(*811*) $\sqsubseteq$ *while FUNS.TEST*(X) $\$$*till* $[\![$ *TILL* X $]\!]$
loop
  Δ X [*PRE1* X, *POST1* X] (*812*)
end loop;

Generated VCs

vc811_1 $\forall$ X : *INTEGER* | *PRE* X $\wedge$ *FUNSoTEST* X = *TRUE* $\bullet$ *PRE1* X

vc811_2 $\forall$ X : *INTEGER* | *PRE* X $\wedge$ *FUNSoTEST* X = *FALSE* $\bullet$ *POST* X

vc811_3 $\forall$ X, X_0 : *INTEGER*
| *PRE* X_0 $\wedge$ *POST1* X $\wedge$ *FUNSoTEST* X = *TRUE*
$\bullet$ *PRE1* X

vc811_4 $\forall$ X, X_0 : *INTEGER*
| *PRE* X_0 $\wedge$ *POST1* X $\wedge$ *FUNSoTEST* X = *FALSE*
$\bullet$ *POST* X

vc811_5 $\forall$ X, X_0 : *INTEGER* | *PRE* X_0 $\wedge$ *TILL* X $\bullet$ *POST* X

Notes

If the till predicate is omitted, then the last VC is not produced.

3.3.9 For Loop Statement

Compliance Notation Script: For Loop Statement: Case 1

Compliance Notation

procedure vc_for_loop1
is
  X : *INTEGER*;
begin
 Δ X [*PRE* X, *POST* X] (*911*)
end vc_for_loop1;

Compliance Notation

(*911*) $\sqsubseteq$ *for I in INTEGER range 1 .. 10 $till* $[\![$ *TILL* (X, I) $]\!]$
loop
  Δ X [*PRE1* (X, I) , *POST1* (X, I)] (*912*)
end loop;

Generated VCs

vc911_1 $\forall$ X : *INTEGER* | *PRE* X $\wedge$ $1 \leq 10$ $\bullet$ *PRE1* (X, 1)

vc911_2 $\forall$ X : *INTEGER* | *PRE* X $\wedge$ $1 > 10$ $\bullet$ *POST* X

vc911_3 $\forall$ I, X, X_0 : *INTEGER*
| *PRE* X_0 $\wedge$ $I \in 1$ .. 10 $\wedge$ $I \neq 10$ $\wedge$ *POST1* (X, I)
$\bullet$ *PRE1* (X, $I + 1$)

vc911_4 $\forall$ X, X_0 : *INTEGER* | *PRE* X_0 $\wedge$ *POST1* (X, 10) $\bullet$ *POST* X

vc911_5 $\forall$ I, X, X_0 : *INTEGER*
| *PRE* X_0 $\wedge$ $I \in 1$ .. 10 $\wedge$ *TILL* (X, I)
$\bullet$ *POST* X

Notes

If the till predicate is omitted, then the last VC is not produced.

If the type mark in the loop parameter specification is omitted it will be taken to be *universal_discrete* in any VCs involving the loop control variable.

Compliance Notation Script: For Loop Statement: Case 2

Compliance Notation

```
procedure vc_for_loop2
is
   type T_FOR_LOOP2 is range 1 .. 10;
   X : INTEGER;
begin
 Δ X [ PRE X, POST X ]          (921)
end vc_for_loop2;
```

Compliance Notation

```
(921) ⊑      for I in T_FOR_LOOP2 $till ⟦ TILL (X, I) ⟧
             loop
               Δ X [ PRE1 (X, I), POST1 (X, I) ] (922)
             end loop;
```

Generated VCs

vc921_1
$$\forall\ X : INTEGER \mid PRE\ X \wedge T\_FOR\_LOOP2vFIRST \leq T\_FOR\_LOOP2vLAST \bullet PRE1\ (X,\ T\_FOR\_LOOP2vFIRST)$$

vc921_2
$$\forall\ X : INTEGER \mid PRE\ X \wedge T\_FOR\_LOOP2vFIRST > T\_FOR\_LOOP2vLAST \bullet POST\ X$$

vc921_3
$$\forall\ X,\ X_0 : INTEGER;\ I : T\_FOR\_LOOP2 \mid PRE\ X_0 \wedge I \neq T\_FOR\_LOOP2vLAST \wedge POST1\ (X,\ I) \bullet PRE1\ (X,\ I + 1)$$

vc921_4
$$\forall\ X,\ X_0 : INTEGER \mid PRE\ X_0 \wedge POST1\ (X,\ T\_FOR\_LOOP2vLAST) \bullet POST\ X$$

vc921_5
$$\forall\ X,\ X_0 : INTEGER;\ I : T\_FOR\_LOOP2 \mid PRE\ X_0 \wedge TILL\ (X,\ I) \bullet POST\ X$$

Notes

If the till predicate is omitted, then the last VC is not produced.

If the type mark in the loop parameter specification is omitted it will be taken to be *universal_discrete* in any VCs involving the loop control variable.

3.3.10 Block Statement

Compliance Notation Script: Block Statement

Compliance Notation

procedure vc_block
is
  X : *INTEGER*;
begin
  $\Delta\ X$ [*PRE* X, *POST* X] (*1411*)
end vc_block;

Compliance Notation

(*1411*) $\sqsubseteq$
blk:
  declare
    Y : *INTEGER* := *0*;
  begin
    $\Delta\ X,\ Y$ [*PRE1*($X,\ Y$), *POST1*($X,\ Y$)]
  end blk;

Generated VCs

vc1411_1 $\forall\ X,\ Y : INTEGER \mid PRE\ X \wedge Y = 0 \bullet PRE1\ (X,\ Y)$

vc1411_2 $\forall\ X,\ X_0,\ Y,\ Y_0 : INTEGER \mid (PRE\ X_0 \wedge Y_0 = 0) \wedge POST1\ (X,\ Y) \bullet POST\ X$

Notes

Equations derived from any initial values in variable declarations in the block are conjoined with the pre-condition, *PRE X*.

3.3.11 Exit Statement

Compliance Notation Script: Exit Statement: Case 1

Compliance Notation

with FUNS;
procedure vc_exit1
is
  X : *INTEGER*;
begin
 Δ X [*PRE X*, *POST X*] (*1011*)
end vc_exit1;

Compliance Notation

(*1011*) ⊑ *for I in INTEGER range 1 .. 10 $till* ⟦*TILL X*⟧
loop
  Δ X [*PRE1* (*X*, *I*), *POST1* (*X*, *I*)] (*1012*)
end loop;

Compliance Notation

(*1012*) ⊑ *Δ X*[*PRE2 X*, *POST2 X*]
exit when FUNS.TEST(*X*);

Generated VCs

vc1012_1 ∀ *I*, *X* : *INTEGER* | *PRE1* (*X*, *I*) • *PRE2 X*

vc1012_2 ∀ *I*, *X*, X_0 : *INTEGER*
| *PRE1* (X_0, *I*) ∧ *POST2 X* ∧ *FUNSoTEST X* = *TRUE*
• *TILL X*

vc1012_3 ∀ *I*, *X*, X_0 : *INTEGER*
| *PRE1* (X_0, *I*) ∧ *POST2 X* ∧ *FUNSoTEST X* = *FALSE*
• *POST1* (*X*, *I*)

Notes

For clarity, the VCs from the first refinement step are not shown above.

The till predicate must not be omitted on the loop being exited. For an exit with a loop name, the till predicate is taken from the corresponding enclosing named loop.

If the statement before the exit is omitted the first VC does not appear.

Compliance Notation Script: Exit Statement: Case 2

Compliance Notation

```
with FUNS;
procedure vc_exit2
is
   X : INTEGER;
begin
 Δ X [ PRE X, POST X ]          (1021)
end vc_exit2;
```

Compliance Notation

```
(1021) ⊑    for I in INTEGER range 1 .. 10 $till ⟦TILL X⟧
            loop
               Δ X [ PRE1 (X, I), POST1 (X, I) ]          (1022)
            end loop;
```

Compliance Notation

```
(1022) ⊑    if FUNS.TEST(X)
            then   Δ X[PRE2 X, POST2 X]
                   exit;
            end if;
```

Generated VCs

vc1022_1

$$\forall\ I, X : INTEGER \mid PRE1\ (X, I) \wedge FUNSoTEST\ X = TRUE \bullet PRE2\ X$$

vc1022_2

$$\forall\ I, X : INTEGER \mid PRE1\ (X, I) \wedge FUNSoTEST\ X = FALSE \bullet POST1\ (X, I)$$

vc1022_3

$$\forall\ I, X, X_0 : INTEGER \mid PRE1\ (X_0, I) \wedge POST2\ X \bullet TILL\ X$$

Notes

For clarity, the VCs from the first refinement step have been omitted.

The till predicate must not be omitted.

If the statement before the exit is omitted the first VC does not appear.

3.3.12 Return Statement

Compliance Notation Script: Return Statement: Case 1

Compliance Notation

```
with FUNS;
 function vc_return1 (A : INTEGER) return INTEGER
 Ξ [PRE A, POST (A, VC_RETURN1(A))]
 is
 begin
  return FUNS.COMPUTE(A);
 end vc_return1;
```

Generated VCs

vcVC_RETURN1_1

$$\forall\ A : INTEGER;\ VC\_RETURN1 : INTEGER \rightarrow INTEGER$$
$$\mid PRE\ A \wedge VC\_RETURN1\ A = FUNSoCOMPUTE\ A$$
$$\bullet\ POST\ (A,\ VC\_RETURN1\ A)$$

For a return from a procedure, i.e., without an expression, the equation defining the returned value is omitted from the VC.

Compliance Notation Script: Return Statement: Case 2

Compliance Notation

```
with FUNS;
function vc_return2 (A : INTEGER) return INTEGER
Ξ [PRE A, POST (A, VC_RETURN2(A))]
is
begin
   Δ  [ PRE1 A , false ]   (1121)
end vc_return2;
```

Compliance Notation

$(1121) \sqsubseteq$ *return FUNS.COMPUTE*(A);

Generated VCs

vcVC_RETURN2_1

$\forall\ A : INTEGER \mid PRE\ A \bullet PRE1\ A$

vcVC_RETURN2_2

$\forall\ A : INTEGER;\ VC\_RETURN2 : INTEGER \rightarrow INTEGER$
$\mid PRE\ A \wedge false$
$\bullet\ POST\ (A,\ VC\_RETURN2\ A)$

vc1121_1

$\forall\ A : INTEGER;\ VC\_RETURN2 : INTEGER \rightarrow INTEGER$
$\mid PRE1\ A \wedge VC\_RETURN2\ A = FUNSoCOMPUTE\ A$
$\bullet\ POST\ (A,\ VC\_RETURN2\ A)$

Notes

The post-condition for the body of a function can be *false* since control never returns from the body (instead, control returns to the caller of the function when the return statement is executed). By using *false* we ensure that the second VC is a tautology.

For a return from a procedure, i.e., without an expression, the equation defining the returned value is omitted from the last VC. A post-condition of *false* is also appropriate for a procedure if all paths through the body of the procedure end in a return statement.

3.3.13 Procedure Call Statement

Compliance Notation Script: Procedure Call Statement

Compliance Notation

procedure vc_procedure_call
is
  procedure PROC (*A* : *in out INTEGER*)
   Δ [*PRE1 A*, *PRE1* (*A*$_0$, *A*)]
  is separate;
  X : *INTEGER*;
begin
 Δ X [*PRE X*, *POST X*] (*1211*)
end vc_procedure_call;

Compliance Notation

(*1211*) ⊑ *PROC*(*X*);

Generated VCs

vc1211_1 ∀ *X* : *INTEGER* | *PRE X* • *PRE1 X*
vc1211_2 ∀ *X*, *X*$_0$: *INTEGER* | *PRE X* ∧ *PRE1* (*X*$_0$, *X*) • *POST X*

3.3.14 Logical Constant Statement

Compliance Notation Script: Logical Constant Statement: Case 1

Compliance Notation

procedure vc_logical_constant1
is
  $X : INTEGER;$
begin
 $\Delta\ X\ [\ PRE\ X,\ POST\ X\ ]$ *(1311)*
end vc_logical_constant1;

Compliance Notation

$(1311) \sqsubseteq$ $\$CON\ X\_INIT : \mathbb{Z}\bullet$
$\Delta\ X\ [X\_INIT = X \land PRE1\ X,\ POST1(X,\ X\_INIT)]$ *(1312)*

Compliance Notation

$(1312) \sqsubseteq$ $\Delta\ X\ [\ PRE2\ X,\ POST2(X,\ X\_INIT)]$ *(1313)*
$\Delta\ X\ [\ PRE3(X,\ X\_INIT),\ POST3(X,\ X\_INIT)]$ *(1314)*

Generated VCs

vc1311_1 $\forall\ X : INTEGER;\ X\_INIT : \mathbb{Z} \mid PRE\ X \land X\_INIT = X \bullet PRE1\ X$

vc1311_2 $\forall\ X : INTEGER \mid PRE\ X \bullet X \in \mathbb{Z}$

vc1311_3 $\forall\ X,\ X_0 : INTEGER;\ X\_INIT : \mathbb{Z}$
$\mid PRE\ X_0 \land X\_INIT = X_0 \land POST1\ (X,\ X\_INIT)$
$\bullet\ POST\ X$

vc1312_1 $\forall\ X : INTEGER;\ X\_INIT : \mathbb{Z}$
$\mid X\_INIT = X \land PRE1\ X$
$\bullet\ PRE2\ X$

vc1312_2 $\forall\ X,\ X_0 : INTEGER;\ X\_INIT : \mathbb{Z}$
$\mid (X\_INIT = X_0 \land PRE1\ X_0) \land POST2\ (X,\ X\_INIT)$
$\bullet\ PRE3\ (X,\ X\_INIT)$

vc1312_3 $\forall\ X,\ X_0 : INTEGER;\ X\_INIT : \mathbb{Z}$
$\mid (X\_INIT = X_0 \land PRE1\ X_0) \land POST3\ (X,\ X\_INIT)$
$\bullet\ POST1\ (X,\ X\_INIT)$

Notes

A refinement of the logical constant statement into a sequence of statements i s shown to illustrate how the statement may be used to capture an initial value.

Compliance Notation Script: Logical Constant Statement: Case 2

Compliance Notation

procedure vc_logical_constant2
is
  $X : INTEGER;$
begin
 $\Delta\ X\ [\ PRE\ X,\ POST\ X\ ]$ *(1321)*
end vc_logical_constant2;

Compliance Notation

$(1321) \sqsubseteq$ $\$CON\ Y, Z : \mathbb{Z} \bullet$
$\Delta\ X\ [Y = X+X \land Z = Y*Y \land PRE1\ X,\ POST1(X,\ Y,\ Z)]$ *(1312)*

Generated VCs

vc1321_1 $\forall\ X : INTEGER;\ Y, Z : \mathbb{Z}$
$\mid PRE\ X \land Y = X + X \land Z = Y * Y$
$\bullet\ PRE1\ X$

vc1321_2 $\forall\ X : INTEGER \mid PRE\ X \bullet X + X \in \mathbb{Z}$

vc1321_3 $\forall\ X : INTEGER;\ Y : \mathbb{Z} \mid PRE\ X \land Y = X + X \bullet Y * Y \in \mathbb{Z}$

vc1321_4 $\forall\ X, X_0 : INTEGER;\ Y, Z : \mathbb{Z}$
$\mid PRE\ X_0$
$\land\ Y = X_0 + X_0$
$\land\ Z = Y * Y$
$\land\ POST1\ (X,\ Y,\ Z)$
$\bullet\ POST\ X$

3.3.15 Subprogram Body

Compliance Notation Script: Subprogram Body: Case 1 (Procedure)

Compliance Notation

procedure vc_procedure_body(*X* : *in out INTEGER*)
Δ X[*PRE X*, *POST X*]
is
  Y : *INTEGER* := *0*;
begin
 Δ X [*PRE1* (*X*, *Y*), *POST1* (*X*, *Y*)]
end vc_procedure_body;

Generated VCs

vcVC_PROCEDURE_BODY_1

$$\forall\ X,\ Y : INTEGER \mid PRE\ X \land Y = 0 \bullet PRE1\ (X,\ Y)$$

vcVC_PROCEDURE_BODY_2

$$\forall\ X,\ X_0,\ Y : INTEGER \mid (PRE\ X_0 \land Y = 0) \land POST1\ (X,\ Y) \bullet POST\ X$$

Notes

This shows the VCs generated for a formal procedure body. The VCs assert that the sequence of statements in the body (given in the example as a single specification statement) refines the specification statement in the procedure specification. Equations derived from any initial values in variable declarations in the subprogram are conjoined with the pre-condition, *PRE X*.

If the body contains one or more program statements then the VCs generated are similar to those generated if the statements were used to implement the specification statement in a refinement step as described in sections 3.3.1 to 3.3.14 above.

Compliance Notation Script: Subprogram Body: Case 2 (Function)

Compliance Notation

function vc_function_body(X : *in INTEGER*) *return INTEGER*
Ξ [*PRE* X, *POST* (X, *VC_FUNCTION_BODY* X)]
is
  Y : *INTEGER*;
begin
 Δ Y [*PRE1* (X, Y), *POST1* (X, Y)]
end vc_function_body;

Generated VCs

vcVC_FUNCTION_BODY_1

$$\forall\ X,\ Y : INTEGER \mid PRE\ X \bullet PRE1\ (X,\ Y)$$

vcVC_FUNCTION_BODY_2

$$\forall\ X,\ Y : INTEGER;\ VC\_FUNCTION\_BODY : INTEGER \rightarrow INTEGER \mid PRE\ X \land POST1\ (X,\ Y) \bullet POST\ (X,\ VC\_FUNCTION\_BODY\ X)$$

Notes

This shows the VCs generated for a formal function body. The VCs assert that the sequence of statements in the body (given in the example as a single specification statement) refines the specification statement in the function specification.

Here, and in further refinements of the sequence of statements, the function itself appears under a universal quantifier, so that the translation of the function as a Z global variable is not available inside the formal development of the function itself.

3.3.16 Subprogram in Package Body

Compliance Notation Script: Subprogram in Package Body: Case 1

Compliance Notation

```
package vc_pack_body1
is
  X : INTEGER;
  procedure P
  Δ VC_PACK_BODY1oX[ PRE VC_PACK_BODY1oX, POST VC_PACK_BODY1oX ] ;
end vc_pack_body1;
```

Compliance Notation

```
package body vc_pack_body1
is
  procedure P
  Δ VC_PACK_BODY1oX[ PRE1 VC_PACK_BODY1oX, POST1 VC_PACK_BODY1oX ]
  is
  begin
  Δ VC_PACK_BODY1oX[ PRE2 VC_PACK_BODY1oX, POST2 VC_PACK_BODY1oX]
  end P;
end vc_pack_body1;
```

Generated VCs

vcVC_PACK_BODY1_1

$$\forall\ VC\_PACK\_BODY1oX : INTEGER \mid PRE\ VC\_PACK\_BODY1oX \bullet PRE1\ VC\_PACK\_BODY1oX$$

vcVC_PACK_BODY1_2

$$\forall\ VC\_PACK\_BODY1oX, VC\_PACK\_BODY1oX_0 : INTEGER \mid PRE\ VC\_PACK\_BODY1oX_0 \wedge POST1\ VC\_PACK\_BODY1oX \bullet POST\ VC\_PACK\_BODY1oX$$

Notes

The subprogram body in this example cause 2 additional VCs to be generated that are not shown here. See sections 3.3.15 for these VCs.

Compliance Notation Script: Subprogram in Package Body: Case 2

Compliance Notation

package vc_pack_body2
is
  $*auxiliary* $A : \mathbb{N}$;
  procedure P
  Δ *A* [*APRE A*, *APOST A*] ;
end vc_pack_body2;

Compliance Notation

package body vc_pack_body2
is
  $*using C* : *INTEGER*; $*implement A* $*by INV*(*A*, *C*);
  procedure P
  Δ *C* [*CPRE C*, *CPOST C*]
  is begin
   Δ *C* [*SPRE C*, *SPOST C*] (*1*)
  end P;
begin
  Δ *C*[*IPRE C*, *IPOST C*] (*2*)
end vc_pack_body2;

Generated VCs

vcVC_PACK_BODY2_1

$$\forall\ C : INTEGER \mid \exists\ A : \mathbb{N} \bullet APRE\ A \wedge INV\ (A,\ C) \bullet CPRE\ C$$

vcVC_PACK_BODY2_2

$$\begin{array}{l}\forall\ C,\ C_0 : INTEGER \\ \quad \mid (\exists\ A : \mathbb{N} \bullet APRE\ A \wedge INV\ (A,\ C_0)) \wedge CPOST\ C \\ \quad \bullet \forall\ A_0 : \mathbb{N} \\ \quad\quad \bullet\ APRE\ A_0 \wedge INV\ (A_0,\ C_0) \\ \quad\quad\quad \Rightarrow (\exists\ A : \mathbb{N} \bullet APOST\ A \wedge INV\ (A,\ C))\end{array}$$

Notes

This example shows a **data refinement**: the situation in which one or more auxiliary variables are used in the package specification to model the state of the package. If the using clause is omitted, then the invariant $INV(A, C)$ does not appear in the VCs (which will typically not then be provable).

The subprogram body and the package initialisation in this example cause additional VCs to be generated that are not shown here. See sections 3.3.15 and 3.3.18 for these VCs.

Compliance Notation Script: Subprogram in Package Body: Case 3

Compliance Notation

package vc_pack_body3
is
$*auxiliary* $A : \mathbb{N}$;
procedure $Q(X : in\ out\ INTEGER)$
$\Delta\ X\ \Xi\ A[APRE\ (A,\ X),\ APOST(A,\ X,\ X_0)]$;
end vc_pack_body3;

Compliance Notation

package body vc_pack_body3
is
$*using*\ C : INTEGER$; $*implement*\ A\ *by*\ INV(A,\ C)$;
procedure $Q(X : in\ out\ INTEGER)$
$\Delta\ X,\ C\ [CPRE\ C,\ CPOST(C,\ X,\ X_0)]$
is begin
$\Delta\ X,\ C\ [SPRE\ C,\ SPOST(C,\ X,\ X_0)]$ *(1)*
end Q;
begin
$\Delta\ C[IPRE\ C,\ IPOST\ C]$ *(2)*
end vc_pack_body3;

Generated VCs

vcVC_PACK_BODY3_1

$$\forall\ C,\ X : INTEGER \mid \exists\ A : \mathbb{N} \bullet APRE\ (A,\ X) \land INV\ (A,\ C) \bullet CPRE\ C$$

vcVC_PACK_BODY3_2

$$\forall\ C,\ C_0,\ X,\ X_0 : INTEGER \mid (\exists\ A : \mathbb{N} \bullet APRE\ (A,\ X_0) \land INV\ (A,\ C_0)) \land CPOST\ (C,\ X,\ X_0) \bullet \forall\ A_0 : \mathbb{N} \bullet APRE\ (A_0,\ X_0) \land INV\ (A_0,\ C_0) \Rightarrow (\exists\ A : \mathbb{N} \bullet (APOST\ (A,\ X,\ X_0) \land A = A_0) \land INV\ (A,\ C))$$

Notes

This case shows a data refinement in which a procedure specification does not refer to the auxiliary variables in its frame. In this case, equations requiring that the procedure does not change the auxiliary variable are introduced.

The subprogram body and the package initialisation in this example cause additional VCs to be generated that are not shown here. See sections 3.3.15 and 3.3.18 for these VCs.

3.3.17 Subunit

Compliance Notation Script: Subunit

Compliance Notation

```
procedure vc_subunit
is
  X : INTEGER;
  procedure P
  Δ X[ PRE X, POST X ]
  is separate;
begin
  null;
end vc_subunit;
```

Compliance Notation

```
separate (vc_subunit)
procedure P
  Δ X[ PRE1 X, POST1 X ]
  is
  begin
  Δ X[ PRE2 X, POST2 X]
  end P;
```

Generated VCs

vcVC_SUBUNIToP_1

$$\forall\ X : INTEGER \mid PRE\ X \bullet PRE1\ X$$

vcVC_SUBUNIToP_2

$$\forall\ X, X_0 : INTEGER \mid PRE\ X_0 \land POST1\ X \bullet POST\ X$$

vcVC_SUBUNIToP_3

$$\forall\ X : INTEGER \mid PRE1\ X \bullet PRE2\ X$$

vcVC_SUBUNIToP_4

$$\forall\ X, X_0 : INTEGER \mid PRE1\ X_0 \land POST2\ X \bullet POST1\ X$$

Notes

The ProofPower-ML command required to introduce the new script to contain the subunit has been suppressed for clarity. The first two VCs assert that the subunit specification statement refines the specification statement in the stub. The second two VCs arise from the subprogram body as in section 3.3.15.

3.3.18 Package Initialisation

Compliance Notation Script: Package Initialisation

Compliance Notation

package package_initialisation
is
  $*auxiliary* $A : \mathbb{N}$;
  procedure $Q(X : in\ out\ INTEGER)$
  $\Delta\ X\ \Xi\ A\ [APRE\ (A,\ X),\ APOST(A,\ X,\ X_0)]$;
end package_initialisation;

Compliance Notation

package body package_initialisation
is
  $*using* $C : INTEGER$; $*implement* A $*by* $INV(A,\ C)$;
  procedure $Q(X : in\ out\ INTEGER)$
  $\Delta\ X,\ C\ [CPRE\ C,\ CPOST(C,\ X,\ X_0)]$
  is begin
   $\Delta\ X,\ C\ [SPRE\ C,\ SPOST(C,\ X,\ X_0)]$ *(1)*
  end Q;
begin
  $\Delta\ C[IPRE\ C,\ IPOST\ C]$ *(2)*
end package_initialisation;

Generated VCs

vcPACKAGE_INITIALISATION_3

$$\forall\ C : INTEGER \bullet IPRE\ C$$

vcPACKAGE_INITIALISATION_4

$$\forall\ C : INTEGER \mid true \land IPOST\ C \bullet \exists\ A : \mathbb{N} \bullet INV\ (A,\ C)$$

Notes

The ProofPower-ML command required to introduce the new script to contain the subunit has been suppressed for clarity. The first 4 VCs generated by this example do not relate to the package initialisation and may be seen in cases 2 and 3 in section 3.3.16 above.

3.3.19 Range in Type Definition

Compliance Notation Script: Range in Type Definition

Compliance Notation

```
procedure range_type
is
 type T3 is (ONE, TWO, THREE);
 subtype SUB is T3 range ONE .. T3'SUCC(ONE);
begin
 null;
end range_type;
```

Generated VCs

vcSUB_1 $SUB \neq \varnothing$

Notes

Here, the heuristics used in the attempt to prove that the type *SUB* is non-empty did not succeed (because of the use of the successor attribute) and a VC has been generated. In normal use, the heuristics typically succeed in evaluating the bounds of the range and such a VC is not generated.

3.4 Domain Conditions

The Z translation of an Ada expression as described in section 3.1 may involve applications of partial functions. For example, the expression `1 / X`, is translated into *1 intdiv X*, which is an application of the partial function *_intdiv_*. Typically, application of a partial function to a value outside its domain would correspond to a situation in which execution of the expression would cause an exception to be raised. For example, ALRM requires execution of the expression `1 / X` to cause the exception NUMERIC_ERROR to be raised if `X` is equal to zero.

The Compliance Tool has an option to generate *domain conditions* derived from the Ada expressions in the program. These appear as additional hypotheses in the VCs. For example, the domain condition derived from the expression `1 / X` would be the hypothesis $X \neq 0$. These domain conditions satisfy the following soundness condition: if h is a domain condition, then any assignment of values to variables which makes h false would cause the code from which h has been derived to raise an exception.

Domain conditions are intended to enable the proof of VCs that would be unprovable without them. Domain conditions are not generated for operations such as integer addition which may raise an exception but which are represented in Z by total functions.

ALRM does not require real arithmetic operations to detect error conditions and raise exceptions in erroneous cases. The compiler-dependent value of the attribute P'MACHINE_OVERFLOWS indicates whether or not arithmetic on the real type P will cause exceptions to be raised for errors such as division by 0. If generation of domain conditions for real arithmetic is enabled, the VCs produced are sound subject to the proviso that this attribute is true for all real types used in the program.

3.5 Program Structure

The Z paragraphs in a Z document produced by processing a Compliance Notation script are put in ProofPower theories so as to manage the namespace in a way which represents the Ada program structure. For this to work correctly, the following rules are imposed: *(a)* a script may contain at most one compilation unit, and, *(b)*, a script delimits the scope of all the k-slots tags and specification statement tags it contains, i.e., the refinement, replacement, or arbitrary replacement step that expands a tag must be given in the same script as the k-slot or specification statement that introduced the tag.

A Z theory, referred to as the *script theory* is created for each compilation unit. The script theory is to be named according to conventions based on the name and type of the compilation unit in the script as shown in table 3.1. (The name is given as a parameter to the function *new_script* or *new_script1* called by convention at the beginning of each script, see *Compliance Tool — User Guide* [8]).

The script theory holds the Z paragraphs arising from the declarative part of the compilation unit. For each subprogram body, subprogram stub or block statement a theory is generated to hold the Z paragraphs associated with the declarative part of that subprogram body or block. A subprogram theory has as its parent a context theory which holds a snapshot of the state of the theory for the enclosing declarative part. A block theory has as its parent the theory associated with the enclosing declarative part. Any VCs arising from a declarative part go in the corresponding theory. Any VCs arising from the formal development of a sequence of statements go in the theory associated with the immediately enclosing declarative part.

A command *open_scope* is provided to navigate into the appropriate theory for a given declarative region. See *Compliance Tool — User Guide* [8] for more information.

The scheme for naming the theories is shown in figure 3.1. Note that the expanded names of blocks will include the names of any enclosing loop and block statements and these may be omitted in Ada. Omitted loop or block names are treated as empty strings and may give rise to clashes. The $*block* keyword may be used, if necessary, to provide a Compliance Notation name for an anonymous loop or block (see sections 2.5.5 and 2.5.6 for details).

Compilation Unit/Item Type	Example Unit Name	Example Script Name
Package Specification	`utils`	*UTILS'spec*
Package Body	`utils`	*UTILS'body*
Procedure Body	`update`	*UPDATE'proc*
	`utils.write`	*UTILSoWRITE'proc*
Function Body	`head`	*HEAD'func*
	`utils.READ`	*UTILSoREAD'func*
Context Theory	`utils.sort`	*UTILSoSORT'context*
Block Statement	`utils.sort.local`	*UTILSoSORToLOCAL'block*

Table 3.1: Use of Theories

COMPLIANCE NOTATION TOOLKIT

The Z paragraphs below support the translation of expressions and declaration into Z as described in chapter 3.

Essentially these paragraphs define the representations of the supported predefined types and their supported attributes. They also give representations for those predefined operators of Ada which are not directly supported by the Z library and provide for the translation of multi-dimensional array aggregates into Z.

z
function 0 **char_lit** _

z
function 0 **string_lit** _

z
function 0 _ **and** _, _ **or** _, _ **xor** _

z
function 0 _ **and_then** _, _ **or_else** _

z
function 0 _ **array_and** _, _ **array_or** _, _ **array_xor** _

z
function 0 _ **mod_and** _, _ **mod_or** _, _ **mod_xor** _

z
function 10 _ **mem** _, _ **notmem** _, _ **eq** _, _ **noteq** _

z
function 20 _ **less** _, _ **less_eq** _, _ **greater** _, _ **greater_eq** _

z
function 20 _ **real_less** _, _ **real_less_eq** _, _ **real_greater** _, _ **real_greater_eq** _

z
function 20 _ **array_less** _, _ **array_less_eq** _,
_ **array_greater** _, _ **array_greater_eq** _

z
function 30 _ $\&_0$ _, _ $\&_1$ _, _ $\&_2$ _

z

function 40 _ **intdiv** _, _ **rem** _, _ **intmod** _

z

function 50 _ ** _, **not** _, **array_not** _, **mod_not** _

z

$\mathbf{FALSE} \mathrel{\widehat=} 0$

z

$\mathbf{TRUE} \mathrel{\widehat=} 1$

z

$\mathbf{BOOLEAN} \mathrel{\widehat=} FALSE..TRUE$

z

$\mathbf{BOOLEANvFIRST} \mathrel{\widehat=} FALSE$

z

$\mathbf{BOOLEANvLAST} \mathrel{\widehat=} TRUE$

z

$\mathbf{BOOLEANvSUCC} \mathrel{\widehat=} (BOOLEAN \setminus \{BOOLEANvLAST\}) \lhd succ$

z

$\mathbf{BOOLEANvPRED} \mathrel{\widehat=} BOOLEANvSUCC^{\sim}$

z

$\mathbf{BOOLEANvPOS} \mathrel{\widehat=} id\ BOOLEAN$

z

$\mathbf{BOOLEANvVAL} \mathrel{\widehat=} BOOLEANvPOS^{\sim}$

z

$$
\begin{array}{|l}
\mathbf{not}\ \_ : BOOLEAN \to BOOLEAN; \\
\_\ \mathbf{and}\ \_, \_\ \mathbf{or}\ \_, \_\ \mathbf{xor}\ \_ : (BOOLEAN \times BOOLEAN) \to BOOLEAN \\
\hline
\forall\ b : BOOLEAN \bullet \\
\quad not\ FALSE = TRUE \land not\ TRUE = FALSE \land \\
\quad (b\ and\ FALSE = FALSE \land b\ and\ TRUE = b) \land \\
\quad (b\ or\ FALSE = b \land b\ or\ TRUE = TRUE) \land \\
\quad (b\ xor\ FALSE = b \land b\ xor\ TRUE = not\ b)
\end{array}
$$

z

$$
\begin{array}{|l}
\_\ \mathbf{and\_then}\ \_, \_\ \mathbf{or\_else}\ \_ : (BOOLEAN \times BOOLEAN) \to BOOLEAN \\
\hline
(\_\ and\_then\ \_) = (\_\ and\ \_) \land (\_\ or\_else\ \_) = (\_\ or\ \_)
\end{array}
$$

z

$$[X]$$

$$\_ \textbf{ mem } \_,\ \_ \textbf{ notmem } \_ : (X \times \mathbb{P}\ X) \to BOOLEAN;$$
$$\_ \textbf{ eq } \_,\ \_ \textbf{ noteq } \_ : (X \times X) \to BOOLEAN$$

$$\forall\ x,\ y : X;\ S : \mathbb{P}\ X;\ b : BOOLEAN \bullet$$
$$(b = x\ mem\ S \Leftrightarrow (b = TRUE \Leftrightarrow x \in S)) \land$$
$$(b = x\ notmem\ S \Leftrightarrow (b = TRUE \Leftrightarrow x \notin S)) \land$$
$$(b = x\ eq\ y \Leftrightarrow (b = TRUE \Leftrightarrow x = y)) \land$$
$$(b = x\ noteq\ y \Leftrightarrow (b = TRUE \Leftrightarrow x \neq y))$$

z

$$[X]$$

$$\textbf{array\_not } \_ : (X \pfun BOOLEAN) \to (X \pfun BOOLEAN);$$
$$\_ \textbf{ array\_and } \_,$$
$$\_ \textbf{ array\_or } \_,$$
$$\_ \textbf{ array\_xor } \_ : ((X \pfun BOOLEAN) \times (X \pfun BOOLEAN)) \to (X \pfun BOOLEAN)$$

$$\forall\ a,\ b : X \pfun BOOLEAN \bullet$$
$$array\_not\ a = (\lambda i : dom\ a \bullet not\ (a\ i)) \land$$
$$a\ array\_and\ b = (\lambda i : dom\ a \cap dom\ b \bullet a\ i\ and\ b\ i) \land$$
$$a\ array\_or\ b = (\lambda i : dom\ a \cap dom\ b \bullet a\ i\ or\ b\ i) \land$$
$$a\ array\_xor\ b = (\lambda i : dom\ a \cap dom\ b \bullet a\ i\ xor\ b\ i)$$

z

$$\_ \textbf{ less } \_,\ \_ \textbf{ less\_eq } \_,\ \_ \textbf{ greater } \_,$$
$$\_ \textbf{ greater\_eq } \_ : (\mathbb{Z} \times \mathbb{Z}) \to BOOLEAN$$

$$\forall\ x,\ y : \mathbb{Z};\ b : BOOLEAN \bullet$$
$$(b = x\ less\ y \Leftrightarrow (b = TRUE \Leftrightarrow x < y)) \land$$
$$(b = x\ less\_eq\ y \Leftrightarrow (b = TRUE \Leftrightarrow x \leq y)) \land$$
$$(b = x\ greater\ y \Leftrightarrow (b = TRUE \Leftrightarrow x > y)) \land$$
$$(b = x\ greater\_eq\ y \Leftrightarrow (b = TRUE \Leftrightarrow x \geq y))$$

z

$$\_ \textbf{ real\_less } \_,\ \_ \textbf{ real\_less\_eq } \_,\ \_ \textbf{ real\_greater } \_,$$
$$\_ \textbf{ real\_greater\_eq } \_ : (\mathbb{R} \times \mathbb{R}) \to BOOLEAN$$

$$\forall\ x,\ y : \mathbb{R};\ b : BOOLEAN \bullet$$
$$(b = x\ real\_less\ y \Leftrightarrow (b = TRUE \Leftrightarrow x <_R y)) \land$$
$$(b = x\ real\_less\_eq\ y \Leftrightarrow (b = TRUE \Leftrightarrow x \leq_R y)) \land$$
$$(b = x\ real\_greater\ y \Leftrightarrow (b = TRUE \Leftrightarrow x >_R y)) \land$$
$$(b = x\ real\_greater\_eq\ y \Leftrightarrow (b = TRUE \Leftrightarrow x \geq_R y))$$

z

$\_$ **array_less** $\_$,
$\_$ **array_less_eq** $\_$,
$\_$ **array_greater** $\_$,
$\_$ **array_greater_eq** $\_ : ((\mathbb{Z} \pfun \mathbb{Z}) \times (\mathbb{Z} \pfun \mathbb{Z})) \to BOOLEAN$

$\forall\ a, b : \mathbb{Z} \pfun \mathbb{Z} \bullet$
$\quad (a\ array\_less\ b = TRUE \Leftrightarrow$
$\quad (\exists i, j, k : \mathbb{Z} \bullet$
$\quad\quad \{i, j\} \subseteq dom\ b \land i - 1 \notin dom\ b \land i + k - 1 \notin dom\ a$
$\quad \land \quad (\forall t : i\ ..\ j - 1 \bullet t + k \in dom\ a \land b\ t = a(t + k))$
$\quad \land \quad j + k \in dom\ a \Rightarrow a(j + k) < b\ j)) \land$
$\quad a\ array\_less\_eq\ b = a\ array\_less\ b\ or\ a\ eq\ b \land$
$\quad a\ array\_greater\ b = b\ array\_less\ a \land$
$\quad a\ array\_greater\_eq\ b = b\ array\_less\_eq\ a$

z

$\_$ **intdiv** $\_$, $\_$ **rem** $\_$, $\_$ **intmod** $\_ : (\mathbb{Z} \times \mathbb{Z} \setminus \{0\}) \to \mathbb{Z}$

$\forall\ x, y : \mathbb{Z} \mid y \neq 0 \bullet$
$\quad (x * y \geq 0 \Rightarrow x\ intdiv\ y = abs\ x\ div\ abs\ y\ )$
$\land \quad (x * y < 0 \Rightarrow x\ intdiv\ y = \sim(abs\ x\ div\ abs\ y))$
$\land \quad x\ rem\ y = x - (x\ intdiv\ y) * y$
$\land \quad (x * y \geq 0 \lor x\ rem\ y = 0 \Rightarrow x\ intmod\ y = x\ rem\ y\ )$
$\land \quad (x * y < 0 \land x\ rem\ y \neq 0 \Rightarrow x\ intmod\ y = x\ rem\ y + y)$

z

$\_ ** \_ : (\mathbb{Z} \times \mathbb{N}) \to \mathbb{Z}$

$\forall\ x : \mathbb{Z}; y : \mathbb{N} \bullet x ** 0 = 1 \land x ** (y + 1) = x * x ** y$

z

integer_to_real: $\mathbb{Z} \to \mathbb{R}$

$\forall\ i : \mathbb{Z} \bullet integer\_to\_real\ i = real\ i$

z

real_to_integer : $\mathbb{R} \to \mathbb{Z}$

$\forall\ x : \mathbb{R} \bullet \sim_R 0.5 \leq_R x -_R real\ (real\_to\_integer\ x) \leq_R 0.5$

z

$\_$ **mod_and** $\_$, $\_$ **mod_or** $\_$, $\_$ **mod_xor** $\_ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

$\forall i : \mathbb{N} \bullet i \; mod\_and \; 0 = 0;$
$\forall i : \mathbb{N}; j : \mathbb{N}_1 \bullet$
$\quad i \; mod\_and \; j = 2 * (i \; div \; 2 \; mod\_and \; j \; div \; 2) + (i \; mod \; 2) * (j \; mod \; 2);$
$\forall i : \mathbb{N} \bullet i \; mod\_or \; 0 = i;$
$\forall i : \mathbb{N}; j : \mathbb{N}_1 \bullet$
$\quad i \; mod\_or \; j = 2 * (i \; div \; 2 \; mod\_or \; j \; div \; 2) + max\{i \; mod \; 2, j \; mod \; 2\};$
$\forall i : \mathbb{N} \bullet i \; mod\_xor \; 0 = i;$
$\forall i : \mathbb{N}; j : \mathbb{N}_1 \bullet$
$\quad i \; mod\_xor \; j = 2 * (i \; div \; 2 \; mod\_xor \; j \; div \; 2) + (i + j) \; mod \; 2$

z

mod_not $\_ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

$\forall i, \; modulus : \mathbb{Z} \bullet mod\_not(i, \; modulus) = modulus - (i + 1)$

z

INTEGER : $\mathbb{P} \; \mathbb{Z}$

z

INTEGERvFIRST, **INTEGERvLAST** : $\mathbb{Z}$;
INTEGERvSUCC, **INTEGERvPRED**,
INTEGERvPOS, **INTEGERvVAL** : $\mathbb{Z} \pfun \mathbb{Z}$

z

NATURAL $\widehat{=}$ *0 .. INTEGERvLAST*

z

NATURALvFIRST $\widehat{=}$ *0*

z

NATURALvLAST $\widehat{=}$ *INTEGERvLAST*

z

NATURALvSUCC $\widehat{=}$ *INTEGERvSUCC*

z

NATURALvPRED $\widehat{=}$ *INTEGERvPRED*

z

NATURALvPOS $\widehat{=}$ *INTEGERvPOS*

z

NATURALvVAL $\widehat{=}$ *INTEGERvVAL*

z

POSITIVE $\widehat{=}$ *1 .. INTEGERvLAST*

z
POSITIVEvFIRST $\widehat{=}$ *1*

z
POSITIVEvLAST $\widehat{=}$ *INTEGERvLAST*

z
POSITIVEvSUCC $\widehat{=}$ *INTEGERvSUCC*

z
POSITIVEvPRED $\widehat{=}$ *INTEGERvPRED*

z
POSITIVEvPOS $\widehat{=}$ *INTEGERvPOS*

z
POSITIVEvVAL $\widehat{=}$ *INTEGERvVAL*

z
| **LONG_INTEGER** : $\mathbb{P}\ \mathbb{Z}$

z
| **LONG_INTEGERvFIRST**, **LONG_INTEGERvLAST** : $\mathbb{Z}$;
| **LONG_INTEGERvSUCC**, **LONG_INTEGERvPRED**,
| **LONG_INTEGERvPOS**, **LONG_INTEGERvVAL** : $\mathbb{Z} \nrightarrow \mathbb{Z}$

z
| **SHORT_INTEGER** : $\mathbb{P}\ \mathbb{Z}$

z
| **SHORT_INTEGERvFIRST**, **SHORT_INTEGERvLAST** : $\mathbb{Z}$;
| **SHORT_INTEGERvSUCC**, **SHORT_INTEGERvPRED**,
| **SHORT_INTEGERvPOS**, **SHORT_INTEGERvVAL** : $\mathbb{Z} \nrightarrow \mathbb{Z}$

z
| **universal_discrete** : $\mathbb{P}\mathbb{Z}$

z
| **universal_discretevFIRST**, **universal_discretevLAST** : $\mathbb{Z}$;
| **universal_discretevSUCC**, **universal_discretevPRED**,
| **universal_discretevPOS**, **universal_discretevVAL** : $\mathbb{Z} \nrightarrow \mathbb{Z}$

z
| **FLOAT** : $\mathbb{P}\ \mathbb{R}$

z
| **FLOATvFIRST**, **FLOATvLAST** : $\mathbb{R}$;
| **FLOATvDIGITS** : $\mathbb{Z}$

z
$$\mathbf{SHORT\_FLOAT} : \mathbb{P}\,\mathbb{R}$$

z
$$\mathbf{SHORT\_FLOATvFIRST}, \mathbf{SHORT\_FLOATvLAST} : \mathbb{R}\ ;\ \mathbf{SHORT\_FLOATvDIGITS} : \mathbb{Z}$$

z
$$\mathbf{LONG\_FLOAT} : \mathbb{P}\,\mathbb{R}$$

z
$$\mathbf{LONG\_FLOATvFIRST}, \mathbf{LONG\_FLOATvLAST} : \mathbb{R}\ ;\ \mathbf{LONG\_FLOATvDIGITS} : \mathbb{Z}$$

z
$$\mathbf{CHARACTERvFIRST} \mathrel{\widehat=} 0$$

z
$$\mathbf{CHARACTERvLAST} : \mathbb{Z}$$
$$\vdash$$
$$CHARACTERvLAST \geq 127$$

z
$$\mathbf{CHARACTER} \mathrel{\widehat=} CHARACTERvFIRST \mathinner{..} CHARACTERvLAST$$

z
$$\mathbf{CHARACTERvSUCC} \mathrel{\widehat=} (CHARACTER \setminus \{CHARACTERvLAST\}) \lhd succ$$

z
$$\mathbf{CHARACTERvPRED} \mathrel{\widehat=} CHARACTERvSUCC^{\sim}$$

z
$$\mathbf{CHARACTERvPOS} \mathrel{\widehat=} id\ CHARACTER$$

z
$$\mathbf{CHARACTERvVAL} \mathrel{\widehat=} CHARACTERvPOS^{\sim}$$

z
$$\mathbf{STRING} : \mathbb{P}\,(POSITIVE \pfun CHARACTER)$$

z
$$\mathbf{Z\_CHAR} \mathrel{\widehat=} seq\ \mathbb{S}$$

z
$$\mathbf{Z\_STRING} \mathrel{\widehat=} seq\ \mathbb{S}$$

z
$$\mathbf{dest\_char} : \mathbb{S} \to \mathbb{Z}$$
$$\vdash$$
$$\forall ch : \mathbb{S} \bullet dest\_char\ ch = \ulcorner \mathbb{NZ}\ (RepChar\ ch) \urcorner$$

z

string_lit _ : $Z\_STRING \rightarrow seq\ CHARACTER$

(*

z

*)

$\forall str : Z\_STRING \bullet string\_lit\ str = dest\_char \circ str$

z

char_lit _ : $Z\_CHAR \rightarrow CHARACTER$

(*

z

*)

$(char\_lit\ \_) = head \circ (string\_lit\ \_)$

z

[**Informal_Function**]

z

$[X]$

$\_\&_{\mathbf{0}}\_ : (\mathbb{Z} \pfun X) \times (\mathbb{Z} \pfun X) \rightarrow (\mathbb{Z} \pfun X)$;
$\_\&_{\mathbf{1}}\_ : (\mathbb{Z} \pfun X) \times X \rightarrow (\mathbb{Z} \pfun X)$;
$\_\&_{\mathbf{2}}\_ : X \times (\mathbb{Z} \pfun X) \rightarrow (\mathbb{Z} \pfun X)$

$\forall\ a, b : \mathbb{Z} \pfun X;\ m, n : \mathbb{Z}$
$\mid dom\ a \mapsto m \in max \land dom\ b \mapsto n \in min \bullet$
$a\ \&_0\ b = a \oplus \{i : dom\ b \bullet i + m + 1 - n \mapsto b\ i\}$;

$\forall\ a : \mathbb{Z} \pfun X \bullet a\ \&_0\ \varnothing = a$;

$\forall\ a : \mathbb{Z} \pfun X;\ x : X \bullet a\ \&_1\ x = a\ \&_0\ \langle x \rangle$;

$\forall\ a : \mathbb{Z} \pfun X;\ x : X \bullet x\ \&_2\ a = \langle x \rangle\ \&_0\ a$

z

$[X, Y]$

---

slide : $(X \pfun Y) \times \mathbb{P}X \rightarrow (X \pfun Y)$

$\forall f : X \pfun Y;\ r : \mathbb{P}X \mid dom\ f = r \bullet slide(f, r) = f$

Informal Z

$[X1, X2, X]$

$array\_agg2 : (X1 \rightarrow X2 \rightarrow X) \rightarrow (X1 \times X2 \rightarrow X)$

$\forall f : X1 \rightarrow X2 \rightarrow X;\ x1 : X1;\ x2 : X2 \bullet$
$array\_agg2\ f\ (x1, x2) = f\ x1\ x2$

Informal Z

$$[X1, X2, X3, X]$$

$$array\_agg3 : (X1 \to X2 \to X3 \to X) \to (\ X1 \times X2 \times X3 \to X)$$

$$\forall f : X1 \to X2 \to X3 \to X;\ x1 : X1;\ x2 : X2;\ x3 : X3 \bullet$$
$$array\_agg3\ f\ (x1,\ x2,\ x3) = f\ x1\ x2\ x3$$

Informal Z

$$[X1, X2, X3, X4, X]$$

$$array\_agg4 : (X1 \to X2 \to X3 \to X4 \to X) \to (\ X1 \times X2 \times X3 \times X4 \to X)$$

$$\forall f : X1 \to X2 \to X3 \to X4 \to X;\ x1 : X1;\ x2 : X2;\ x3 : X3;\ x4 : X4 \bullet$$
$$array\_agg4\ f\ (x1,\ x2,\ x3,\ x4) = f\ x1\ x2\ x3\ x4$$

and so on up to *array_agg20*.

REFERENCES

[1] John Barnes. *High Integrity Ada — The Spark Approach*. Addison-Wesley, 1997.

[2] ANSI/MIL-STD-1815A-1983. *The Annotated Ada Reference Manual*. Karl A. Nyberg, Ada Joint Program Office, 1983.

[3] DRA/CIS/CSE3/TR/94/27. *Specification of the compliance notation for SPARK and Z. (3 Volumes)*. C.M. O'Halloran, C.T. Sennett, and A. Smith, Defence Research Agency, Malvern.

[4] DRA/CIS(SE2)/PROJ/SWI/TR/1/1.1. *A commentary on the specification of the compliance notation for SPARK and Z*. C.M. O'Halloran, C.T. Sennett, and A. Smith, Defence Research Agency, Malvern, 1st November 1995.

[5] DS/FMU/IED/USR005. *ProofPower Description Manual*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[6] DS/FMU/IED/USR014. *ProofPower Software and Services*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[7] ISO/IEC 8652:1995. *Ada Reference Manual*. International Standards Organisation, 1995.

[8] ISS/HAT/DAZ/USR501. *Compliance Tool — User Guide*. Lemma 1 Ltd., `http://www.lemma-one.com`.

INDEX