A Formal Description of the Incremental Translation of Stage 2 VHDL into State Deltas in the State Delta Verification System (SDVS)

30 September 1992

Prepared by

I. V. FILIPPENKO
Computer Systems Division

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group
A FORMAL DESCRIPTION OF THE INCREMENTAL TRANSLATION OF STAGE 2 VHDL INTO STATE DELTAS IN THE STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared by
I. V. Filippenko
Computer Systems Division

30 September 1992

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for
NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

PUBLIC RELEASE IS AUTHORIZED
A FORMAL DESCRIPTION OF THE INCREMENTAL TRANSLATION
OF STAGE 2 VHDL INTO STATE DELTAS IN THE
STATE DELTA VERIFICATION SYSTEM (SDVS)

Prepared

I. V. Filipenko

I. V. Filipenko

Approved

B. H. Levy, Manager
Computer Assurance Section

D. B. Baker, Director
Trusted Computer Systems Department

C. A. Sunshine, Principal Director
Computer Science and Technology
Subdivision
Abstract

This report documents a formal semantic specification of Stage 2 VHDL, a subset of the VHSIC Hardware Description Language (VHDL), via translation into the temporal logic of the State Delta Verification System (SDVS). Stage 2 VHDL is the third of successively more sophisticated VHDL subsets to be interfaced to SDVS.

The specification is a continuation-style denotational semantics of Stage 2 VHDL in terms of state deltas, the distinguishing logical formulas used by SDVS to describe state transitions. The semantics is basically specified in two phases. The first phase performs static semantic analysis, including type checking and other static error checking, and collects an environment for use by the second phase. The second phase performs the actual translation of the subject Stage 2 VHDL description into state deltas. An abstract syntax tree transformation is interposed between the two translation phases.

The translator specification was, for the most part, written in DL, the semantic metalanguage of a denotational semantics specification system called DENOTE. DENOTE enables the semantic equations of the specification to be realized both as a printable representation (included in this report) and an executable Common Lisp program that constitutes the translator's implementation. However, the second phase semantics of the VHDL simulation cycle has a direct operational implementation in the VHDL translator code.
# Contents

Abstract v

1 Introduction 1

2 Our Semantic Approach to Stage 2 VHDL 5

3 Overview of Stage 2 VHDL 7
   3.1 General Remarks ................................ 7
   3.2 Stage 2 VHDL Language Features .................... 8

4 Preliminaries 11
   4.1 Environments ................................ 11
   4.2 Continuations ................................ 14
   4.3 Other Notation and Functions ..................... 14

5 Syntax of Stage 2 VHDL 17
   5.1 Syntactic Domains ................................ 18
   5.2 Syntax Equations ................................ 18
      5.2.1 Concrete Syntax ............................. 18
      5.2.2 Abstract Syntax: Phase 1 .................... 29
      5.2.3 Abstract Syntax: Phase 2 .................... 33

6 Phase 1: Static Semantic Analysis and Environment Collection 35
   6.1 Overview ........................................ 35
   6.2 Descriptors, Types, and Type Modes ................ 36
   6.3 Special-Purpose Environment Components and Functions .... 44
   6.4 Phase 1 Semantic Domains and Functions ............... 45
   6.5 Phase 1 Semantic Equations ........................ 49
      6.5.1 Stage 2 VHDL Design Files ................... 49
      6.5.2 Entity Declarations .......................... 50
      6.5.3 Architecture Bodies .......................... 50
      6.5.4 Port Declarations ............................ 51
6.5.5 Declarations ........................................ 52
6.5.6 Concurrent Statements ............................. 67
6.5.7 Sensitivity Lists .................................... 69
6.5.8 Sequential Statements ............................... 70
6.5.9 Case Alternatives .................................. 77
6.5.10 Discrete Ranges .................................... 78
6.5.11 Waveforms and Transactions ..................... 79
6.5.12 Expressions ........................................ 80
6.5.13 Primitive Semantic Equations .................... 86

7 Interphase Abstract Syntax Tree Transformation 87
7.1 Interphase Semantic Functions ........................ 87
7.2 Transformed Abstract Syntax of Names ............... 88
7.3 Interphase Semantic Equations ........................ 89
  7.3.1 Stage 2 VHDL Design Files ....................... 89
  7.3.2 Entity Declarations ............................... 89
  7.3.3 Architecture Bodies ............................... 89
  7.3.4 Port Declarations ................................ 89
  7.3.5 Declarations ...................................... 90
  7.3.6 Concurrent Statements ............................. 91
  7.3.7 Sensitivity Lists .................................. 91
  7.3.8 Sequential Statements ............................. 92
  7.3.9 Case Alternatives ................................ 93
  7.3.10 Discrete Ranges .................................. 93
  7.3.11 Waveforms and Transactions .................... 93
  7.3.12 Expressions ...................................... 93

8 Phase 2: State Delta Generation 97
8.1 Phase 2 Semantic Domains and Functions ............. 97
8.2 Phase 2 Execution State ............................. 101
  8.2.1 Unique Name Qualification ....................... 101
1 Introduction

The State Delta Verification System (SDVS), under development over the course of several years at The Aerospace Corporation, is an automated verification system that aids in writing and checking proofs that a computer program or (design of a) digital device satisfies a formal specification.

The long-term goal of the SDVS project is to create a prototype of a production-quality verification system that is useful at all levels of the hierarchy of digital computer systems; our aim is to verify hardware from gate-level designs to high-level architecture, and to verify software from the microcode level to application programs written in high-level programming languages. We are currently extending the applicability of SDVS to both lower levels of hardware design and higher levels of computer programs. A technical overview of the system is provided by [1], while detailed information on the system may be found in [2] and [3].

Several features distinguish SDVS from other verification systems (refer to [4] for a detailed discussion). The underlying temporal logic of SDVS, called the state delta logic, has a formal model-theoretic semantics. SDVS is equipped with a theorem prover that runs in interactive or batch modes; the user supplies high-level proof commands, while many low-level proof steps are executed automatically. One of the more distinctive features of SDVS is its flexibility — there is a well-defined and relatively straightforward method of adapting the system to arbitrary application languages (to date: ISPS, Ada, and VHDL). Furthermore, descriptions in the application languages potentially serve as either specifications or implementations in the verification paradigm. Incorporation of a given application language is accomplished by translation to the state delta logic via a Common Lisp translator program, which is (generally) automatically derived from a formal denotational semantics for the application language.

Prior to 1987 we adapted SDVS to handle a subset of the hardware description language ISPS. However, ISPS has serious limitations regarding the specification of hardware at levels other than the register transfer level. In fiscal year 1988 we documented a study of some of the hardware verification research being conducted outside Aerospace and investigated VHDL, an IEEE and DoD standard hardware description language released in December 1987. We selected VHDL as a possible medium for hardware description within SDVS.

The aim of the ongoing formal hardware verification effort in SDVS is to verify hardware descriptions written in VHDL (VHSIC Hardware Description Language). This choice of hardware description language is particularly well-suited to our overall aim of verifying hardware designs across the spectrum from gate-level designs to high-level architectures. Indeed, the primary hardware abstraction in VHDL, the design entity, represents any portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function. As such, "a design entity may represent an entire system, a sub-system, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between" [5].

Prerequisites for adapting SDVS to VHDL are (1) to define VHDL semantics formally in terms of SDVS's underlying logic (the state delta logic) and (2) to implement a translator from VHDL to the state delta logic. As with the incorporation of Ada into SDVS [6], the
approach taken with VHDL has been to implement increasingly complex language subsets; this has enabled a graded, structured approach to hardware verification.

In fiscal year 1989 we defined an initial subset of VHDL, called Core VHDL, that captured the most essential behavioral features of VHDL, including: ENTITY declarations; ARCHITECTURE bodies; CONSTANT, VARIABLE, SIGNAL, and PORT declarations; variable and signal assignment statements; IF, CASE, WAIT, and NULL statements; and concurrent PROCESS statements. We defined both the concrete syntax and the abstract syntax for Core VHDL, formally specified its semantics and, on the basis of this semantic definition, implemented a Core-VHDL-to-state-delta translator [7].

In fiscal year 1990, SDVS was enhanced to provide the capability of verifying hardware descriptions written in Core VHDL [8, 9]. In fiscal year 1991, the translator underwent extensive revision to accommodate a second VHDL subset, Stage 1 VHDL [10], which included: WAIT statements in arbitrary contexts; LOOP, WHILE, and EXIT statements; TRANSPORT delay; aggregate signal assignments; and a revised translator structure.

Implemented in fiscal year 1992, Stage 2 VHDL is a considerably more complex and capable VHDL language subset. Stage 2 VHDL extends Stage 1 VHDL with the addition of the following VHDL language features: (restricted) design files, declarative parts in entity declarations, package STANDARD (containing predefined types BOOLEAN, BIT, INTEGER, TIME, CHARACTER, REAL, STRING, and BIT VECTOR), user-defined packages, USE clauses, array type declarations, enumeration types, subprograms (procedures and functions, excluding parameters of object class SIGNAL), concurrent signal assignment statements, FOR loops, octal and hexadecimal representations of bitstrings, ports of default object class SIGNAL, and general expressions of type TIME in AFTER clauses.

The VHDL and Ada translators have been reengineered to a uniform implementation reflecting language similarities where these exist, and optimized for greater space- and time-efficiency.

The purpose of the present report is to provide a formal description of the translation of Stage 2 VHDL hardware descriptions into state deltas. This amounts to a formal semantic specification of Stage 2 VHDL, presented herein as a continuation-style denotational semantics [11] for which the state delta language provides the domain of meanings. The translation basically consists of parsing followed by two semantic analysis phases.

The first phase receives the abstract syntax tree generated by the Stage 2 VHDL parser for a given hardware description, and:

- performs static semantic analysis, including type checking;
- collects an environment that associates all names declared in the subject Stage 2 VHDL hardware description with their attributes;
- appropriately uniquely qualifies identical names declared in different scopes, as required by the static block structure of the hardware description; and
- for the convenience of the second phase, transforms the abstract syntax tree of the subject hardware description.
Phase 2 receives the transformed abstract syntax tree and the environment constructed by Phase 1, and uses these to translate the Stage 2 VHDL hardware description into state deltas. This translation is incremental, in the sense that it is driven by symbolic execution of the hardware description, producing further state deltas as symbolic execution proceeds.

The Stage 2 VHDL formal description is an extensive revision and expansion of the formal specifications of the Core VHDL and Stage 1 VHDL translators [7, 10]. The Stage 2 VHDL translator specification was written in DL, the semantic metalanguage of a denotational semantics specification system called DENOTE [12]. DENOTE enables the semantic equations of the specification to be automatically translated into both a printable representation (included in this report) and an executable Common Lisp program that constitutes the translator's implementation.

This report is organized as follows.

- Our approach to the semantics of Stage 2 VHDL is discussed in Section 2.
- Section 3 contains an overview of the Stage 2 VHDL subset.
- Section 4 provides preliminary information (background and notation) on the particular method of semantic description used.
- Section 5 lists both the concrete and abstract syntax of Stage 2 VHDL.
- Section 6 presents the Stage 2 VHDL static semantics.
- Section 7 presents the interphase abstract syntax tree transformation.
- Section 8 presents the Stage 2 VHDL dynamic semantics in terms of state deltas.
- Finally, some concluding remarks are made in Section 9.
Our Semantic Approach to Stage 2 VHDL

The approach we have taken to translating Stage 2 VHDL descriptions is much the same as that for Stage 1 VHDL, but differs in an essential respect from the Core VHDL approach. For the sake of completeness, we shall recapitulate this difference here.

The VHDL translator essentially functions as a simulator kernel, maintaining a clock and a list of future events that are defined as state deltas. For Core VHDL, however, the translator transformed possibly multiple Core VHDL statements: sequential statements between \texttt{WAIT} statements within a process were all translated and then \textit{composed} into a single state delta. The translator updated the clock to the next time at which a signal driver became active or a process resumed. As the clock advanced, the translator \textit{merged} the composite state deltas into a single state delta that specified the behavior of all processes at that point in the execution.

For Stage 1 VHDL, we re-evaluated the feasibility of using composition in the translation of VHDL to state deltas, and concluded that although composition had initially seemed viable in the case of Core VHDL, it is \textit{impossible in principle} to apply the technique to more complex VHDL subsets, as the attempt would require the ability to compose over sections of VHDL code that would necessitate static proof in SDVS. More generally, the ability to compose over arbitrary \texttt{WAIT}-bracketed code in \texttt{PROCESS} statements would be tantamount to the automatic construction of correctness proofs without user intervention — a theoretically undecidable problem.

Therefore, we decided to abandon composition for Stage 1 VHDL and succeeding SDVS VHDL subsets. Instead, within a given execution (simulation) cycle, processes are translated sequentially, in the order in which they appear in the VHDL description, and the user has control over stepping through the sequential statements within each process. Thus, rather than trying to have the VHDL translator model the concurrency of the processes, we choose to take for granted a certain “metatheorem” about VHDL: that any two sequentializations of the processes are equivalent. A brief justification for this point of view is that the problem of mutual exclusion is not a concern in VHDL, since

- all variables are local to the process in which they are declared, and
- distinct processes modify distinct drivers of a given signal (known as a \textit{resolved signal}), and the ultimate signal value is obtained by application of a user-defined \textit{resolution function}.\footnote{As of Stage 2 VHDL, however, \textit{resolved signals} are still disallowed.}

A gratifying benefit of the revised translation strategy is that the understandability of the resulting proofs has been remarkably improved — the dynamic flow of process execution precisely reflects the simulation semantics of VHDL (as defined in the \textit{VHDL Language Reference Manual} \cite{5}), but with the crucial aspect of symbolic execution (use of abstract values rather than concrete) thrown in. The current VHDL translator thus functions as a “symbolic simulator,” and is a considerably more intuitive proof engine than was its incarnation for Core VHDL.
Enhancements that need to be made to SDVS to support efficient Stage 2 VHDL proofs consist principally of Simplifier support for reasoning about symbolic representations of VHDL time, and a command for induction over simulation cycles of a VHDL description (or an articulation of how to use the current induct command for that purpose).
3 Overview of Stage 2 VHDL

Stage 2 VHDL comprises a relatively powerful behavioral subset of VHDL. That is to say, Stage 2 VHDL descriptions are confined to the specification of hardware behavior or data flow, rather than structure. More comprehensive VHDL subsets for SDVS (anticipated: Stage 3 VHDL) will include constructs for the structural description of hardware in terms of its hierarchical decomposition into connected subcomponents. The Stage 2 VHDL data types are: BOOLEAN, BIT, INTEGER, REAL (preliminary version), TIME (a predefined physical type of INTEGER range), CHARACTER, STRING (arrays of characters), BIT_VECTOR (arrays of bits), user-defined enumeration types, and user-defined array types.

3.1 General Remarks

The primary VHDL abstraction for modeling a digital device is the design entity. A design entity consists of two parts: an entity declaration, providing an external view of the component by declaring the input and output ports, and an architecture body, giving an internal view in terms of component behavior or structure.

In Stage 2 VHDL, each architecture body is constrained to be behavioral, consisting of a set of declarations and concurrent statements defining the functional interpretation of the device being modeled. The allowable concurrent statements are of two kinds: PROCESS statements and concurrent signal assignment statements, to be discussed below.

A PROCESS statement, the most fundamental kind of behavioral concurrent statement in VHDL, is a block of sequential zero-time statements that execute sequentially but "instantaneously" in zero time [13], and some (possibly none) distinguished sequential WAIT statements whose purpose is to suspend process execution and allow time to elapse.

A process typically schedules future values to appear on data holders called signals, by means of sequential signal assignment statements. The execution of a signal assignment statement does not immediately update the value of the target signal (the signal assigned to); rather, it updates the driver associated with the signal by placing (at least one) new transaction, or time-value pair, on the waveform that is the list of such transactions contained in the driver. Each transaction projects that the signal will assume the indicated value at the indicated time; the time is computed as the sum of the current clock time of the model and the delay specified (explicitly or implicitly) by the signal assignment statement.

Two types of time delay can be specified by a sequential signal assignment statement, and Stage 2 VHDL encompasses both. Inertial delay, the default, models a target signal's inertia that must be overcome in order for the signal to change value; that is, the scheduled new value must persist for at least the time period specified by the delay in order actually to be attained by the target signal. Transport delay, on the other hand, must be explicitly indicated in the signal assignment statement with the reserved word TRANSPORT, and models a "wire delay" wherein any pulse of whatever duration is propagated to the target signal after the specified delay.
In lieu of explicit WAITs, a process may have a sensitivity list of signals that activate process resumption upon receiving a distinct new value (an event). The sensitivity list implicitly inserts a WAIT statement as the last statement of the process body.

The other class of concurrent statement in Stage 2 VHDL is that of concurrent signal assignment statements. These always represent equivalent PROCESS statements, and come in two varieties: conditional signal assignment and selected signal assignment. A conditional signal assignment is equivalent to a process with an embedded IF statement whose branches are sequential signal assignments; similarly, a selected signal assignment is equivalent to a process with an embedded (possibly degenerate) CASE statement whose branches are sequential signal assignments. The VHDL translator syntactically transforms concurrent signal assignment statements to their corresponding PROCESS statements prior to translation into state deltas.

Signals act as data pathways between processes. Each process applies operations to values being passed through the design entity. We may regard a process as a program implementing an algorithm, and a Stage 2 VHDL description as a collection of independent programs running in parallel.

In full VHDL, a target signal can be assigned to in multiple processes, in which case it possesses correspondingly many drivers for updating by the different processes; the value taken on by the signal at any particular time is then computed by a user-defined resolution function of these drivers. As did previous SDVS VHDL subsets, Stage 2 VHDL disallows such resolved signals: a signal is not permitted to appear as the target of a sequential signal assignment statement in more than one process body; equivalently, every signal has a unique driver.

3.2 Stage 2 VHDL Language Features

Concrete and abstract syntaxes for Stage 2 VHDL have been defined — see Section 5 — as required, of course, for the implementation of the Stage 2 VHDL translator. Perhaps the following summary provides the best way of seeing the Stage 2 VHDL language subset and translator at a glance.

- VHDL design files
  - user-defined packages (optional), USE clauses (optional), entity declaration, architecture body
  - restriction: unique entity and architecture per file

- package STANDARD
  - predefined types: BOOLEAN, BIT, INTEGER, TIME, CHARACTER, REAL, STRING, BIT_VECTOR
  - various units of type TIME: FS, PS, NS, US, MS, SEC, MIN, HR
  - restriction: the implementation of type REAL is preliminary; it supports parsing and Phase 1 translation but no Phase 2 reasoning
• user-defined packages
  – package declarations
  – package bodies

• USE clauses for accessing packages
  – restriction: ALL is the only permissible suffix

• entity declarations
  – entity header: port declarations
  – entity declarative part: other declarations

• architecture bodies

• object declarations
  – CONSTANT, VARIABLE, SIGNAL
  – octal and hexadecimal representations of bitstrings
  – entity ports of default object class SIGNAL

• array type declarations
  – arrays of arbitrary element type
  – bidirectional arrays, unconstrained arrays

• user-defined enumeration types

• signals of arbitrary array and enumeration types

• subprograms
  – procedures and functions: declarations and bodies
  – restriction: excluding parameters of object class SIGNAL

• concurrent statements
  – PROCESS statements
  – conditional signal assignments
  – selected signal assignments

• sequential statements
  – null statement: NULL
  – variable assignments (scalar & composite)
  – signal assignments (scalar & composite, inertial or TRANSPORT delay)
  – conditionals: IF, CASE
  – loops: LOOP, WHILE, FOR
loop exits: EXIT
- subprogram calls
- subprogram return: RETURN
- process suspension: WAIT

operators
- numeric unary operators: ABS, +, -
- numeric binary operators: +, -, *, /, ** (exponentiation), MOD (modulus), REM (remainder)
- boolean and bit operators: NOT, AND, NAND, OR, NOR, XOR
- relational operators: =, /=, <, <=, >, and >=
- array concatenation operator: &

restriction: =, /=, and & are the only Stage 2 VHDL operators defined for composite types (i.e., BIT_VECTOR and user-defined array types).
4 Preliminaries

The purpose of this section is to provide some of the background and notation necessary for the research documented in this report. It is assumed that the reader is familiar with

- the descriptive aspects of the denotational technique for expressing the semantics of programming languages (including concepts such as syntax, semantic functions, lambda notation, curried function notation, environments, and continuations) as presented in [11]; and
- the theory and practice of state deltas [2, 14, 15].

Denotational semantic definitions of programming languages consist of two parts: syntax and semantics. The syntax part consists of domain equations (equivalent to productions of a context-free grammar) that define the syntactic variables (analogous to grammar nonterminals) and the (abstract) syntactic elements of the language. The semantic part defines a semantic function for each syntactic variable and the definition (by syntactic cases) of these functions; it also defines auxiliary functions that are used in the definition of the semantic functions. The semantic functions constitute a syntax-directed mapping from the syntactic constructs of the language to their corresponding semantics.

Certain principal notions, among which are environments and continuations, are central to standard denotational semantic definitions of programming languages.

4.1 Environments

Environments are functions from identifiers to their "definitions"; these definitions are called denotable values. Identifiers that have no corresponding definition are formally bound to the special token *UNBOUND*. The identifiers are names for objects (e.g., constants, variables, procedures, and exceptions) in a program written in the language being defined. Environments are usually created and modified by the elaboration of declarations in the language.

The domain of environments, Env, is typically

\[ \text{Env} = \text{Id} \rightarrow (\text{Dv} + \text{*UNBOUND*}) \]

where Id and Dv are, respectively, the domains of identifiers and denotable values. If \( r \) is an environment, then \( r(id) \) is the value (*UNBOUND* or a Dv-value) bound to the identifier id. The empty environment \( r0 \) is the environment in which \( r0(id) = \text{*UNBOUND*} \) for every identifier id. In definitions of languages that have block-structured scoping, it is necessary to combine two environments that may each associate a denotable value with the same identifier. If \( r1 \) and \( r2 \) are environments, then \( r1[r2] \) is a combined environment defined by

\[ r1[r2](id) = (r2(id) = \text{*UNBOUND*} \rightarrow r1(id), r2(id)) \]

where \( (a \rightarrow b,c) \) is an abbreviation for if a then b else c. That is, in \( r1[r2] \), the \( r2 \)-value of an identifier "overrides" the \( r1 \)-value of that same identifier, except when its \( r2 \)-value is
*UNBOUND*. An environment can be changed by this means. If \( r \) is an environment, \( d \) a value, and \( id \) an identifier, then \( r[d/id] \) denotes an environment that is the same as \( r \) except that \((r[d/id])(id) = d\).

**Tree-Structured Environments**

When the use of the above combination of environments is inconvenient or inappropriate, it is sometimes necessary to use a structured collection of environments. A tree-structured environment (TSE) is a tree whose nodes are environments and whose edges are labeled by identifiers or numerals, called edge labels, where no two edges emanating from a given node can have the same label. A path is a list of zero or more edge labels. Such a path denotes a sequence of connected edges from the root node to another node of a tree-structured environment. A path \( p \) can be extended by an edge labeled \( elbl \) via \(%(p)(elbl)\), where

\[
%(path)(id) = append(path,(id))
\]

Formally, a TSE can be regarded as a partial function from paths to environments. Thus the set of paths in a TSE \( t \) is precisely the set of paths \( p \) for which \( t(p) \) is defined. If \( t \) is a TSE and \( p \) is a path in \( t \), then \( t(p) \) denotes the unique environment in \( t \) located at the end of \( p \).

If \( t \) is a TSE and \( p \) is one of its paths, the pair \((t,p)\) can be used to represent the set of environments containing all of the identifier bindings visible at a given point in a Stage 2 VHDL hardware description, where the identifiers in \( p \) are the names of the lexical scopes whose local environments are on the path \( p \). At the program point whose identifier bindings are represented by \((t, (elbl_1, \ldots, elbl_n))\), \( t((elbl_1, \ldots, elbl_n)) \) is the most local set of bindings, \( \ldots \), and \( t(\epsilon) \) is the most global set of bindings, where \( \epsilon \) denotes the empty path. Thus \( t(p)(id) \) is the value bound to \( id \) in the most local environment of \((t,p)\).

**Qualified Names**

The same identifier is bound in every component environment of a TSE, although many (if not most) of those bindings may be to *UNBOUND*. It is convenient to be able to distinguish uniquely an occurrence of an identifier by prefixing to the identifier a representation of the path that designates the location in the TSE of the environment associated with that instance. Such a uniquely distinguished identifier will be called a fully qualified name. Thus if \( t \) is a TSE, \( p \) one of its paths, and \( id \) an identifier, then \( $(p)(id) \) is \( id \)'s fully qualified name relative to \( t(p) \). If \( p = (elbl_1, \ldots, elbl_n) \), then \( $(p)(id) \) is represented as \( elbl_1.elbl_2. \ldots .elbl_n.id \). When \( p = \epsilon \) (empty path), \( $(\epsilon)(id) \) is simply represented by \( id \). \$ is defined by

\[
$(path)(id) = (path = \epsilon \rightarrow id, $(rest(path))(catenate(last(path), ".", id)))
\]

The function rest returns a list consisting of the first \( n - 1 \) elements of an \( n \)-element list, and catenate is a curried function that concatenates its (variable number of) arguments into an atom.

Identifiers qualified with the full TSE path that locates their associated component environment are cumbersome and hard to read. If only those instances of identifiers not bound to *UNBOUND* are of interest, then such full name qualification may be unnecessary.
Often a suffix of this path is sufficient to distinguish uniquely an instance of such an identifier. An identifier so qualified is said to be uniquely qualified. In the limit, if all identifiers not bound to *UNBOUND* were distinct, then no qualification (an empty suffix) would be necessary to distinguish them. Given a TSE, it is possible to determine the minimum path suffix necessary to distinguish uniquely each identifier instance; this is done in our implementation of Stage 2 VHDL.

Descriptors

The denotable values to which identifiers are bound in the component environments of a TSE are called descriptors.

A descriptor contains several fields of information, each of which holds an attribute of the identifier instance to which the descriptor is bound in a given TSE component environment. The number of fields in a descriptor depends on the attributes of its associated identifier, but each descriptor always has fields that contain the identifier to which it is bound, the identifier instance's *statically uniquely qualified name* (see Section 8.2.1), and a tag that identifies the kind of descriptor (and hence its remaining fields).

Descriptors for Stage 2 VHDL are discussed in detail in Section 6.2.

Tree-Structured Environment Access

Certain non-*UNBOUND* (i.e., denotable) values of an identifier id in (t,p) can be accessed by the functions lookup and lookup-local. These functions are given later in the context of semantics equations in which they are used.

Tree-Structured Environment Modification

A TSE's component environments can be modified (in particular, descriptors can be bound to unbound identifiers or existing descriptors can be modified) via a function built into DENOTE. This function, enter, is used extensively in the DENOTE description of the Stage 2 VHDL translator. enter(t)(p)(id)(d), where t is a TSE, p a path in t, id an identifier, and d a partial descriptor (containing all its fields except the identifier field), yields a TSE that is the same as t except that its component environment t(p) is replaced by the environment

\[ t(p)[d'/id], \]

where if \( d = \langle \text{qid}, \text{tag}, \ldots \rangle \), then \( d' = \langle \text{id}, \text{qid}, \text{tag}, \ldots \rangle \).

Tree-Structured Environment Extension

One can add additional component environments to a TSE by extending it. If t is a TSE, p a path in t, and elbl an edge label, and if \%(p)(elbl) is not a path in t, then extend(t)(p)(elbl) denotes the TSE that is the same as t except that \( (\text{extend}(t)(p)(\text{elbl}))(\%(p)(\text{elbl})) = r0 \). Thus one can extend t along one of its paths p by adding a legally labeled edge onto the end of p and placing a node that is the empty environment r0 at the end of that extended path \%(p)(elbl).
4.2 Continuations

Continuations are a technical device for capturing the semantics of transfers of control, whether they be explicit (gotos, returns from procedures and functions) or implicit (normal sequential flow of control to the next program element, abnormal termination of program execution). Continuations are functions intended to map the "normal" result of a semantic function to some ultimate "final answer" [some final value(s) or an error message]. If the semantic function does not produce a normal result, its continuation can be ignored and some "abnormal" final answer (such as an error message) can be produced instead.

For example, in the first phase of our formal description of the Stage 2 VHDL translator, a continuation supplied to a semantic function that elaborates declarations normally maps a new "translation state" to a final answer, but if a declaration illegally duplicates or conflicts with an existing definition, then the continuation is ignored and an error message (such as DUPLICATE-DECLARATION) is the resulting final answer.

The initiation of the second phase of our formal description of the Stage 2 VHDL translator assumes that the program has first "passed" the first phase without error. In fact, the second phase is used as the continuation for the first.

4.3 Other Notation and Functions

Fairly standard lambda notation (see [11]) is used in this report, except that structured arguments are permitted in lambda-abstractions. Lambda-abstractions normally have the form \( \lambda x.\text{body} \), where \( \text{body} \) is a lambda-term and \( x \) may be free in \( \text{body} \). The term \( \lambda x.\lambda y.\text{body} \) is printed as \( \lambda x,y.\text{body} \). If \( x \) is, for example, a pair, then the components of \( x \) can be represented in \( \text{body} \) by the application of projection functions to \( x \). Instead, the individual components of \( x \) can be bound to variables \( y \) and \( z \) that appear free in \( \text{body} \) (instead of projection functions applied to \( x \)) by using the abstraction \( \lambda(y,z).\text{body} \). This is defined if and only if the value of \( x \) is indeed a pair. This notation will be used only when its result is defined.

A list is represented in the usual way: \((x,y,z)\). Standard Lisp functions are used, but they are curried, as in \(\text{cons}(x)(y)\) and \(\text{append}(x)(y)\). If \( x \) is a nonempty sequence (list), then \( \text{hd}(x) \) denotes its first element and \( \text{tl}(x) \) the sequence (list) of its remaining components; \( x = \text{cons}(\text{hd}(x))(\text{tl}(x)) \).

Some general-purpose functions are second, third, fourth, fifth, sixth, and last, which return the second, third, fourth, fifth, sixth, and last elements, respectively, of a list. Additionally, we have rest, which returns a list consisting of the first \( n - 1 \) elements of an \( n \)-element list, and length, which returns the integer length of a list.

second\( (x) = \text{hd}(\text{tl}(x)) \)

third\( (x) = \text{hd}(\text{tl}(\text{tl}(x))) \)

fourth\( (x) = \text{hd}(\text{tl}(\text{tl}(\text{tl}(x)))) \)

fifth\( (x) = \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(x)))))) \)
sixth(x) = hd(tl(tl(tl(tl(x))))))

last(id⁺) = (null(tl(id⁺))→ hd(id⁺), last(tl(id⁺))))

rest(id⁺) = (null(tl(id⁺))→ ε, cons(hd(id⁺),rest(tl(id⁺))))

length(x) = (null(x)→ 0, 1+length(tl(x))))
5 Syntax of Stage 2 VHDL

Three Stage 2 VHDL syntaxes are used by the translator: a concrete syntax, which is SLR(1) and is used for parsing Stage 2 VHDL hardware descriptions; and two abstract syntaxes, which are used, respectively, in Phases 1 and 2 of the semantic definition. The concrete syntax is intended to be the "reference" grammar for the Stage 2 VHDL language subset.

In all three syntaxes the syntactic constructs are the members of syntactic domains, which are of two kinds: primitive and compound. The primitive syntactic domains are given. The compound syntactic domains are functions of the primitive domains; these functional dependencies are expressed as a set of syntax equations represented as productions of a context-free grammar. Terminals and nonterminals of this grammar range, respectively, over the primitive and compound syntactic domains. Only those syntactic domains of the abstract syntax that actually appear in a semantic equation will be given explicit names; other syntactic domains will be unnamed, as these names are not used in the specification.

The terminal classes are: identifiers, unsigned decimal numerals, bit literals, character literals, bitstrings (binary, octal, and hexadecimal), and strings. The remaining terminal symbols serve as reserved words.

The concrete syntax of Stage 2 VHDL, being SLR(1), is unambiguous. The abstract syntaxes are considerably smaller than the concrete syntax, because they are not concerned with providing a parsable representation of Stage 2 VHDL, but rather simply provide the minimum syntactic information necessary for a syntax-directed semantic specification. Their use yields a more compact formal definition.

The translation of a hardware description (from concrete syntax) to its abstract syntax representation is accomplished by semantic action routines in the Stage 2 VHDL parser. This process is straightforward, and a formal specification of how the Phase 1 abstract syntax is derived from the concrete syntax is omitted from this report. It is felt that the correspondence between the concrete and Phase 1 syntaxes is so close that no such formal specification is needed. The derivation of Phase 2 syntactic objects from corresponding Phase 1 syntactic objects is explicit in the specification of the interphase abstract syntax tree transformation; see Section 7.

There are some minor variations between the concrete and abstract syntaxes of Stage 2 VHDL. For example, in the concrete syntax, labels for PROCESS statements and loops (LOOP, WHILE, FOR statements) are optional. It was found, however, that the semantics of Stage 2 VHDL requires that every process and loop have a label. Thus in the abstract syntaxes (which drive the semantics), process and loop labels are required. This is enforced by having the parser and the constructor of the Phase 1 abstract syntax tree supply a distinct system-generated label for each unnamed process and loop. These labels are taken from a primitive syntactic domain SysId of system-generated identifiers, disjoint from the primitive syntactic domain Id of identifiers. Similarly, anonymous array types are given distinct system-generated names.

The following subsections present the syntactic domains and equations for Stage 2 VHDL.
5.1 Syntactic Domains

### Primitive Syntactic Domains

- **id**: identifiers
- **SysId**: system-generated identifiers (disjoint from Id)
- **bit**: bit literals
- **constant**: numeric literals (unsigned decimal numerals)
- **char**: character literals
- **bitstring, octstring, hexstring**: bitstring literals
- **string**: string literals

### Compound Syntactic Domains

- **design-file**: design files
- **ent-decl**: entity specifications
- **arch-body**: architecture body specifications
- **port-decl**: port declarations
- **decl, pkg-decl, pkg-body, use-clause**: declarations
- **con-stat**: concurrent statements
- **seq-stat**: sequential statements
- **case-alt**: case alternatives
- **discrete-range**: discrete ranges
- **waveform**: waveforms
- **transaction**: transactions
- **expr**: expressions
- **ref**: references
- **unary-op**: unary operators
- **binary-op**: binary operators
- **relational-op**: relational operators

5.2 Syntax Equations

5.2.1 Concrete Syntax

The concrete syntax for Stage 2 VHDL is shown below.

The productions are numbered for reference purposes. The first production and the nonterminal **start** are inserted by the SLR(1) grammar analyzer to facilitate SLR(1) parsing, and the (terminal) symbol *E* denotes the beginning or end of a file. Terminal symbols appear in upper case letters, while nonterminal symbols and pseudo-terminals (terminals denoting a set of values) are in lower case; pseudo-terminals are prefixed by a “dot” (.).

One slight deviation from official VHDL syntax, necessitated by idiosyncrasies of the Stage 2 VHDL parser, is that we need to use the single reserved word PACKAGEBODY instead of the pair of reserved words PACKAGE BODY. This anomaly will be corrected in future releases of SDVS.
STAGE 2 VHDL CONCRETE SYNTAX

1 **start**
   ::= **E* design-file **E*

2 design-file
   ::= DESIGN_FILE .id IS init pkg-decl-list use-clause-list
      ent-decl arch-body

3 init
   ::= 

4 pkg-decl-list
   ::= 

5 | pkg-decl-list pkg-decl

6 pkg-decl
   ::= PACKAGE .id IS pkg-decl-part END opt-id ;

7 pkg-decl-part
   ::= pkg-decl-item-list

8 pkg-decl-item-list
   ::= 

9 | pkg-decl-item-list pkg-decl-item

10 pkg-decl-item
    ::= const-decl

11 | sig-decl

12 | type-decl

13 use-clause-list
   ::= 

14 | use-clause-list use-clause

15 use-clause
   ::= USE dotted-name-list ;

16 dotted-name-list
   ::= dotted-name

17 | dotted-name-list , dotted-name

18 ent-decl
   ::= ENTITY .id IS ent-header END opt-id ;

19 ent-header
   ::= opt-port-clause

20 opt-port-clause
   ::= 

21 | port-clause

22 opt-id
   ::= 

23 | .id
24 arch-body
    ::= ARCHITECTURE .id OF .id IS decl-part BEGIN
        con-stats END opt-id ;

25 port-clause
    ::= PORT ( port-list ) ;

26 port-list
    ::= interface-list

27 interface-list
    ::= interface-sig-decl
        | interface-list ; interface-sig-decl

29 interface-sig-decl
    ::= opt-signal id-list : opt-mode type-mark opt-init
        | opt-signal id-list : opt-mode slice-name opt-init

31 opt-signal
    ::= SIGNAL

33 id-list
    ::= .id
        | id-list , .id

35 opt-mode
    ::= mode

37 mode
    ::= IN
        | OUT
        | INOUT
        | BUFFER

41 type-mark
    ::= dotted-name

42 dotted-name
    ::= .id
        | dotted-name . .id

44 slice-name
    ::= type-mark ( discrete-range )

46 discrete-range
    ::= range

46 range
    ::= simple-expr direction simple-expr

47 opt-init
    ::= expr
49 direction
   ::= TO
   | DOWNTO

51 decl-part
   ::= decl-item-list

52 decl-item-list
   ::= decl-item-list
   | decl-item-list decl-item

54 decl-item
   ::= object-decl
   | type-decl
   | subprog-decl
   | subprog-body
   | use-clause

59 object-decl
   ::= const-decl
   | var-decl
   | sig-decl

62 const-decl
   ::= CONSTANT id-list : type-mark := expr ;
   | CONSTANT id-list : slice-name := expr ;

64 var-decl
   ::= VARIABLE id-list : type-mark opt-init ;
   | VARIABLE id-list : slice-name opt-init ;

66 sig-decl
   ::= SIGNAL id-list : type-mark opt-init ;
   | SIGNAL id-list : slice-name opt-init ;

68 type-decl
   ::= enum-type-decl
   | array-type-decl

70 enum-type-decl
   ::= TYPE .id IS enum-type-def ;

71 enum-type-def
   ::= ( id-list )
   | ( char-list )

73 char-list
   ::= character-literal
   | char-list , character-literal

75 array-type-decl
   ::= TYPE .id IS array-type-def ;

76 array-type-def
   ::= ARRAY ( discrete-range ) OF type-mark
77 subprog-decl
    ::= subprog-spec ;

78 subprog-spec
    ::= PROCEDURE .id opt-procedure-formal-part
    | FUNCTION .id opt-function-formal-part RETURN type-mark

80 subprog-body
    ::= subprog-spec IS decl-part BEGIN seq-stats END opt-id ;

81 opt-procedure-formal-part
    ::= ( procedure-par-spec-list )

83 opt-function-formal-part
    ::= ( function-par-spec-list )

85 procedure-par-spec-list
    ::= procedure-par-spec
    | procedure-par-spec-list ; procedure-par-spec

87 function-par-spec-list
    ::= function-par-spec
    | function-par-spec-list ; function-par-spec

89 procedure-par-spec
    ::= object-class id-list : procedure-par-mode type-mark

90 function-par-spec
    ::= object-class id-list : function-par-mode type-mark

91 object-class
    ::= CONSTANT
    | VARIABLE
    | SIGNAL

94 procedure-par-mode
    ::= IN
    | OUT
    | INOUT

98 function-par-mode
    ::= IN

100 con-stats
    ::= con-stats con-stat

102 con-stat
    ::= process-stat
    | concurrent-sig-assn-stat
104 process-stat
  ::= opt-unit-label PROCESS decl-part BEGIN seq-stats END
     PROCESS opt-id ;
105   | opt-unit-label PROCESS ( sensitivity-list ) decl-part
     BEGIN seq-stats END PROCESS opt-id ;
106 opt-unit-label
  ::= .id :
107 concurrent-sig-assn-stat
  ::= selected-sig-assn-stat
     | conditional-sig-assn-stat
110 selected-sig-assn-stat
  ::= opt-unit-label WITH expr SELECT target <=
     opt-transport selected-waveforms ;
111 selected-waveforms
  ::= selected-waveform
     | selected-waveforms , selected-waveform
113 selected-waveform
  ::= waveform WHEN choices
114 conditional-sig-assn-stat
  ::= target <= opt-transport conditional-waveforms waveform
     ;
115   | .id : target <= opt-transport conditional-waveforms
     waveform ;
116 conditional-waveforms
  ::= conditional-waveforms conditional-waveform
118 conditional-waveform
  ::= waveform WHEN expr ELSE
119 seq-stats
  ::= seq-stats seq-stat
121 seq-stat
  ::= null-stat
     | var-assn-stat
     | sig-assn-stat
     | if-stat
     | case-stat
     | loop-stat
     | exit-stat
     | return-stat
     | proc-call-stat
     | wait-stat
null-stat ::= NULL ;

var-assn-stat ::= name := expr ;

sig-assn-stat ::= target <= opt-transport waveform ;

opt-transport ::= TRANSPORT

waveform ::= waveform-elt-list

waveform-elt-list ::= waveform-elt |
| waveform-elt-list , waveform-elt

waveform-elt ::= expr |
| expr AFTER expr

if-stat ::= if-head if-tail

if-head ::= IF expr THEN seq-stats |
| if-head ELSIF expr THEN seq-stats

if-tail ::= END IF ; |
| ELSE seq-stats END IF ;

case-stat ::= CASE expr IS case-alt-list END CASE ;

case-alt-list ::= case-alt |
| case-other-alt |
| case-alt case-alt-list

case-alt ::= WHEN choices => seq-stats

case-other-alt ::= WHEN OTHERS => seq-stats

choices ::= choice |
| choices | choice
choice ::= simple-expr

| discrete-range

loop-stat ::= opt-unit-label LOOP seq-stats END LOOP opt-id ;

| opt-unit-label WHILE expr LOOP seq-stats END LOOP opt-id ;

| opt-unit-label FOR name IN discrete-range LOOP seq-stats END LOOP opt-id ;

exit-stat ::= EXIT opt-dotted-name opt-when-cond ;

opt-dotted-name ::= dotted-name

| WHEN expr

return-stat ::= RETURN ;

| RETURN expr ;

proc-call-stat ::= name ;

wait-stat ::= WAIT opt-sensitivity-clause opt-condition-clause opt-timeout-clause ;

eopt-sensitivity-clause ::= sensitivity-clause

| sensitivity-clause

eopt-condition-clause ::= condition-clause

| condition-clause

condition-clause ::= UNTIL expr
opt-timeout-clause
 ::= 
  | timeout-clause

timeout-clause
 ::= FOR expr

expr-list
 ::= expr
  | expr-list, expr

eq
eq
rel
 ::= simple-expr
  | simple-expr relop simple-expr

and-expr
 ::= and-part
  | and-part and-expr

and-part
 ::= AND rel

nand-expr
 ::= nand-part
  | nand-part nand-expr

nand-part
 ::= NAND rel

or-expr
 ::= or-part
  | or-part or-expr

or-part
 ::= OR rel

nor-expr
 ::= nor-part
  | nor-part nor-expr

nor-part
 ::= NOR rel

xor-expr
 ::= xor-part
  | xor-part xor-expr
xor-part
  ::= XOR rel

simple-expr
  ::= simple-expr1
  |  +  simple-expr1
  |  -  simple-expr1

simple-expr1
  ::= term
  |  simple-expr1 addop term

term
  ::= factor
  |  term mulop factor

factor
  ::= primary
  |  primary ** primary
  |  ABS primary
  |  NOT primary

primary
  ::= literal
  |  aggregate
  |  name
  |  ( expr )

literal
  ::= boolean-literal
  |  bit-literal
  |  character-literal
  |  numeric-literal
  |  time-literal
  |  bitstring-literal
  |  string-literal

boolean-literal
  ::= FALSE
  |  TRUE

bit-literal
  ::= .bit

character-literal
  ::= .char

numeric-literal
  ::= .constant

time-literal
  ::= opt-time-constant time-unit

opt-time-constant
  ::=
time-unit ::= FS
          | PS
          | NS
          | US
          | MS
          | SEC
          | MIN
          | HR

bitstring-literal ::= .bitstring
                    | .octstring
                    | .hexstring

string-literal ::= .string

aggregate ::= ( 2-expr-list )

2-expr-list ::= expr , expr
               | 2-expr-list , expr

target ::= name

name ::= namel

namel ::= selector
        | namel . selector
        | namel ( expr-list )

selector ::= .id

relop ::= =
        | /=
        | <
        | <=
        | >
        | >=

addop ::= +
         | -
         | &

mulop ::= *
         | /
         | MOD
         | REM
5.2.2 Abstract Syntax: Phase 1

The abstract syntax of Stage 2 VHDL used during Phase 1 translation is shown below.

The superscript "*" denotes Kleene Closure (e.g. "decl*" denotes zero or more occurrences of the syntactic object "decl"), and a superscript "+" denotes one or more occurrences. In a syntactic clause, subscripts denote (possibly) different objects of the same class.

As in the concrete syntax, terminal symbols appear in upper case, while all other symbols are either nonterminals or pseudo-terminals (id, bitlit, and constant).

**STAGE 2 VHDL ABSTRACT SYNTAX: PHASE 1**

design-file ::= DESIGN-FILE id pkg-decl* pkg-body* use-clause* ent-decl arch-body

pkg-decl ::= PACKAGE id decl* opt-id

pkg-body ::= PACKAGEBODY id decl* opt-id

use-clause ::= USE dotted-name+

dotted-name ::= id+

ent-decl ::= ENTITY id port-decl* decl* opt-id

arch-body ::= ARCHITECTURE id1 id2 decl* con-stat* opt-id

port-decl ::= DEC PORT id+ mode type-mark opt-expr

| SLCDEC PORT id+ mode slice-name opt-expr

mode ::= IN | OUT | INOUT | BUFFER

type-mark ::= dotted-name

slice-name ::= type-mark discrete-range

discrete-range ::= direction expr1 expr2

direction ::= TO | DOWNTO

arch-body ::= ARCHITECTURE id1 id2 decl* con-stat* opt-id

decl ::= object-decl

| type-decl

| pkg-decl

| pkg-body

| subprog-decl

| subprog-body

| use-clause
object-decl ::= DEC object-class id+ type-mark opt-expr
  | SLCDEC object-class id+ slice-name opt-expr

object-class ::= CONST | VAR | SIG

type-decl ::= enum-type-decl
  | array-type-decl

dec enum-type-decl ::= ETDEC id id+
dec array-type-decl ::= ATDEC id discrete-range type-mark

subprog-decl ::= subprogspec

subprogspec ::= PROCEDURE id proc-par-spec* type-mark
  | FUNCTION id func-par-spec* type-mark

proc-par-spec ::= object-class id+ proc-par-mode type-mark opt-expr
func-par-spec ::= object-class id+ func-par-mode type-mark opt-expr

proc-par-mode ::= IN | OUT | INOUT
func-par-mode ::= IN

subprogsbody ::= SUBPROGBODY subprogspec decl* seq-stat* opt-id

con-stat ::= process-stat
  | selected-sig-assn-stat
  | conditional-sig-assn-stat

process-stat ::= PROCESS id ref* decl* seq-stat* opt-id

selected-sig-assn-stat ::= SEL-SIGASSN delay-type id expr ref selected-waveform+
selected-waveform ::= SEL-WAVE waveform discrete-range+

case-stat ::= COND-SIGASSN delay-type id ref cond-waveform waveform
cond-waveform ::= COND-WAVE waveform expr

seq-stat ::= null-stat
  | var-assn-stat
  | sig-assn-stat
  | if-stat
  | case-stat
  | loop-stat
  | while-stat
  | for-stat

30
null-stat ::= NULL

var-assn-stat ::= VARASSN ref expr

sig-assn-stat ::= SIGASSN delay-type ref waveform

delay-type ::= INERTIAL | TRANSPORT

waveform ::= WAVE transaction+
transaction ::= TRANS expr opt-expr

if-stat ::= IF cond-part+ else-part
cond-part ::= expr seq-stat*
else-part ::= seq-stat*

case-stat ::= CASE expr case-alt+
case-alt ::= CASECHOICE discrete-range+ seq-stat*
   | CASEOTHERS seq-stat*

loop-stat ::= LOOP id seq-stat* opt-id

while-stat ::= WHILE id expr seq-stat* opt-id

for-stat ::= FOR id ref discrete-range seq-stat* opt-id

exit-stat ::= EXIT opt-dotted-name opt-expr

call-stat ::= CALL ref

return-stat ::= RETURN opt-expr

wait-stat ::= WAIT ref* opt-expr₁ opt-expr₂

expr ::= ε
   | bool-lit
   | bit-lit
   | num-lit
   | time-lit
   | char-lit
   | bitstr-lit
   | str-lit
ref
| positional-aggregate
| unary-op expr
| binary-op expr₁ expr₂
| relational-op expr₁ expr₂

bool-lit ::= FALSE | TRUE

bit-lit ::= BIT bitlit

num-lit ::= NUM constant

time-lit ::= TIME constant time-unit

char-lit ::= CHAR constant

bitstr-lit ::= BITSTR bit-lit*

str-lit ::= STR char-lit*

ref ::= REF name

ame ::= id
    | name id
    | name expr*

positional-aggregate ::= PAGGR expr*

unary-op ::= NOT | PLUS | NEG | ABS

binary-op ::= AND | NAND | OR | NOR | XOR
    | ADD | SUB | MUL | DIV | MOD | REM | EXP
    | CONCAT

relational-op ::= EQ | NE | LT | LE | GT | GE

time-unit ::= FS | PS | NS | US | MS | SEC | MIN | HR

opt-id ::= ε | id

opt-dotted-name ::= ε | dotted-name

opt-expr ::= ε | expr
5.2.3 Abstract Syntax: Phase 2

The abstract syntax of Stage 2 VHDL used during Phase 2 translation differs in certain respects from that employed by Phase 1, at exactly those points indicated below.

The abstract syntax transformation occurs at the very end of Phase 1, and just prior to the invocation of Phase 2, as described in Section 7.

STAGE 2 VHDL ABSTRACT SYNTAX: PHASE 2

ent-decl ::= ENTITY id decl*1 decl*2 opt-id

con-stat ::= process-stat

process-stat ::= PROCESS id decl* seq-stat* opt-id

expr ::= ε
   | bool-lit
   | bit-lit
   | num-lit
   | time-lit
   | char-lit
   | enum-lit
   | bitstr-lit
   | str-lit
   | ref
   | positional-aggregate
   | unary-op expr
   | binary-op expr₁ expr₂
   | relational-op expr₁ expr₂

time-lit ::= TIME constant FS

enum-lit ::= ENUMLIT id

ref ::= REF modifier+

modifier ::= SREF id+ id
   | INDEX expr
   | SELECTOR id
   | PARLIST expr*

unary-op ::= NOT | BNOT | PLUS | NEG | ABS | RNEG | RABS
binary-op ::= \texttt{AND} \mid \texttt{NAND} \mid \texttt{OR} \mid \texttt{NOR} \mid \texttt{XOR} \\
\quad \mid \texttt{BAND} \mid \texttt{BNAND} \mid \texttt{BOR} \mid \texttt{BNOR} \mid \texttt{BXOR} \\
\quad \mid \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV} \mid \texttt{MOD} \mid \texttt{REM} \mid \texttt{EXP} \\
\quad \mid \texttt{RPLUS} \mid \texttt{RMINUS} \mid \texttt{RTIMES} \mid \texttt{RDIV} \mid \texttt{REXPT} \\
\quad \mid \texttt{CONCAT}

relational-op ::= \texttt{EQ} \mid \texttt{NE} \mid \texttt{LT} \mid \texttt{LE} \mid \texttt{GT} \mid \texttt{GE} \\
\quad \mid \texttt{RLT} \mid \texttt{RLE} \mid \texttt{RGT} \mid \texttt{RGE}
6 Phase 1: Static Semantic Analysis and Environment Collection

Now that the necessary background has been established, we are ready to examine the formal description of the Stage 2 VHDL translator.

In this section, an overview of Phase 1 and its relation to Phase 2 will be presented, followed by detailed discussions of the environment manipulated by the translator and the Phase 1 semantic domains and function types, and finally the Phase 1 semantic equations themselves.

6.1 Overview

A Stage 2 VHDL hardware description is first parsed according to the Stage 2 VHDL concrete grammar, producing an abstract syntax tree that serves as the input to Phase 1 translation.

Phase 1 of the translation accomplishes the following.

- Performs static semantic checks to verify that certain conditions are met, including:
  - Objects, subprograms, packages, and process and loop labels must be declared prior to use.
  - Identifiers with the same name cannot be declared in the same local context.
  - References to objects and labels must be proper, e.g. scalar objects must not be indexed, array references must have the correct number of indices, and EXIT statements must reference a loop label.
  - All components of statements and expressions must have the proper type, e.g. expressions used as conditions must be boolean, array indices must be of the proper type, operators must receive operands of the correct type, procedure and function calls must receive actual parameters of the proper type, function calls must return a result of a type appropriate for their use in an expression.
  - Sensitivity lists in PROCESS and WAIT statements must contain signal identifiers.
  - The collection of discrete ranges defining a CASE statement alternative must be exhaustive and mutually exclusive.
  - The time delays in the AFTER clause of a signal assignment statement must be increasing.

- Creates a new abstract syntax tree — a transformed version of the original abstract syntax tree (used by Phase 1) — that will be more conveniently utilized by Phase 2 of the translation.

- Creates and manipulates a tree-structured environment (TSE) that, in the absence of errors, is provided to Phase 2 of the translation.
If the VHDL translator completes Phase 1 without error, then it can proceed with Phase 2, *state delta generation*. Phase 2 requires two inputs: the transformed abstract syntax tree and the tree-structured environment for the hardware description, both constructed by Phase 1.

The tree-structured environment contains a complete record of the name/attribute associations corresponding to the hardware description’s declarations, and its structure reflects that of the description. Referring to this TSE, Phase 2 incrementally generates and (per user proof commands) applies state deltas via symbolic execution and the theories built into the Simplifier.

### 6.2 Descriptors, Types, and Type Modes

When a declaration of an identifier is processed by Phase 1, that identifier is bound in the TSE to a *descriptor*, a structured object that contains the attributes of the identifier instance associated to it by that declaration.

At the time a descriptor is created and entered into the TSE, its *qid* field is set to *e*. The value of the *qid* field is eventually set to the proper *statically uniquely qualified name (SUQN)*, when such a qualified name makes sense; see Section 8.2.1. These updates to the *qid* fields become possible only once the TSE is fully constructed, i.e., at the very end of Phase 1 — or in other words, at the very beginning of Phase 2, the phase in which these uniquely qualified names are needed.

Eight kinds of descriptor are used in Phase 1: *object, package, process name, loop name, function, procedure, enumeration type element, and type*. Their structures are as follows:

**object**:

```plaintext
< id, qid, tag, path, exported, type, value, process >
```

The *id* field contains the identifier to which this descriptor is bound, and the *qid* field contains its *statically uniquely qualified name (SUQN)*. The *tag* field contains *OBJECT*#. The *path* field contains the path in the tree-structured environment to the component environment in which this instance of the identifier is bound. The *exported* field indicates whether the definition of this identifier instance can be exported to other environments. A value *true* (represented by DENOTE symbol *tt*) indicates exportation is permitted, and a value *false* (represented by DENOTE symbol *ff*) indicates that it is not. This becomes an issue when the declaration whose elaboration created this descriptor was contained in a package specification (exportable) or package body (not exportable).

If the identifier *id* represents a constant initialized via a *static* expression, then the *value* field contains the initial value; otherwise it contains *UNDEF* (undefined). Array and record references never represent static values in VHDL, so the *value* field of corresponding object descriptors contains *UNDEF*.

If the identifier *id* represents a signal, then the label of the first *PROCESS* statement in which *id* is the target of a signal assignment is entered into the *process* field, to
enable the detection of assignments to the signal by multiple processes (disallowed in Stage 2 VHDL).

Finally, the object descriptor’s type field contains the type of the identifier, represented by a pair \(<\textit{tmode}, \textit{tdesc} >:\)

- \textit{tmode} is the \textit{type mode}, itself a pair;
  normally,
  \[ \textit{tmode} = <\textit{object-class, ref-mode}>, \]
where \(\textit{object-class} \in \{\text{CONST, VAR, SIG}\}\)
and \(\textit{ref-mode} \in \{\text{VAL, REF, OUT}\}\).
The \textit{tmode} indicates, first, whether the object is a constant (\(\textit{object-class} = \text{CONST}\)), variable (\(\textit{object-class} = \text{VAR}\)), or signal (\(\textit{object-class} = \text{SIG}\)), and second, whether the object is read-only (\(\textit{ref-mode} = \text{VAL}\)), read-write (\(\textit{ref-mode} = \text{REF}\)), or write-only (\(\textit{ref-mode} = \text{OUT}\)).
For technical purposes, it is also occasionally convenient for Phase 1 translation to manipulate “dummy” type modes of the form \(<\text{DUMMY, VAL}>,\)
\(<\text{DUMMY, OBJ}>, <\text{DUMMY, ACC}>, <\text{DUMMY, AGR}>,\) and
\(<\text{DUMMY, TYP}>,\) as well as “path” type modes of the form \(<\text{PATH, p} >\)
where \(p\) is a path in the TSE.
- \textit{tdesc} is the \textit{type descriptor} (see below). It gives the object’s \textit{basic type}, irrespective of the type mode.

To introduce a bit more terminology, a type in which the \textit{ref-mode} is \text{REF} or \text{OUT} will be called a \textit{reference type}, while one whose \textit{ref-mode} is \text{VAL} will be called a \textit{value type}. A reference type indicates that the associated object can have its value altered (by an assignment, say), as opposed to a value type.

Finally, the type descriptor \(d = \textit{tdesc}\) is the \textit{basic type} of the type \(<\textit{tmode}, \textit{tdesc} >\) of which it is the second component.

**package :**
\[ <\textit{id}, \textit{qid}, \text{*PACKAGE*}, \textit{path}, \text{exported}, \textit{pbody} > \]
The \textit{id}, \textit{qid}, \textit{path}, and \text{exported} fields are as above. The tag field contains \text{*PACKAGE*}. If this package has a body, the \textit{pbody} field contains the transformed abstract syntax tree of the package body; otherwise it contains \(\varepsilon\).

**process name :**
\[ <\textit{id}, \textit{qid}, \text{*PROCESSNAME*}, \textit{path} > \]
The \textit{id} and \textit{path} fields are as above. The tag field contains \text{*PROCESSNAME*} (the process label). The \textit{qid} field has no relevance here, and contains \(\varepsilon\).

**loop name :**
\[ <\textit{id}, \textit{qid}, \text{*LOOPNAME*}, \textit{path} > \]
The \textit{id}, \textit{qid}, and \textit{path} fields are as in a process name. The tag field contains \text{*LOOPNAME*} (the loop label).
function:

< id, qid, *FUNCTION*, path, exported, signatures, body, characterizations >

The id, qid, exported, and path fields are as above. The tag field contains *FUNCTION*.

The signatures field contains a list of signatures, that is, < pars, rtype > pairs; this list will be a singleton unless the function is overloaded. The pars field of a signature is a list that indicates the names and types of the function’s formal parameters. Each list element is a pair, whose first component is the identifier that denotes the formal parameter’s name and whose second component is its type. The rtype (result type) field of a signature contains the type of the function’s result for these particular parameter types; this type is always a value type.

The body field of a function descriptor contains the transformed abstract syntax tree of the function’s body (including its local declarations) if a body exists, and ε otherwise. The characterizations field of a function descriptor always contains ε (see procedure descriptors for a description of this field).

procedure:

< id, qid, *PROCEDURE*, path, exported, signatures, body, characterizations >

The id, qid, path, exported, signatures, body, and characterizations fields are as in the function descriptor. The tag field contains *PROCEDURE* (procedure). Since procedures return no result, all rtype fields in each signature contain the void standard value type (see below).

The characterizations field of a procedure descriptor, unlike that of a function descriptor, is potentially nonempty. One of either the body or the characterizations must contain ε; either a procedure has a body that may be symbolically executed, or it has been characterized by a set of state deltas. In fact, Stage 2 VHDL does not yet support offline characterizations for procedures ([16], [17]), but we expect that this facility will be incorporated in Stage 3 VHDL.

A characterization is a 6-tuple containing the following information:

- the path to the procedure;
- the identifier that names the procedure;
- a list of the identifiers that name the arguments to the procedure;
- a (possibly empty) precondition that determines under which conditions this characterization may be used;
- a modification list of the names of variables changed by this procedure; and
- a postcondition that states the effects of the procedure.

The last three items in the tuple must be given in SDVS internal state delta notation, as they form the basis for a state delta that characterizes the actions of the procedure.
enumeration type element:

< id, qid, *ENUMELT*, path, exported, type >

The id field contains the name of an enumeration type element, the tag field is *ENUMELT*, and the type field contains the descriptor of the enumeration type.

type:

There are four kinds of type descriptor: those for standard types, enumeration types, array types, and record types. Although record types are not actually incorporated in the Stage 2 VHDL language subset, the Stage 2 VHDL translator contains support for their eventual implementation in Stage 3 VHDL.

Each type descriptor has an id field (containing the name of that type), a corresponding qid field, a tag field (indicating the kind of type descriptor), path and exported fields (that serve the usual purpose), and additional fields that contain information appropriate to the type represented by the descriptor. The detailed structures of the type descriptors are as follows:

standard type:

< id, qid, tag, path, exported >

Standard types are those considered to be predeclared; they are always exportable. In Stage 2 VHDL, the standard types are boolean, bit, integer, real, time, character, bit_vector, and string; they cannot be redeclared.

The id and tag fields denote the following Stage 2 VHDL standard types:

\[
\begin{align*}
\text{id} &= \text{BOOLEAN}, & \text{tag} &= *\text{BOOL}^* \\
\text{id} &= \text{BIT}, & \text{tag} &= *\text{BIT}^* \\
\text{id} &= \text{INTEGER}, & \text{tag} &= *\text{INT}^* \\
\text{id} &= \text{REAL}, & \text{tag} &= *\text{REAL}^* \\
\text{id} &= \text{TIME}, & \text{tag} &= *\text{TIME}^* \\
\text{id} &= \text{BIT\_VECTOR}, & \text{tag} &= *\text{ARRAYTYPE}^* \\
\text{id} &= \text{STRING}, & \text{tag} &= *\text{ARRAYTYPE}^* \\
\end{align*}
\]

For completeness, we also provide a void and a polymorphic standard type for Stage 2 VHDL:

\[
\begin{align*}
\text{id} &= \text{VOID}, & \text{tag} &= *\text{VOID}^* \\
\text{id} &= \text{POLY}, & \text{tag} &= *\text{POLY}^* \\
\end{align*}
\]
**enumeration type**:  

\[ < \text{id}, \text{qid}, \text{*ENUMTYPE*}, \text{path}, \text{exported}, \text{literals} > \]

The **literals** field is a nonempty list of identifiers giving the enumeration literals (in order) for this type. Both characters and identifiers are admissible enumeration literals in Stage 2 VHDL.

**array type**:  

\[ < \text{id}, \text{qid}, \text{*ARRAYTYPE*}, \text{path}, \text{exported}, \text{direction}, \text{lb}, \text{ub}, \text{elty} > \]

Every array type has a name; unique names are generated for anonymous array types. Arrays in Stage 2 VHDL are **one-dimensional**, of **INTEGER** index type. The **direction** field contains either **TO** or **DOWNTO**, indicating whether the indices of the array increase or decrease, respectively. The **lb** and **ub** fields contain, respectively, abstract syntax trees for expressions that denote the array type's lower and upper bounds. The **elty** (element type) field contains the descriptor of the type of the array's elements. The values of the array's lower and upper bounds are not necessarily static; therefore, array bounds-checking generally cannot be performed in Phase 1, but must be deferred to Phase 2 ("run time"), when state deltas are applied ("executed").

**record type**:  

\[ < \text{id}, \text{qid}, \text{*RECORDTYPE*}, \text{path}, \text{exported}, \text{components} > \]

The **components** field is a nonempty list of triplets; each triplet represents a field of this record type. The first element of each triplet is an identifier that is this field's name. The second element is a descriptor representing this field's basic type. The third element either is empty or contains an abstract syntax tree for Phase 2 initialization for components of objects declared to be of this record type. As noted above, records are not implemented as part of Stage 2 VHDL, and record types are included simply in preparation for the anticipated implementation of records in Stage 3 VHDL.

Constant functions returning type descriptors are:

\begin{align*}
\text{bool-type-desc()} &= < \text{BOOLEAN}, \epsilon, \text{*BOOL*}, \text{(STANDARD)}, \text{tt}> \\
\text{bit-type-desc()} &= < \text{BIT}, \epsilon, \text{*BIT*}, \text{(STANDARD)}, \text{tt}> \\
\text{int-type-desc()} &= < \text{INTEGER}, \epsilon, \text{*INT*}, \text{(STANDARD)}, \text{tt}> \\
\text{real-type-desc()} &= < \text{REAL}, \epsilon, \text{*REAL*}, \text{(STANDARD)}, \text{tt}> \\
\text{time-type-desc()} &= < \text{TIME}, \epsilon, \text{*TIME*}, \text{(STANDARD)}, \text{tt}> \\
\text{void-type-desc()} &= < \text{VOID}, \epsilon, \text{*VOID*}, \text{(STANDARD)}, \text{tt}> \\
\text{poly-type-desc()} &= < \text{POLY}, \epsilon, \text{*POLY*}, \text{(STANDARD)}, \text{tt}> \\
\end{align*}
The function `char-type-desc` looks up the descriptor for standard type `CHARACTER` which Phase 1 will have entered in the `t((STANDARD))` component environment of the TSE `t`:

```plaintext
char-type-desc(t) = t((STANDARD))(CHARACTER )
```

In addition, the following function accepts arguments to create an appropriate array type:

```plaintext
array-type-desc(array-name,qid,path,exported,direction,lower-bound,upper-bound,element-type) = <array-name,qid,*ARRAYTYPE*,path,exported,direction,lower-bound,upper-bound,element-type>
```

In particular, the `BIT_VECTOR` and `STRING` standard types are created by calls to `array-type-desc`:

```plaintext
bitvector-type-desc() = array-type-desc(BIT_VECTOR )((STANDARD))(tt)(TO )((NUM 0))(bit-type-desc())
```

```plaintext
string-type-desc(t) = array-type-desc(STRING)(t)((STANDARD))(tt)(TO )((NUM 1))(char-type-desc(t))
```

Predicates are available for distinguishing specific types and type descriptors:

```plaintext
is-boolean?(type) = is-boolean-tdesc?(tdesc(type))
```

```plaintext
is-boolean-tdesc?(d) = idf(d) = BOOLEAN
```

```plaintext
is-bit?(type) = is-bit-tdesc?(tdesc(type))
```

```plaintext
is-bit-tdesc?(d) = idf(d) = BIT
```

```plaintext
is-integer?(type) = is-integer-tdesc?(tdesc(type))
```

```plaintext
is-integer-tdesc?(d) = idf(d) = INTEGER
```

```plaintext
is-real?(type) = is-real-tdesc?(tdesc(type))
```

```plaintext
is-real-tdesc?(d) = idf(d) = REAL
```

```plaintext
is-time?(type) = is-time-tdesc?(tdesc(type))
```

```plaintext
is-time-tdesc?(d) = idf(d) = TIME
```

```plaintext
is-void?(type) = is-void-tdesc?(tdesc(type))
```

```plaintext
is-void-tdesc?(d) = idf(d) = VOID
```
is-poly?(type) = is-poly-tdesc?(tdesc(type))

is-poly-tdesc?(d) = idf(d) = POLY

is-character?(type) = is-character-tdesc?(tdesc(type))

is-character-tdesc?(d) = idf(d) = CHARACTER

is-array?(type) = is-array-tdesc?(tdesc(type))

is-array-tdesc?(d) = tag(d) = *ARRAYTYPE*

is-record?(type) = is-record-tdesc?(tdesc(type))

is-record-tdesc?(d) = tag(d) = *RECORDTYPE*

is-bitvector?(type) = is-bitvector-tdesc?(tdesc(type))

is-bitvector-tdesc?(d)
= let idf = idf(d) in
  idf = BIT-VECTOR ∨ (consp(idf) ∧ hd(idf) = BIT-VECTOR )

is-string?(type) = is-string-tdesc?(tdesc(type))

is-string-tdesc?(d)
= let idf = idf(d) in
  idf = STRING ∨ (consp(idf) ∧ hd(idf) = STRING )

is-const?(type) = object-class(tmode(type)) = CONST

is-var?(type) = object-class(tmode(type)) = VAR

is-sig?(type) = object-class(tmode(type)) = SIG

Certain primitive functions can be applied to descriptors. For each kind of descriptor and
field there exists an access function, ordinarily with the same name as the field (the only
exception being idf instead of id). When applied to a descriptor of the proper kind, the
access function extracts the contents of that descriptor’s corresponding field. For example,
if d is an object descriptor, then tag(d) = *OBJECT*.

If d is any descriptor, then the fully qualified name of the corresponding identifier instance
is returned by function namef:

namef(d) = $(path(d))(idf(d))
Defined below are the descriptor component access functions, a few related constructor and access functions, and some convenient additional predicates.

\[
\begin{align*}
\text{idf}(d) &= \text{hd}(d) \\
\text{qid}(d) &= \text{hd}(\text{tl}(d)) \\
\text{tag}(d) &= \text{hd}(\text{tl}(\text{tl}(d))) \\
\text{path}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(d)))) \\
\text{exported}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{type}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{value}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{process}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d))))))) \\
\text{body}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{signatures}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{body}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{characterizations}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{direction}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{lb}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{ub}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{elty}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{components}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{literals}(d) &= \text{hd}(\text{tl}(\text{tl}(\text{tl}(\text{tl}(d)))))) \\
\text{pars}(\text{signature}) &= \text{hd}(\text{signature}) \\
\text{rtype}(\text{signature}) &= \text{hd}(\text{tl}(\text{signature}))
\end{align*}
\]

\[
\begin{align*}
\text{mk-type}(\text{tmode})(\text{tdesc}) &= (\text{tmode}, \text{tdesc}) \\
\text{tmode}(\text{type}) &= \text{hd}(\text{type}) \\
\text{tdesc}(\text{type}) &= \text{hd}(\text{tl}(\text{type}))
\end{align*}
\]

\[
\begin{align*}
\text{mk-tmode}(\text{object-class})(\text{ref-mode}) &= (\text{object-class}, \text{ref-mode}) \\
\text{object-class}(\text{tmode}) &= \text{hd}(\text{tmode}) \\
\text{ref-mode}(\text{tmode}) &= \text{hd}(\text{tl}(\text{tmode}))
\end{align*}
\]

\[
\begin{align*}
\text{is-const?}(\text{type}) &= \text{object-class}(\text{tmode}(\text{type}))= \text{CONST} \\
\text{is-var?}(\text{type}) &= \text{object-class}(\text{tmode}(\text{type}))= \text{VAR} \\
\text{is-sig?}(\text{type}) &= \text{object-class}(\text{tmode}(\text{type}))= \text{SIG}
\end{align*}
\]

\[
\begin{align*}
\text{is-readable?}(\text{type}) &= \text{ref-mode}(\text{tmode}(\text{type}))\in (\text{VAL REF}) \\
\text{is-writable?}(\text{type}) &= \text{ref-mode}(\text{tmode}(\text{type}))\in (\text{REF OUT})
\end{align*}
\]

\[
\begin{align*}
\text{is-ref?}(\text{expr}) &= \text{hd}(\text{expr})= \text{REF} \\
\text{is-paggr?}(\text{expr}) &= \text{hd}(\text{expr})= \text{PAGGR}
\end{align*}
\]

\[
\begin{align*}
\text{is-unary-op?}(\text{op}) &= \text{op} \in (\text{NOT PLUS NEG ABS}) \\
\text{is-binary-op?}(\text{op}) &= \text{op} \in (\text{AND NAND OR NOR XOR ADD SUB MUL DIV MOD REM EXP CONCAT}) \\
\text{is-relational-op?}(\text{op}) &= \text{op} \in (\text{EQ NE LT LE GT GE})
\end{align*}
\]
6.3 Special-Purpose Environment Components and Functions

Certain component environments \( r \) of the tree-structured environment (TSE) part of the translation state have special identifier-like names that are bound to values specific to that environment’s associated program unit (design file, package, entity, architecture, process, procedure, function, or loop):

\textbf{*UNIT*} :
\( r(*\text{UNIT*}) \) contains a tag that identifies what kind of program unit led to the creation of \( r \). These tags are \texttt{*DESIGN-FILE*} (design file), \texttt{*PACKAGE*} (package), \texttt{*ENTITY*} (entity), \texttt{*ARCHITECTURE*} (architecture), \texttt{*PROCESS*} (process), \texttt{*PROCEDURE*} (procedure), \texttt{*FUNCTION*} (function), and \texttt{*LOOP*} (loop). These tags are used to locate the innermost instance of a specific kind of environment (such as one associated with a process) on the current lookup path in the TSE.

\textbf{*LAB*} :
When the tag of \( r(*\text{UNIT*}) \) is \texttt{*ARCHITECTURE*}, the value bound to \( r(*\text{LAB*}) \) contains an identifier list of all the process labels declared in the program unit. When the tag of \( r(*\text{UNIT*}) \) is \texttt{*PROCESS*}, \texttt{*PROCEDURE*}, \texttt{*FUNCTION*}, or \texttt{*LOOP*}, the value bound to \( r(*\text{LAB*}) \) contains an identifier list of all the loop labels declared in the program unit. These lists are used to ensure that the identifiers serving as process and loop labels are distinct in (the top-level scope of) each program unit.

\textbf{*USED*} :
The environment corresponding to any program unit admitting USE clauses in its declarative part has a \texttt{*USED*} component. In this case, \( r(*\text{USED*}) \) is a list representing the set of fully qualified names of packages named in USE clauses appearing in that declarative part, omitting the qualified names of packages that textually enclose those USE clauses. In order to ensure that the TSE used in Phase 2 of the Stage 2 VHDL translator can remain fixed as that generated by Phase 1, a slight restriction is imposed on the concrete syntax of Stage 2 VHDL. This restriction requires that all of the USE clauses in a declarative part appear only at the end of that declarative part. This will be discussed more fully later.

\textbf{*IMPT*} :
Whenever a program unit has a \texttt{*USED*} component, it also has a \texttt{*IMPT*} component. \( r(*\text{IMPT*}) \) is a list of the fully qualified names of those items that can be imported into the program unit’s environment by the elaboration of the USE clauses in its declarative part. Consequently, no two of these fully qualified names can have the same last identifier (unqualified name), nor can the last identifier of any of these fully qualified names be the same as an identifier whose (local) declaration appears in this program unit’s declarative part.

\textbf{*SENS*} :
When the tag of \( r(*\text{UNIT*}) \) is \texttt{*PROCESS*}, the value bound to \( r(*\text{SENS*}) \) con-
tains a list of the transformed abstract syntax trees of the refs appearing in that process' sensitivity list. Phase 1 translation of a WAIT statement occurring in a PROCESS statement checks to make sure this *SENS* list is empty; otherwise, the WAIT occurs illegally in a process with a sensitivity list.

Special Phase 1 Functions

Three special-purpose Phase 1 functions defined by SDVS are set-difference, new-array-type-name, and delete-duplicates; these are provided by SDVS because of the difficulty of writing their definitions in the DENOTE language (DL).

Function set-difference returns the set difference of two lists. Function new-array-type-name returns a new unique name for an anonymous array type. Function delete-duplicates destructively deletes duplicate items from a list.

Error Reporting

Phase 1 errors are reported by three SDVS functions: error, which takes a string-valued error message; error-pp, which takes a string-valued error message and an additional VHDL abstract syntax subtree to be pretty-printed; and cat, which makes a string from its (variable number of) arguments, each of which is made into a string.

6.4 Phase 1 Semantic Domains and Functions

The formal description of Phase 1 translation consists of semantic domains and semantic functions, the latter being functions from syntactic to semantic domains. Compound semantic domains are defined in terms of primitive semantic domains. Similarly, primitive semantic functions are unspecified (their definitions being understood implicitly) and the remaining semantic functions are defined (by syntactic cases) via semantic equations.

The principal Phase 1 semantic functions (and corresponding Stage 2 VHDL language constructs for which they perform static analysis) are: DFT (design files), ENT (entity declarations), ART (architecture bodies), PDT (port declarations), DT (declarations), CST (concurrent statements), SLT (sensitivity lists), SST (sequential statements), AT (case alternatives), DRT (discrete ranges), WT (waveforms), TRT (transactions), ET and RT (expressions), OT1 (unary operators), OT2 (binary and relational operators), B (bit literals), and N (numeric literals).

Each of the principal semantic functions requires an appropriate syntactic argument — an abstract syntactic object (tree) generated by the Stage 2 VHDL language parser. Most of the semantic functions take (at least) the following additional arguments:

- a path, indicating the currently visible portion of the (partially constructed) tree-structured environment;
- a continuation, specifying which Phase 1 semantic function to invoke next;
• a (partially constructed) TSE, containing the information gathered from declarations previously elaborated and checked.

In the absence of errors, the Phase 1 semantic functions update the TSE. Moreover, ET and RT also construct a pair consisting of an expression's type and its static value. The type is either a value type or a reference type; see Section 6.2. Only an expression with a reference type may be the target of an assignment operation.

An expression's static value is *UNDEF* ("undefined") unless it is a static expression, in which case its static value is determined as follows. A static expression is:

- a boolean, bit, numeric, or character literal: the static value is the value of the corresponding constant;
- an identifier explicitly declared as a scalar constant and initialized by a static expression: the static value is the static value of the initialization expression;
- an operator applied to operands that are static expressions: the static value is determined by the semantics of the operator and the static value of the operands;
- a static expression enclosed in parentheses: the static value is the static value of the enclosed static expression.

Note that a subscripted array reference, even if the subscript is a static expression and the array was declared as a constant initialized with a list of static expressions, is not a static expression. (The same is true for a selected record component.)

Figures 1 and 2 depict, respectively, the semantic domains and function types for Phase 1 of the Stage 2 VHDL translator.
**Primitive Semantic Domains**

- **Bool** = \{FALSE, TRUE\}  
  boolean constants
- **Bit** = \{0, 1\}  
  bit constants
- **Char** = \{(CHAR 0), \ldots, (CHAR 127)\}  
  character constants (ASCII-128 representations)
- **n : N** = \{0, 1, 2, \ldots\}  
  numeric constants (natural numbers)
- **id : Id**  
  identifiers
- **SysId**  
  system-generated identifiers (disjoint from Id)
- **t : TEnv**  
  tree-structured environments (TSEs)
- **d : Desc**  
  descriptors (see Section 6.2)
- **sd : SD**  
  state deltas
- **Assert**  
  SDVS Simplifier assertions
- **Error**  
  error messages

**Compound Semantic Domains**

- **elbl : Elbl** = Id + SysId  
  TSE edge labels
- **p, q: Path = Elbl^***  
  TSE paths
- **qname: Name = Elbl (. Elbl)^***  
  qualified names
- **d : Dv = Desc**  
  denotable values (descriptors)
- **r : Env = Id \rightarrow (Dv + \{*UNBOUND*\})**  
  environments
- **Tmode = \{PATH\} \times Id^* + \{(CONST, VAR, SIG, DUMMY) \times \{VAL, OUT, REF, OBJ, ACC, TYPI\}\}**  
  type modes
- **w : Type = Tmode \times Desc**  
  types
- **e : Value**  
  values
- **h : CSet = P(Bool) + P(Char) + P_f(N) + \{INT\} + \{ENUM\}**  
  case selection sets [P(\bullet) denotes "powerset of" and P_f(\bullet) denotes "set of finite subsets of"]
- **u : TDc = TEnv \rightarrow Ans**  
  declaration & concurrent statement continuations
- **c : TSc = TDc**  
  sequential statement continuations
- **k : TEc = (Type \times Value) \rightarrow TSc**  
  expression continuations
- **y : TAc = CSet \rightarrow TSc**  
  case alternative continuations
- **v : TTc = Type \rightarrow Ans**  
  type continuations
- **z : Desc \rightarrow TDc**  
  descriptor continuations
- **Ans = (SD + Assert)^* + Error**  
  final answers

Figure 1: Phase 1 Semantic Domains
DFT : Design → Ans
ENT : Ent → Path → TDC → TDC
ART : Arch → Path → TDC → TDC
PDT : PDec* → Path → Bool → TDC → TDC
DT : Dec* → Path → Bool → TDC → TDC
CST : CStat* → Path → TDC → TDC
SLT : Ref* → Path → TDC → TDC
SST : SStat* → Path → TSc → TSc
AT : Alt* → Type → Path → TAc → TSc
DRT : Drg → Type → Path → TAc → TSc
WT : Wave → Path → TEC → TSc
TRT : Trans* → Path → TEC → TSc
ET : Expr → Path → TEC → TSc
RT : Expr → Path → TEC → TSc
OT1 : Uop → TEC → TEC
OT2 : Bop → TEC → (Type × Value) → TEC
B : BitLit → Bit
N : NumLit → N

design file static semantics
entity declaration static semantics
architecture body static semantics
port declaration static semantics
declaration static semantics
concurrent statement static semantics
sensitivity list static semantics
sequential statement static semantics
case alternative static semantics
discrete range static semantics
waveform static semantics
transaction static semantics
expression static semantics
expression static semantics
unary operator static semantics
binary, relational operator static semantics
bit values of bit literals (primitive)
integer values of numeric literals (primitive)

Figure 2: Phase 1 Semantic Functions
6.5 Phase 1 Semantic Equations

6.5.1 Stage 2 VHDL Design Files

(DFT1) DFT \[[ DESIGN-FILE id pkg-decl* pkg-body* use-clause* ent-decl arch-body ]\]

\[= \text{let } t_0 = \text{mk-initial-tse()}\]
\[\text{and } p_0 = \%(e)(id) \text{ in}\]
\[\text{let } id_1 = \text{hd}(\text{ul}(\text{ent-decl})) \text{ in}\]
\[\text{let } t_1 = \text{enter-standard}(t_0)\]
\[\text{and } p_1 = \%(p_0)(id_1) \text{ in}\]
\[\text{let } t_2 = \text{enter}\]
\[
(t_1)(e)(id)\]
\[<(e,^*\text{DESIGN-FILE}^*,e,tt,\]
\[(e,(\text{VAL },\text{void-type-desc()})>,e,e)>\text{ in}\]
\[\text{let } t_3 = \text{enter}(\text{extend}(t_2)(e)(id))(p_0)(*\text{UNIT* }<(e,^*\text{DESIGN-FILE}^*>) \text{ in}\]
\[\text{let } t_4 = \text{enter}(t_3)(p_0)(*\text{LAB* }<(e,e)> \text{ in}\]
\[\text{let use-clause = } (\text{USE, }((\text{STANDARD } \text{, ALL } )))\text{ in}\]
\[\text{DT [ use-clause ] } (e)(t)(u_1)(t_6)\]
\[\text{where } u_1 = \text{At. DT [ pkg-decl* ] } (p_0)(t)(u_2)(t)\]
\[\text{where } u_2 = \text{At. DT [ pkg-body* ] } (p_0)(t)(u_3)(t)\]
\[\text{where } u_3 = \text{At. DT [ use-clause* ] } (p_0)(t)(u_4)(t)\]
\[\text{where } u_4 = \text{At. ENT [ ent-decl ] } (p_0)(u_5)(t)\]
\[\text{where } u_5 = \text{At. ART [ arch-body ] } (p_1)(u_6)(t)\]
\[\text{where}\]
\[u_6 = \text{At. let unit = DFX [ design-file ] (t) in}\]
\[\text{phase2(unit)(t) in}\]

enter-standard(t)\]
\[= \text{let } t_1 = \text{enter-package}(t)(e)(\text{STANDARD } ) \text{ in}\]
\[\text{let } t_2 = \text{enter-package}(t_1)(e)(\text{TEXTIO } ) \text{ in}\]
\[\text{let } t_3 = \text{enter}(t_2)(e)(\text{USED* })(<e,e>) \text{ in}\]
\[\text{let } t_4 = \text{enter}(t_3)(e)(\text{IMPT* })(<e,e>) \text{ in}\]
\[\text{let use-clause = } (\text{USE, }((\text{STANDARD } , \text{ALL } )))\text{ in}\]
\[\text{DT [ use-clause ] } (e)(t)(u_1)(t_6)\]
\[\text{where } u_1 = \text{At. DT [ pkg-decl* ] } (p_0)(t)(u_2)(t)\]
\[\text{where } u_2 = \text{At. DT [ pkg-body* ] } (p_0)(t)(u_3)(t)\]
\[\text{where } u_3 = \text{At. DT [ use-clause* ] } (p_0)(t)(u_4)(t)\]
\[\text{where } u_4 = \text{At. ENT [ ent-decl ] } (p_0)(u_5)(t)\]
\[\text{where } u_5 = \text{At. ART [ arch-body ] } (p_1)(u_6)(t)\]
\[\text{where}\]
\[u_6 = \text{At. let unit = DFX [ design-file ] (t) in}\]
\[\text{phase2(unit)(t) in}\]

enter-package(t)(p)(id)\]
\[= \text{let } p_1 = \%(p)(id) \text{ in}\]
\[\text{let package-desc = } <e,^*\text{PACKAGE* },p,tt,e> \text{ in}\]
\[\text{let } t_1 = \text{enter}(t)(p)(id)(package-desc) \text{ in}\]
\[\text{let } t_2 = \text{enter}(\text{extend}(t_1)(p)(id))(p_1)(*\text{UNIT* }<(e,^*\text{PACKAGE* }>) \text{ in}\]
\[\text{let } t_3 = \text{enter}(t_2)(p_1)(*\text{USED* })(<e,e>) \text{ in}\]
\[\text{let } t_4 = \text{enter}(t_3)(p_1)(*\text{IMPT* })(<e,e>) \text{ in}\]
\[\text{let } t_5 = \text{enter-predefined}(t_4)((\text{STANDARD}) ) \text{ in}\]
\[t_5\]

enter-predefined(t)(p)\]
\[= \text{let } t_1 = \text{enter}(t)(p)(\text{BOOLEAN })(t)(\text{bool-type-desc()}) \text{ in}\]
\[\text{let } t_2 = \text{enter}(t_1)(p)(\text{BIT })(t)(\text{bit-type-desc()}) \text{ in}\]
\[\text{let } t_3 = \text{enter}(t_2)(p)(\text{INTEGER })(t)(\text{int-type-desc()}) \text{ in}\]
\[\text{let } t_4 = \text{enter}(t_3)(p)(\text{REAL })(t)(\text{real-type-desc()}) \text{ in}\]
\[\text{let } t_5 = \text{enter}(t_4)(p)(\text{TIME })(t)(\text{time-type-desc()}) \text{ in}\]
\[\text{let } t_6 = \text{enter}(t_5)(p)(\text{VOID })(t)(\text{void-type-desc()}) \text{ in}\]
\[\text{let } t_7 = \text{enter}(t_6)(p)(\text{POLY })(t)(\text{poly-type-desc()}) \text{ in}\]
\[\text{let } t_8 = \text{enter}(t_7)(p)(\text{BIT-VECTOR })(t)(\text{bitvector-type-desc()}) \text{ in}\]
\[\text{let } t_9 = \text{enter-characters}(t_8)(p) \text{ in}\]
\[\text{let } t_10 = \text{enter-string}(t_9)(p) \text{ in}\]
\[t_10\]
enter-characters(t)(p)
= let id⁺ = gen-characters(0)(127) in
  let field-values₁ = <ε,*ENUMTYPE*,p,tt,id⁺> in
  let char-type-desc = cons(CHARACTER ,field-values₁) in
  let field-values₂ = <ε,*ENUMELT*,p,tt,mk-type((CONST VAL) )(char-type-desc)> in
  enter-objects(id⁺)(field-values₂)(t)(p)(u)
  where u = λt₁. enter(t₁)(p)(CHARACTER )(field-values₁)

enter-string(t)(p)
= let expr = (NUM 1) in
  let string-type-desc = array-type-desc
  (STRING )(e)(p)(tt)(TO )((second(ΕX ![ expr ]] (p)(t))(e)
  (char-type-desc(t)) in
  enter(t)(p)(STRING )(tl(string-type-desc))

gen-characters(start)(finish)
= (start = finish → ((CHAR ,finish)),
  cons((CHAR ,start),gen-characters(start+1)(finish)))

enter-objects(id* )(field-values)(t)(p)(u)
= (null(id*) → u(t),
  let id = hd(id*) in
  t, enter(t)(p)(id) (<-,*ENTITY* >)
  in
  let t₂ = enter(extend(t₁)(p)(id))(p₁)(*UNIT* )(p₁)(*ENTITY* ) in
  let t₃ = enter(t₂)(p₁)(*LAB* )(p₁)(*ENTITY* ) in
  let t₄ = enter(t₃)(p₁)(*USED* )(p₁)(*ENTITY* ) in
  let t₅ = enter(t₄)(p₁)(*IMPT* )(p₁)(*ENTITY* ) in
  enter(t₅)(p₁)(*ENTITY* )

6.5.2 Entity Declarations

(ENT1) ENT ![ ENTITY id port-decl* decl* opt-id ](p)(u)(t)
  = (¬null(opt-id) ∧ opt-id ≠ id
  → error
    (cat("Entity declaration ");(id)
      (" ended with incorrect identifier ";(opt-id)),
    let t₁ = enter(t)(p)(id)(<ε,*ENTITY* ,p,tt,tt,tt,tt>) in
    let p₁ = %p(id) in
    let t₂ = enter(extend(t₁)(p)(id))(p₁)(*UNIT* )(<ε,*ENTITY* ) in
    let t₃ = enter(t₂)(p₁)(*LAB* )(<ε,tt,tt,tt,tt>) in
    let t₄ = enter(t₃)(p₁)(*USED* )(<ε,tt,tt,tt,tt>) in
    let t₅ = enter(t₄)(p₁)(*IMPT* )(<ε,tt,tt,tt,tt>) in
    PDT ![ port-decl* ](p₁)(tt)(u₁)(t₁)
    where u₁ = λt.DT ![ decl* ](p₁)(tt)(u₁)(t₁)

6.5.3 Architecture Bodies

(ART1) ART ![ ARCHITECTURE id₁ id₂ decl* con-stat* opt-id ](p)(u)(t)
  = (¬null(opt-id) ∧ opt-id ≠ id₁
  → error
    (cat("Architecture body ");(id₁)
      (" ended with incorrect identifier ";(opt-id)),
    let d = lookup(t)(p)(id₂) in
(d = *UNBOUND* v tag(d)≠ *ENTITY*)
→ error(cat("No entity ")(id2)(" for architecture body ")(id1)),
let p1 = %p(id1) in
let t1 = enter(t)(p)(id1)(<e,*ARCHITECTURE* ,p,ff>) in
let t2 = enter(extend(t1)(p)(id1))(p)("UNIT* )(<e,*ARCHITECTURE* >) in
let t3 = enter(t2)(p1)(*LAB* )(<e,e>) in
let t4 = enter(t3)(p1)(*USED* )(<e,e>) in
let t5 = enter(t4)(p1)(*IMPT* )(<e,e,e>) in
D __T
\[ \text{where } \text{ui} = \text{At}.CST \left[ \text{con-stat* } \right] (p1)(u(t))\]

6.5.4 Port Declarations

(PDT0) PDT [ ] (p)(vis)(u)(t) = u(t)

(PDT1) PDT [ port-decl port-decl ] (p)(vis)(u)(t)
= PDT [ port-decl ] (p)(vis)(u)(t)
\[ \text{where } u_1 = \lambda t.PDT [ port-decl ] (p)(vis)(u)(t)\]

The elaboration and checking of a sequence of port declarations proceeds from the first to the last declaration in the sequence.

(PDT2) PDT [ DEC PORT id+ mode type-mark opt-expr ] (p)(vis)(u)(t)
= lookup-type(type-mark)(p)(s)(t)
\[ \text{where } z = \lambda d.let \ \text{type} = (\text{case mode}
\begin{align*}
\text{IN} & \rightarrow \text{mk-type}((\text{SIG VAL}) )(d), \\
\text{OUT} & \rightarrow \text{mk-type}((\text{SIGOUT}) )(d), \\
\text{INOUT, BUFFER} & \rightarrow \text{mk-type}((\text{SIGREF}) )(d), \\
\text{OTHERWISE} & \rightarrow \text{error}
\end{align*}
\left(\text{cat("Illegal mode in port declaration: ") }
\right)(\text{port-decl}))) \text{ in}
\text{process-dec(id+)(type)(opt-expr)(p)(vis)(u)(t)}\]

Refer to the discussion following semantic equation DT5 in Section 6.5.5.

(PDT3) PDT [ SLCDEC PORT id+ mode slice-name opt-expr ] (p)(vis)(u)(t)
= let (type-mark,discrete-range) = slice-name in
lookup-type(type-mark)(p)(s)(t)
\[ \text{where } z = \lambda d.let \ \text{type} = (\text{case mode}
\begin{align*}
\text{IN} & \rightarrow \text{mk-type}((\text{SIG VAL}) )(d), \\
\text{OUT} & \rightarrow \text{mk-type}((\text{SIG OUT}) )(d), \\
\text{INOUT, BUFFER} & \rightarrow \text{mk-type}((\text{SIG REF}) )(d), \\
\text{OTHERWISE} & \rightarrow \text{error}
\end{align*}
\left(\text{cat("Illegal mode in port declaration: ") }
\right)(\text{port-decl}))) \text{ in}
\text{process-slcdec}
\left(\text{id+}(\text{type})(\text{discrete-range})(\text{opt-expr})(p)(\text{vis})(u)(t)\right)\]

Refer to the discussion following semantic equation DT6 in Section 6.5.5.

51
6.5.5 Declarations

(DT0) \( DT \; [ \; \epsilon \; ] \; (p)(vis)(u)(t) = u(t) \)

(DT1) \( DT \; [ \; \text{decl decl*} \; ] \; (p)(vis)(u)(t) \\
= DT \; [ \; \text{decl} \; ] \; (p)(vis)(u_1)(t) \\
\text{where} \; u_1 = \lambda. DT \; [ \; \text{decl*} \; ] \; (p)(vis)(u)(t) \)

(DT2) \( DT \; [ \; \text{pkg-decl pkg-decl*} \; ] \; (p)(vis)(u)(t) \\
= DT \; [ \; \text{pkg-decl} \; ] \; (p)(vis)(u_1)(t) \\
\text{where} \; u_1 = \lambda. DT \; [ \; \text{pkg-decl*} \; ] \; (p)(vis)(u)(t) \)

(DT3) \( DT \; [ \; \text{pkg-body pkg-body*} \; ] \; (p)(vis)(u)(t) \\
= DT \; [ \; \text{pkg-body} \; ] \; (p)(vis)(u_1)(t) \\
\text{where} \; u_1 = \lambda. DT \; [ \; \text{pkg-body*} \; ] \; (p)(vis)(u)(t) \)

(DT4) \( DT \; [ \; \text{use-clause use-clause*} \; ] \; (p)(vis)(u)(t) \\
= DT \; [ \; \text{use-clause} \; ] \; (p)(vis)(u_1)(t) \\
\text{where} \; u_1 = \lambda. DT \; [ \; \text{use-clause*} \; ] \; (p)(vis)(u)(t) \)

The elaboration and checking of a sequence of declarations proceeds from the first to the last declaration in the sequence.

(DT5) \( DT \; [ \; \text{DEC object-class id+ type-mark opt-expr} \; ] \; (p)(vis)(u)(t) \\
= \text{let} \; q = \text{find-progunit-env}(t)(p) \; \text{in} \\
\text{let} \; d = t(q)(*UNIT*) \; \text{in} \\
\text{let} \; tg = \text{tag}(d) \; \text{in} \\
\text{(case object-class} \\
\text{(CONST ,SYSGEN) \rightarrow lookup-type(type-mark)(p)(z)(t),} \\
\text{VAR} \\
\rightarrow \text{(case tg} \\
\text{(*PACKAGE*,*ENTITY*,*ARCHITECTURE*)} \\
\rightarrow \text{error} \\
\text{(cat("Illegal VARIABLE declaration in "))(tg)(" context: ")}(decl)),} \\
\text{OTHERWISE \rightarrow lookup-type(type-mark)(p)(z)(t)),} \\
\text{SIG} \\
\rightarrow \text{(case tg} \\
\text{(*PROCESS*,*PROCEDURE*,*FUNCTION*)} \\
\rightarrow \text{error} \\
\text{(cat("Illegal SIGNAL declaration in "))(tg)(" context: ")}(decl)),} \\
\text{OTHERWISE \rightarrow lookup-type(type-mark)(p)(z)(t)),} \\
\text{OTHERWISE \rightarrow error} \\
\text{(cat("Illegal object class in declaration: "))(decl)))} \\
\text{where} \\
z = \lambda d. \text{let} \; \text{type} = (\text{object-class} = \text{CONST} \rightarrow \text{mk-type}((\text{CONST VAL})(d), \text{mk-type(mk-tmode(object-class)(REF))(d)))) \; \text{in} \\
\text{process-dec(id+)(type)(opt-expr)(p)(vis)(u)(t)} \)

find-progunit-env(t)(p) \\
= (t(p)(*UNIT*) \neq *UNBOUND* \rightarrow p, \\
(null(p) \rightarrow \text{error("No program unit ??! ")}, \\
find-progunit-env(t)(\text{rest}(p))))
lookup-type(id*)(p)(e)(t)
= (null(id*)→ z(void-type-desc()),
  name-type(id*)(e)(p)(t)(v)
  where
  v = \lambda w. \text{second(tmode(w))} = \text{TYP} \rightarrow z(tdesc(w)),
  error(cat("Not a type: ")(namef(tdesc(w))))

name-type(name)(w)(p)(t)(v)
= (null(w)
  → let w₁ = lookup2(t)(p)(e)(hd(name)) in
    (w₁ = *UNBOUND* → error(cat("Unbound identifier: ")(S(p)(hd(name)))),
     let tm = tmode(w₁)
     and d = tdesc(w₁) in
     (second(tm)∈ (OBJ TYP) → name-type(tl(name))(w₁)(p)(t)(v),
      hd(tm)= PATH
     → (~validate-access(name)(w₁)(second(tm))
         → error(cat("Illegal access via: ")(namef(d))),
         name-type(tl(name))((PATH ,tl(second(tm)),d))(p)(t)(v)),
     error
       (cat("Shouldn't happen in auxiliary semantic function NAME-TYPE: ")(w₁))),
     let d = tdesc(w) in
     let tg = tag(d) in
     (null(name)
      → (tg ∈ (*PROCEDURE* *FUNCTION*)
        → (null(body(d))→ error(cat("Missing subprogram body: ")(namef (d))),
         (null(parhs(hd(signatures(d))))→ v(extract-rtype(d)),
          error(cat("Missing subprogram arguments: ")(namef(d)))),
         v(w)),
      let x = hd(name)
      and tm = tmode(w) in
      (cons?(x)
       → list-type(x)(p)(t)(v)
       where
        vv = \lambda w⁺. ((second(tm)= OBJ ∧ is-array?(type(d)))
          \lor (second(tm)∈ (REF VAL) ∧ is-array-tdesc?(d))
         → (length(x)> 1
          → error
           (cat("Too many array indices for: ")(namef (d))),
            (is-integer?(hd(w⁺))
             → name-type
              (tl(name))
              ((second(tm)= OBJ
                → mk-type(tmode(type(d)))(elty(tdesc(type(d)))),
               mk-type(tm)(elty(d)))(p)(t)(v),
              error
               (cat("Non-integer array index for: ")(namef
                (d)))),
            (is-integer?(hd(w⁺))
             → name-type
              (tl(name))
              ((second(tm)= OBJ
                → mk-type(tmode(type(d)))(elty(tdesc(type(d)))),
               mk-type(tm)(elty(d)))(p)(t)(v),
              error
               (cat("Non-integer array index for: ")(namef
                (d)))),
           v(w)),
        let rtype = compatible-signatures(w⁺)(signatures(d)) in
        (null(rtype)
         → error)
(cat("Incompatible parameter types for: ")
(name(d)),
name-type(tl(name))(type(p)(t)(v)),
error(cat(name(d))(" cannot have an argument list")),
((second(tm)= OBJ \& is-record?(type(d)))
\& (second(tm)= (REF VAL) \& is-record-tdesc?(d)))
\rightarrow let d_1 = (second(tm)= OBJ \rightarrow tdesc(type(d)), d) in

let d_2 = lookup-record-field(components(d_1))(x) in
(d_2 = *UNBOUND* \rightarrow error(cat("Unknown record field: ")x),
let tmm = (second(tm)= OBJ \rightarrow tmode(type(d)), tm) in
name-type(tl(name))(mk-type(tmm)(d_2)(p)(t)(v)),
second(tm)= OBJ v second(tm)= TYP
\rightarrow let w_1 = lookup-local(t)(%\(path(d))(idf(d)))(x) in
(w_1 = *UNBOUND* 
\rightarrow error(cat("Unknown identifier: ")(\(path(d))(idf(d)))(x)),
second(tmode(w_1))= ACC \rightarrow name-type(tl(name))(w_1)(p)(t)(v),
hd(tm)= PATH
\rightarrow (\neg null(\(name)) \& \neg validate-access(name)(w_1)(second(tm))
\rightarrow error(cat("Illegal access via: ")(name(tdesc(w_1)))),
name-type
(tl(name))(\(PATH ,tl(second(tm)),tdesc(w_1)))(p)(t)(v),
error
(cat("Shouldn’t happen in auxiliary semantic function NAME-TYPE: "))
(w_1)),
error(cat("Illegal access via: ")(name(d))))

lookup2(t)(p)(q)(id)
= let d = t(p)(id) in
d = *UNBOUND*
\rightarrow (\neg null(p) \rightarrow lookup2(t)(rest(p))(cons(last(p),q))(id), *UNBOUND* ),
(case tag(d)
(*OBJECT* ,*ENUMELT*) \rightarrow ((DUMMY ,OBJ ),d),
(*PACKAGE* ,*PROCESS* ,*PROCEDURE* ,*FUNCTION* ,*LOOPNAME* ,*PROCESSNAME*
\rightarrow ((PATH ,q),d),
OTHERWISE \rightarrow ((DUMMY ,TYP ),d))

validate-access(name)(w)(q)
= let tg = tag(tdesc(w)) in
\neg (tg \in (*PROCESSNAME* *LOOPNAME*)
\rightarrow (tg \in (*PROCEDURE* *FUNCTION*)
\& (\neg null(tl(name)) \& \neg consp(hd(tl(name)))))
\rightarrow (\neg null(q) \& hd(name)= hd(q))

list-type(expr*)(p)(t)(vv)
= (null(expr*) \rightarrow vv(e),
let expr = hd(expr*) in
ET [ expr ] (p)(k)(t)
where
k = \lambda(w,e),t.
(second(tmode(w))= ACC
\rightarrow error(cat("Non-value (an access): ")(name(tdesc(w))))).
list-type(tl(expr*))(p)(t)(\lambda w\.vv(cons(w,w*))))

lookup-local(t)(p)(id)
= let d = t(p)(id) in
(d = *UNBOUND* \rightarrow *UNBOUND*,

54
let tg = tag(d) in
(tg ∈ (*LOOPNAME* *PROCESSNAME*) → ((DUMMY,ACC),d),
(exported(d)
→ (case tg
  (*OBJECT* *ENUMELT*) → ((DUMMY,OBJ),d),
  (*PACKAGE* *PROCESS* *PROCEDURE* *FUNCTION*) → ((DUMMY,ACC),d),
  OTHERWISE → ((DUMMY,TYP),d)),
*UNBOUND* )))

compatible-signatures(x)(signatures)
= (null(signatures) → ε,
  let signature = hd(signatures) in
  (compatible-par-types(x)(extract-par-types(pars(signature)))
  → rtype(signature),
  compatible-signatures(x)(tl(signatures))))

compatible-par-types(x)(y)
= (length(x)≠ length(y) → ⊥,
  length(x)= 0 → tt,
  let w₁ = hd(x)
  and w₂ = hd(y) in
  (tdesc(w₁)≠ tdesc(w₂) → ⊥,
  let m₁ = ref-mode(tmode(w₁))
  and m₂ = ref-mode(tmode(w₂)) in
  (m₁ = REF ∨ m₁ = m₂ → compatible-par-types(tl(x))(tl(y)), ⊥))

extract-par-types(x)
= (null(x) → ε, cons(second(hd(x)),extract-par-types(tl(x))))

extract-rtype(d)
= let signature = hd(signatures(d)) in
  rtype(signature)

lookup-record-field(comp*)(id)
= (null(comp*) → *UNBOUND*,
  let (x,d) = hd(comp*) in
  (x = id → d, lookup-record-field(tl(comp*))(id)))

process-dec(id⁺)(w)(opt-exp)(p)(vis)(u)(t)
= (null(opt-exp)
  → (is-const?(w) → error(cat("Uninitialized constant: ")(t)(hd(id⁺))),
    enter-objects(id⁺)(<c,OBJECT*,p,vis,w,*UNDEF*,e>)(t)(p)(u)),
  let expr = opt-exp in
  RT [ expr ] (p)(k)(t)
where
k = λ(w₂,e).t,
let d = tdesc(w)
  and d₁ = tdesc(w₁) in
  (match-types(d,d₁)
   → let init-val = (is-const?(w)∧ (is-array?(w))∧ is-record?(w)) → c,
     *UNDEF* ) in
  enter-objects(id⁺)(<c,OBJECT*,p,vis,w,init-val,e>)(t)(p)(u),
  error(cat("Initialization type mismatch: ")(d)(d₁))))

55
match-types\((d_1, d_2)\)
\[= (\text{case tag}(d_1)\]
\[\quad (*\text{BOOL}* , *\text{BIT}* , *\text{INT}* , *\text{REAL}* , *\text{TIME}* , *\text{ENUMTYPE}* ) \rightarrow d_1 = d_2,\]
\[\quad *\text{ARRAYTYPE}* \]
\[\quad \rightarrow \text{tag}(d_2)= *\text{ARRAYTYPE}* \land \text{match-array-type-names}(d_1,d_2),\]
\[\quad *\text{RECORDTYPE}* \]
\[\quad \rightarrow \text{tag}(d_2)= *\text{RECORDTYPE}* \]
\[\& \text{null}(\text{set-difference}\]
\[\quad (\text{filter-components}(\text{type}(d_1)))\text{)(filter-components}(\text{type}(d_2))))\]
\[\);\]
\[\text{OTHERWISE } \rightarrow \text{match-type-names}((d_1), (d_2)))\]

match-array-type-names\((d_1,d_2)\)
\[= \text{let} \ idf_1 = \text{hd}(d_1)\]
\[\quad \text{and} \ idf_2 = \text{hd}(d_2) \text{ in}\]
\[\quad (\text{consp}(idf_1) \land \text{consp}(idf_2) \rightarrow \text{match-type-names}((\text{hd}(idf_1),\text{hd}(idf_2))),\]
\[\quad \text{consp}(idf_1) \rightarrow \text{match-type-names}((\text{hd}(idf_1),idf_2),\]
\[\quad \text{consp}(idf_2) \rightarrow \text{match-type-names}(idf_1,\text{hd}(idf_2)),\]
\[\quad \text{match-type-names}(idf_1,idf_2))\]

match-type-names\((id_1,id_2)\)
\[= id_1 = *\text{ANONYMOUS}* \lor id_2 = *\text{ANONYMOUS}*\]

array-size\((d)\)
\[= (\text{ub}(d) \land \text{lb}(d)\]
\[\quad \rightarrow \text{let} \ \text{lbound} = \text{hd}(\text{tl}(\text{lb}(d)))\]
\[\quad \text{and} \ \text{ubound} = \text{hd}(\text{tl}(\text{ub}(d))) \text{ in}\]
\[\quad (\text{ubound}-\text{lbound})+1,\]
\[\quad -1)\]

filter-components\((\text{components})\)
\[= (\text{null}(\text{components}) \rightarrow \epsilon,\]
\[\quad \text{let} \ \text{component} = \text{hd}(\text{components}) \text{ in}\]
\[\quad \text{cons}((\text{hd}(\text{component}),\text{second}(\text{component})),\]
\[\quad \text{filter-components}(\text{tl}(\text{components}))))\]

An object declaration declares a list of identifiers to be of the type given by the type-mark, which must be the name of a type that has already been entered in the visible part of the TSE. The identifiers must be distinct. The first of these identifiers is used in error messages. If the identifiers are being declared as constants but no initialization expression is present, then an UNINITIALIZED-CONSTANT error is reported. If constants are being declared, then their type is a value type; variables and signals have reference types. If variables or signals are being declared without an initialization expression, then the identifiers are entered into the TSE with an undefined initial value *UNDEF* by the function enter-objects, whose operation is explained below. If present, the initialization expression is checked and its type compared to the value type of the declared identifiers. If these types are not equal, then an initialization type mismatch is reported. If the identifiers are being declared as constants, they are entered into the TSE with an initial value equal to the (static) value of the initialization expression.

The function enter-objects enters into the TSE a scalar descriptor for each of a list of identifiers. Duplicate declarations are detected. The descriptors are created from (1) the identifiers and (2) a list of remaining field values input to enter-objects.
The function **name-type** returns the type (consisting of a type *mode* and a type *descriptor*) of a *reference* (*ref*). In Phase 1, *refs* are essentially sequences of identifiers and expression lists; *refs* must begin with an identifier. As **name-type** processes a *ref*, it carries along (in parameters *name* and *w*, respectively) the remainder of the *ref* to be processed and the type to be computed for that portion of the original *ref* processed thus far. During this processing, special type modes that are identifier lists may be used to validate accesses to items declared inside packages or subprograms; **validate-access** checks these accesses. The function **list-type** returns the list of the types of its components; when a list is used as an actual parameter list in a subprogram call, **compatible-par-types** checks whether the types of this list's components are compatible with (not necessarily equal to) the types of the corresponding formal parameters of the subprogram.

(DT6) \[
\text{DT} \begin{ Monospace }
\text{SLCDEC} \text{ object-class } \text{id}\+ \text{ slice-name opt-expr } \text{ (p)(vis)(u)(t)}
\end{ Monospace }
\]

\[
= \text{let } \text{(type-mark,discrete-range)} = \text{ slice-name in}
\]

\[
\text{let } t = \text{find-progunit-env}(t)(p) \text{ in}
\]

\[
\text{let } d = t(q)(\text{UNIT}*) \text{ in}
\]

\[
\text{let } \text{tg} = \text{tag}(d) \text{ in}
\]

\[
\text{(case object-class}
\]

\[
\begin{Monospace}
\text{(CONST ,SYSGEN)} \rightarrow \text{lookup-type(type-mark)(p)(z)(t)},
\text{VAR}
\end{Monospace}
\]

\[
\rightarrow \begin{ Monospace }
\text{(case } \text{tg}
\end{ Monospace }
\]

\[
\begin{Monospace}
\text{(*PACKAGE* ,*ENTITY* ,*ARCHITECTURE*)}
\end{ Monospace}
\]

\[
\rightarrow \text{error}
\]

\[
\text{(cat("Illegal VARIABLE declaration in ")}(tg)
\text{" context: ")}(\text{decl}))
\]

\[
\text{OTHERWISE} \rightarrow \text{lookup-type(type-mark)(p)(z)(t)},
\text{SIG}
\]

\[
\rightarrow \begin{ Monospace }
\text{(case } \text{tg}
\end{ Monospace }
\]

\[
\begin{Monospace}
\text{(*PROCESS* ,*PROCEDURE* ,*FUNCTION*)}
\end{ Monospace}
\]

\[
\rightarrow \text{error}
\]

\[
\text{(cat("Illegal SIGNAL declaration in ")}(tg)" context: ")}(\text{decl}))
\]

\[
\text{OTHERWISE} \rightarrow \text{lookup-type(type-mark)(p)(z)(t)},
\]

\[
\text{OTHERWISE}
\]

\[
\rightarrow \text{error(cat("Illegal object class in declaration: ")}(\text{decl}))
\]

\[
\text{where}
\]

\[
\text{z} = \lambda d.\text{let } \text{type} = \text{(object-class } = \text{CONST } \rightarrow \text{mk-type)((CONST VAL) })(d),
\text{mk-type(mk-tmode(object-class)(REF ))}(d) \text{ in}
\]

\[
\text{process-slcdec(id\+)(type)(discrete-range)(opt-expr)(p)(vis)(u)(t)}
\]

\[
\text{process-slcdec(id\+)(w)(discrete-range)(opt-expr)(p)(vis)(u)(t)}
\]

\[
= \text{let } d = \text{tdesc}(w) \text{ in}
\]

\[
\text{(is-array?}(w)\rightarrow \text{error(cat("Can't form slice of non-array type: ")}(d)),
\]

\[
\text{let } \text{(direction,expr\_1,expr\_2)} = \text{discrete-range in}
\]

\[
\text{RT } \begin{ Monospace }
\text{[ exp1 ] (p)(k\_1)(t)}
\end{ Monospace }
\]

\[
\text{where}
\]

\[
\text{k\_1} = \lambda (w\_1,e\_1).t.
\]

\[
\text{RT } \begin{ Monospace }
\text{[ exp2 ] (p)(k\_2)(t)}
\end{ Monospace }
\]

\[
\text{where}
\]

\[
\text{k\_2} = \lambda (w\_2,e\_2).t.
\]

\[
\text{(is-integer?}(w\_1)\land \text{is-integer?}(w\_2))
\]

\[
\rightarrow \text{error}
\]

\[
\text{(cat("Non-integer array bound for: ")}(\text{hd(id\+)})),
\]

57
let field-values = tl(array-type-desc
  (TEMP_NAME )((p)(vis)
  (direction)
  ((direction = TO
    (e1 = *UNDEF*
     → second(EXPR [ expr1 ] (p)(t)),
     (NUM ,e1)),
    (e2 = *UNDEF*
     → second(EXPR [ expr2 ] (p)(t)),
     (NUM ,e2))))
  ((direction = TO
    (e2 = *UNDEF*
     → second(EXPR [ expr2 ] (p)(t)),
     (NUM ,e2)),
    (e1 = *UNDEF*
     → second(EXPR [ expr1 ] (p)(t)),
     (NUM ,e1))))(elty(d))) in

(null(opt-expr)
  → enter-array-objects
    (id\+)(idf(d))(tmode(w))(field-values)(t)(p)(vis)
    (u),
  check-array-aggregate(opt-expr)(p)(v)(t)
  where
    v = \lambda w.\text{(match-types}(elty(d),tdesc(w3)))
    → enter-array-objects
      (id\+)(idf(d))(tmode(w))
      (field-values)(t)(p)(vis)(u),
    error
      (cat("Initialization type mismatch for: ")
       $(p)(hd(id\+))))))))

enter-array-objects(id\+)(array-type-name)(tmode)(field-values)(t)(p)(vis)(u)
= (null(id\+)) → u(t),
  let id2 = hd(id\+)
  in
  let id3 = new-array-type-name(array-type-name) in
  let d1 = cons(id2,field-values) in
  let t1 = enter(t)(p)(id1)(field-values) in
  let new-type = mk-type(tmode)(d1) in
  (t(p)(id1)\ne *UNBOUND*
   → error(cat("Duplicate array declaration: ")$(p)(id1)))),
  let d2 = <e,*OBJECT* ,p,vis,new-type,*UNDEF* ,e> in
  let t2 = enter(t)(p)(id1)(d2) in
  enter-array-objects
    (tl(id\+))(array-type-name)(tmode)(field-values)(t2)(p)(vis)(u)))

check-array-aggregate(expr)(p)(v)(t)
= let (tg,expr\+)) = expr in
  (tg \ne BITSTR ^\land tg \ne STR
   → error(cat("Improper array initialization aggregate: ")(expr)),
   let expr1 = hd(expr\+) in
   RT [ expr1 ] (p)(k)(t)
   where k = \lambda (w1,e1),t.check-exprs(w1)(tl(expr\+))(p)(v)(t))

check-exprs(w)(expr\+)(p)(v)(t)
= (null(expr\+)) → v(w),
  let expr = hd(expr\+) in
  RT [ expr ] (p)(k)(t)
where
\[ k = \lambda (w_1, e_1). t. \]
\[ (w_1 \neq w \rightarrow \text{"Nonuniform array aggregate ",} \]
\[ \text{check-exprs}(w) (t l (\text{expr}^*)) (p) (v) (t)) \]

A declaration of a slice of a (previously defined) array type is a special form of object declaration for arrays of \textit{anonymous} type. Because a declaration of a list of identifiers is considered to be an abbreviated representation of the sequence of corresponding declarations of each of the individual identifiers in the list, the (anonymous) type of each of the declared identifiers is \textit{distinct}. Each of these distinct anonymous array types is given a distinct, new, system-generated name in Phase 1 of the Stage 2 VHDL translator (via the function \texttt{new-array-type-name}), and corresponding distinct type descriptors are entered into the TSE. If present, the initialization part of the declaration is a \textit{list} of scalar expressions.

The elaboration and checking of a slice declaration begins in the same way as for a scalar declaration. The slice bound expressions are then evaluated and checked to ensure that both are integers. If the initialization part is absent, then descriptors for the declared array identifiers, together with the descriptors for the corresponding anonymous array types, are entered into the environment by \texttt{enter-array-objects}.

If the initialization part is present, then it is first processed by \texttt{check-array-aggregate}, which invokes \texttt{check-exprs} to ensure that each element of the initialization part has the same (value) type; \texttt{check-aggregate} returns this type, which is then compared to the array’s declared value type. Finally, \texttt{enter-array-objects} is invoked to enter the descriptors for the declared arrays into the environment.

Refer also semantic equation \textbf{DT8}, shown below.

\begin{equation}
(DT7) \quad DT [ ETDEC id id^+ ] (p)(vis)(u)(t)
= \text{let field-values}_1 = \langle e, *ENUMTYPE*, p, vis, id^+ \rangle \text{ in }
  (\text{check-enum-lits}(t)(p)(id)(id^+)
   \rightarrow \text{enter-objects}((id))(field-values_2)(t)(p)(u_1),
   \text{nil})
\end{equation}

\textbf{where}

\[ u_1 = \lambda_1. \text{let } d = \text{cons}(id, field-values_1) \text{ in }
  \text{let } field-values_2 = \langle e, *ENUMELT*, p, vis, mk-type((\text{CONST VAL}) ) (d) \rangle \text{ in }
  \text{enter-objects}(id^+)(field-values_2)(t_1)(p)(u)
\]

\begin{equation}
\text{check-enum-lits}(t)(p)(id)(id^*)
= (\text{null}(id^*) \rightarrow \text{tt},
  \text{let } id_1 = \text{hd}(id^*) \text{ in }
  (\text{lookup}(t)(p)(id_2) = *UNBOUND* \rightarrow \text{check-enum-lits}(t)(p)(id)(t\langle id^* \rangle)),
  \text{error}
  (\text{cat("Illegal overloading for enumeration literal ")}(id_1)
   (* in enumeration type: *) (\$\langle p)(id)\rangle))
\end{equation}

An enumeration type declaration causes corresponding enumeration type descriptors to be entered into the TSE. At the same time, descriptors for the individual elements of the enumeration type are entered into the TSE; these elements are treated as constants.

\begin{equation}
(DT8) \quad DT [ ATDEC id discrete-range type-mark ] (p)(vis)(u)(t)
\end{equation}
= lookup-type(type-mark)(p)(z)(t)
  where
  z = λd.let (direction,expr1,expr2) = discrete-range in
  let array-type-desc = array-type-desc
  (id)(e)(p)(vis)(direction)
  ((direction = TO
  → second(EX [ expr1 ] (p)(t)),
  second(EX [ expr2 ] (p)(t))))
  ((direction = TO
  → second(EX [ expr2 ] (p)(t)),
  second(EX [ expr1 ] (p)(t))))(d) in
  attributes-low-high
  ((id,array-type-desc,(INTEGER) ))(p)(vis)(u)(t)
attributes-low-high(id,type-desc,attribute-type-mark)(p)(vis)(u)(t)
= enter-objects((id))(tl(type-desc))(t)(p)(u)
  where
  u₁ = λt₁.let decl = (DEC,SYSGEN,(mk-tick-low(id),mk-tick-high(id)),attribute-type-mark,ε) in
  DT [ decl ] (p)(vis)(u)(t₁)
  mk-tick-low(id) = catenate(id, "'LOW")
  mk-tick-high(id) = catenate(id, "'HIGH")

An array type declaration causes corresponding array type descriptors to be entered into the TSE. The array type attributes 'low and 'high, representing the lower and upper bounds, respectively, are declared as system-generated variables.

(DT9) DT [ PACKAGE id decl* opt-id ] (p)(vis)(u)(t)
  = let d = (t(p)(id))≠ *UNBOUND*
    -> error(cat("Duplicate package declaration: ")($p)(id))),
    (¬null(opt-id)∧ opt-id ≠ id
    -> error
      (cat("Package ")(p)(id))(" ended with incorrect identifier: ")
      (opt-id)),
    let d₁ = enter(t)(p)(id)(d) in
    let t₁ = enter(t₁)(p)(id)(d₁) in
    let t₂ = enter(t₂)(%p(id))(UNIT*) (<,*,PACKAGE*> ) in
    let t₃ = enter(t₃)(%p(id))(USED*) (<,*,PACKAGE*> ) in
    let t₄ = enter(t₄)(%p(id))(IMPT*) (<,*,PACKAGE*> ) in
    u₁(t₄)
    where u₁ = λt.DT [ decl* ] (p)(id)(tt)(u)(t))
(DT10) DT [ PACKAGEBODY id decl* opt-id ] (p)(vis)(u)(t)
  = let d = (t(p)(id)) in
    (d = *UNBOUND* -> error(cat("Missing package declaration: ")($p)(id))),
    tag(d)≠ *PACKAGE* → error(cat("Not a package declaration: ")($p)(id))),
    ¬null(pbody(d))→ error(cat("Duplicate package body: ")($p)(id))),
    ¬null(opt-id)∧ opt-id ≠ id
    -> error
      (cat("Package body ")($p)(id))(" ended with incorrect identifier: ")
      (opt-id)),
    let q = %p(path(d))(id) in
    let t₁ = enter(t₁)(q)(LAB*) (<,*,PACKAGE*> ,path(d),exported(d),*BODY* ) in
    DT [ decl* ] (q)(fl)(u)(t₂)
A package is an encapsulated collection of declarations (including other packages) of logically related entities identified by the package's name. A package is generally provided in two parts: the *package declaration* and the *package body*. The package declaration provides declarations of those items that are *exported* (i.e., made visible) by the package. The package body provides the bodies of items whose declarations appear in the package declaration, together with the declarations and bodies of additional items that support the items exported by the package. These latter items are not exported by the package, i.e., they cannot be made visible outside the package. In our implementation, the descriptors of exported and nonexported items alike are entered into the same local environment. The *exported* field of these descriptors distinguishes between the two kinds of items. If an item can be exported by a USE clause, then the *exported* field of its descriptor contains *tt* (denoting true; if not, then this field contains *ff* (false).

The items declared in a package declaration are not directly visible outside the package, but they can be accessed by using a dotted name beginning with the package name, provided that the package name is visible at the point of access. A descriptor for the package declaration is entered into the current environment. In order to encapsulate the items within a package, the resulting TSE is then extended along the current path by an edge labeled with the package name; the new environment is marked (in its *UNIT* cell) as a package environment. Then the constituent declarations of the package are elaborated and checked in the new environment.

The items declared in a package body are not exported from the package and thus must not be accessible by an extended name. Therefore the *exported* field of the descriptors for the inaccessible entities must be set to *ff*, thus marking them as *not exportable*.

(DT11) \[
\text{PROCEDURE} \text{id proc-par-spec* type-mark } (p)(vis)(u)(t) = \begin{cases} 
& (t(p)(id)\neq *UNBOUND*) \\
& \quad \text{error(cat("Duplicate procedure declaration for: ")$(p)(id)))}, \\
& \quad \text{let } p_1 = \%%(p)(id) \text{ in} \\
& \quad \text{let } t_1 = \text{enter(extend}(t)(p)(id))(p_1)(*UNIT*)(<,\text{*PROCEDURE* }>) \text{ in} \\
& \quad \text{enter-formal-pars(*PROCEDURE*)}(proc-par-spec*)(t_1)(p_1)(u_1) \text{ where} \\
& \quad \quad u_1 = \lambda t_2.\text{let} \text{formals } = \text{let id}^+ = \text{collect-fids(proc-par-spec*) in} \\
& \quad \quad \text{collect-formal-pars(id}^+)(t_2)(p_1) \text{ in} \\
& \quad \quad \text{let } d = <,\text{*PROCEDURE* },p,\text{vis}, \\
& \quad \quad \quad ((\text{formals,mk-type}((\text{CONST} \text{ VAL} ))(\text{void-type-desc}()))),e,e> \text{ in} \\
& \quad \quad \text{u(enter}(t_2)(p)(id)(d)) \\
\end{cases}
\]

(DT12) \[
\text{FUNCTION} \text{id func-par-spec* type-mark } (p)(vis)(u)(t) = \begin{cases} 
& (t(p)(id)\neq *UNBOUND*) \\
& \quad \text{error(cat("Duplicate function declaration for: ")$(p)(id)))}, \\
& \quad \text{let } p_1 = \%%(p)(id) \text{ in} \\
& \quad \text{lookup-type(type-mark)(p)(z)(t) where} \\
& \quad \quad z = \lambda d_1.\text{let } t_1 = \text{enter} \\
& \quad \quad \text{(extend}(t)(p)(id))(p_1)(*UNIT*)(<,\text{*FUNCTION* }>) \text{ in} \\
& \quad \quad \text{enter-formal-pars(*FUNCTION*)}(func-par-spec*)(t_1)(p_1)(u_1) \text{ where} \\
& \quad \quad \quad u_1 = \lambda t_2.\text{let} \text{formals } = \text{let id}^+ = \text{collect-fids} \\
& \quad \quad \quad \quad (func-par-spec*) \text{ in} \\
& \quad \quad \quad \text{collect-formal-pars} \\
\end{cases}
\]
(id+)(t2)(p1) in
let d = <e,*FUNCTION*,p,vis,
((formals,mk-type((VAR VAL))(d1)),e,e) in
u(enter(t2)(p)(id)(d)))

enter-formal-pars(tg)(par-spec*)(t)(p)(u)
= (null(par-spec*)→ u(t),
let par-spec = hd(par-spec*) in
let (object-class,id+,mode,type-mark,opt-expr) = par-spec in
  (case tg
    "*PROCEDURE*"
    → (case object-class
      (CONST ,VAR )
      → (case mode
        (IN ,OUT ,INOUT ) → lookup-type(type-mark)(p)(z)(t),
        OTHERWISE → error
        (cat("Illegal mode for procedure parameters: ")(d)(p)(hd(id+)))))
    OTHERWISE → error
    (cat("Unimplemented object class ")(object-class)
     (" for procedure parameters: ")(d)(p)(hd(id+))),
    "*FUNCTION*"
    → (case object-class
      CONST
      → (case mode
        IN → lookup-type(type-mark)(p)(z)(t),
        OTHERWISE → error
        (cat("Illegal mode for function parameters: ")(d)(p)(hd(id+))),
      OTHERWISE → error(cat("Illegal subprogram tag: ")(t))
    where
  z = λd.let type = (case mode
    IN → mk-type(mk-tmode(object-class)(VAL))(d),
    OUT → mk-type(mk-tmode(object-class)(OUT))(d),
    OTHERWISE → mk-type(mk-tmode(object-class)(REF))(d)) in
let fv = <e,*OBJECT*,p,tt,type,*UNDEF*,e> in
enter-objects(id+)(fv)(t)(p)(u1)
  where u1 = λt.enter-formal-pars(tg)(tl(par-spec*)(t))(p)(u)

collect-fids(par-spec*)
= (null(par-spec*)→ e,
let par-spec = hd(par-spec*) in
let (object-class,id+,mode,type-mark,opt-expr) = par-spec in
append(id+,collect-fids(tl(par-spec*)))

collect-formal-pars(id*)(t)(p)
= (null(id*)→ e,
let d = t(p)(hd(id*)) in
cons((hd(id*),type(d)),collect-formal-pars(tl(id*))(t)(p)))
Checking a subprogram (procedure or function) declaration first extends the TSE and identifies the new environment at the end of the extended path (in its *UNIT* cell) as a procedure or function environment. Then descriptors for the subprogram's formal parameters are entered (by enter-formal-pars) into this new environment. Finally, a descriptor for the subprogram (with a body field of *if*, indicating that no body for this subprogram has been encountered) is entered into the environment in which the subprogram is declared locally. Procedures are always given a *void* return type. The function enter-formal-pars accepts a tag *PROCEDURE* or *FUNCTION* (procedure or function) to enable it to check that the formal parameters are appropriate to the subprogram. For example, functions can have only IN parameters.

(DT13) \[
\text{DT} \text{[SUBPROGBODY subprog-spec decl* seq-stat* opt-id ] (p)(vis)(u)(t)}
\]
\[
= \text{let } (tg, id, par-spec*, type-mark) = \text{subprog-spec in}
\]
\[
\text{let } qname = \$\langle p\rangle(id)
\]
\[
\text{and } d = \langle t\rangle(p)(id) \text{ in}
\]
\[
\text{d = *UNBOUND*}
\]
\[
\rightarrow \text{let } \text{decl = subprog-spec in}
\]
\[
\text{DT} \text{[decl ] (p)(vis)(u1)(t)}
\]
\[
\text{where}
\]
\[
u1 = \lambda t.\text{let } d = \langle t\rangle(p)(id) \text{ in}
\]
\[
\text{process-subprog-body}
\]
\[
\langle t\rangle(p)(id)(d)(\text{decl})(\text{seq-stat})(u),
\]
\[
\forall (\text{tag}(d)\in (*\text{PROCEDURE*} *\text{FUNCTION*}))
\]
\[
\rightarrow \text{error}(\text{cat}(\text{qname})(\text{" is not a subprogram specification\}})) ,
\]
\[
(tg = \text{PROCEDURE} \wedge \text{tag}(d) = *\text{FUNCTION*})
\]
\[
\text{\text{\(\vdash\)}}(tg = \text{FUNCTION} \wedge \text{tag}(d) = *\text{PROCEDURE*})
\]
\[
\rightarrow \text{error}(\text{cat}(\text{Wrong kind of subprogram body: "})(\text{qname})) ,
\]
\[
\text{null(body}(d))\rightarrow \text{error}(\text{cat}(\text{Duplicate subprogram body: "})(\text{qname})) ,
\]
\[
\text{null(opt-id)} \neq \text{id}
\]
\[
\rightarrow \text{error}
\]
\[
\text{cat}(\text{"Subprogram body "})(\text{qname})
\]
\[
\text{(" ended with incorrect identifier "))(\text{opt-id})) ,
\]
\[
\text{let } \text{formals = let } id^+ = \text{collect-fids(par-spec* ) in}
\]
\[
\text{collect-formal-pars(id^+)(t)(\%\langle p\rangle(id)) in}
\]
\[
\text{formals \neq \text{pars(hd(signatures}(d)))}
\]
\[
\rightarrow \text{error}
\]
\[
\text{cat}(\text{"Nonconforming formal parameters for subprogram: "})(\text{qname}),
\]
\[
\text{lookup-type(type-mark)(p)(z)(t) in}
\]
\[
\text{where}
\]
\[
z = \lambda d.1.(d_1 \neq \text{tdesc(extract-rtype}(d))
\]
\[
\rightarrow \text{error}
\]
\[
\text{cat}(\text{"Unequal result types for subprogram: ")}
\]
\[
\text{\text{\(\vdash\)qname}) ,
\]
\[
\text{process-subprog-body}(t\langle p\rangle(id)(d)(\text{decl})(\text{seq-stat})(u))
\]
\[
= \text{let } p_2 = \%\langle p\rangle(id) \text{ in}
\]
\[
\text{let } t_1 = \text{enter}(t\langle p\rangle)(\text{*LAB* })\langle (\varepsilon,\varepsilon) \rangle \text{ in}
\]
\[
\text{let } t_5 = \text{enter}(t\langle 1\rangle)(\text{*USED* })\langle (\varepsilon,\varepsilon) \rangle \text{ in}
\]
\[
\text{let } t_6 = \text{enter}(t\langle s\rangle)(\text{*IMPT* })\langle (\varepsilon,\varepsilon,\varepsilon) \rangle \text{ in}
\]
\[
\text{let } t_7 = \text{enter}
\]
\[
\langle t\rangle(p)(id)
\]
\[
\langle (\varepsilon,\text{tag}(d),\text{path}(d),\text{exported}(d),\text{signatures}(d),\text{\text{\(\vdash\)BODY}} \rangle \text{ in}
\]
\[
\text{DT} \text{[decl* ] (p)(tt)(u1)(tt)}
\]
\[
\text{where } u1 = \lambda t2.\text{SST} \text{[ seq-stat* ] (p1)(u2)(t2)}
\]
63
Checking the declaration of a subprogram body first checks whether a declaration for the subprogram has already been encountered. If not, then descriptors for the subprogram and its formal parameters must be entered into the TSE as above. Otherwise, the declaration part of the subprogram body must be checked for conformity with the corresponding information previously entered in the TSE. In Stage 2 VHDL conformity is very strict: subprogram types and formal parameter names and types must agree exactly, except that formal parameters with no explicit mode are regarded as having been specified with mode IN. The subprogram’s body (which consists of local declarations followed by statements) is checked by process-subprog-body, where initial entries are made into its environment’s *LAB*, *USED*, and *IMPT* cells, and its transformed abstract syntax tree is entered into the body field of the subprogram’s descriptor. Note that a dummy value *BODY* is temporarily entered in the descriptor’s body field, so that recursive calls of this subprogram will not incorrectly indicate that a call is being made to a subprogram for which a body has not been supplied (see the Phase 1 semantics of subprogram calls).

\[ \text{where} \]
\[ u_2 = \lambda t_3. \text{let } t_4 = \text{enter} \]
\[ ((t_3)(p)(id)) \]
\[ ((<r, \text{tag}(d), \text{path}(d), \text{exported}(d), \text{signatures}(d)), \]
\[ (\text{DX} \ll \text{decl}^* \gg (p_1)(t_3), \text{SSX} \ll \text{seq-stat}^* \gg (p_1)(t_3))) \in \]
\[ u(t_4) \]
check-pkg-names
  (tl(dotted-name*))((cons(%(path(d)))(id)(d)),pkg-qualified-names)
  (p)(vis)(j)(i)(t))

remove-enclosing-pkgs(p)(t)(pkg-set)
  = (null(p)→ pkg-set,
     let d = t(p)(*UNIT* ) in
     (d = *UNBOUND* → remove-enclosing-pkgs(rest(p))(t)(pkg-set),
      (third(d)= *PACKAGE* → remove-enclosing-pkgs(rest(p))(t)(set-difference(pkg-set)((p)) ),
       remove-enclosing-pkgs(rest(p))(t)(pkg-set))))

import-qualified-names(pkg-qualified-names)(item-qualified-names)(ids-used)(p)(t)
  = (pkg-qualified-names = e → enter(t)(p)(*IMPT* )((e,item-qualified-names,ids-used)),
     let pkg-qn = hd(pkg-qualified-names) in
     let pkg-env = t(pkg-qn) in
     let exported-qnames = export-qualified-names(pkg-env)(e) in
     let local-env = t(p) in
     let (qname*,id*) = import-legal
       (exported-qnames)(item-qualified-names)(ids-used)
       (local-env) in
     import-qualified-names(tl(pkg-qualified-names))(qname*)(id*)(p)(t)

import-legal(exported-qnames)(qname-list)(id-list)(env)
  = (null(exported-qnames)→ (qname-list,id-list),
     let qname = hd(exported-qnames) in
     let id = last(qname) in
     let remaining-exported-qnames = tl(exported-qnames) in
     (id ∈ id-list → let qn = simple-name-match(id)(qname-list) in
       (null(qn)
        → import-legal(remaining-exported-qnames)(qname-list)(id-list)(env),
        import-legal
        (remaining-exported-qnames)(set-difference(qname-list)((qn)))
        (id-list)(env)),
       let d = env(id) in
       (d = *UNBOUND* → import-legal
        (remaining-exported-qnames)(cons(qname,qname-list))
        (cons(id,id-list))(env),
        import-legal
        (remaining-exported-qnames)(qname-list)(cons(id,id-list))(env)))))

simple-name-match(id)(qname*)
  = (null(qname*)→ ε,
      (id = last(hd(qname*))→ hd(qname*), simple-name-match(id)(tl(qname*)))))

export-qualified-names(env)(qualified-names)
  = (null(env)→ qualified-names,
     let d = hd(env) in
     let id = idf(d) in
     (case id
       (**UNIT* , *LAB* , *USED* , *IMPT* )
       → export-qualified-names(θl(env))(qualified-names),
       OTHERWISE
       → (exported(d)
          → export-qualified-names(tl(env))(cons(%(path(d)))(id),qualified-names),
          export-qualified-names(tl(env))(qualified-names))))
A USE clause is a declaration that makes items declared in a package specification visible at the location of the USE clause. Each of the dotted names in a USE clause, neglecting the (obligatory) suffix ALL, must denote the name of a package. In essence, a USE clause combines the exported environments associated with its named packages both with each other and with the local environment (among whose declarations the USE clause appears). Such a combination of environments may introduce conflicts, since there may be several different declarations of an object of the same name in the packages (as well as one locally). Therefore, certain constraints must govern how environments are combined:

1. If an object x is declared locally, then no declarations of x may be imported to the local environment by the USE clause.

2. If an object x is declared in more than one of the packages named in the USE clause, then none of these declarations of x may be imported to the local environment by the USE clause, even if x is not declared locally.

These constraints ensure that (1) no local declaration is masked by an imported one, and (2) no duplicate or conflicting declarations are imported.

USE clauses are treated by process-use-clause, which assumes that all the USE clauses in a program unit's declarative part are located together at the end of that declarative part. This restriction on the location and grouping of USE clauses enables a determination of those items imported into a local environment to be made once and for all by the time the unit's declarative part has been processed. This ensures that the list of items imported into an environment (stored in its *IMPT* cell) need not vary in Phase 2, thereby ensuring that the entire TSE is fixed throughout Phase 2. If declarations other than USE clauses were allowed to appear between USE clauses, then the set of importable items may change before and after such interposed declarations, requiring a dynamic evaluation of the import list during Phase 2. We feel that such generality is unnecessary, because the names of items can always be changed so that their interposed declarations can be moved in front of the group of USE clauses.

First, the list of names appearing in this USE clause (with duplicates removed) is given to process-use-clause. Then these names are checked by check-pkg-names to ensure that they denote packages; a list of fully qualified package names is returned. The names of packages that enclose packages in this list are removed by remove-enclosing-packages. The (set-theoretic) union of the resulting set of package names (called pkg-qnames) and the set of names of packages already appearing in USE clauses in this declarative part (stored in the *USED* cell of this environment) is computed (in order to avoid duplication); the resulting set of package names is entered back into the *USED* cell. Next, the current set of fully qualified names of items imported into this environment (qname-list) is retrieved from its *IMPT* cell. A separate list of simple identifiers (id-list) is also maintained in the *IMPT* cell; this list is used to prevent illegal importations into the current environment. Then pkg-qnames, qname-list, and id-list are passed to import-qualified-names, which adds the fully qualified names of those items that can be legally imported into the local environment by the USE clause being processed. The auxiliary functions export-qualified-names and import-legal are used by import-qualified-names.
6.5.6 Concurrent Statements

\textit{(CST0) \texttt{CST [ e ] (p)(u)(t) = u(t)}}

\textit{(CST1) \texttt{CST [ con-stat con-stat* ] (p)(u)(t)}
\quad \texttt{= CST [ con-stat ] (p)(u1)(t)}
\quad \texttt{where u1 = \lambda CST [ con-stat* ] (p)(u)(t)}}

Concurrent statements are statically checked in the textual order of their appearance in the hardware description.

\textit{(CST2) \texttt{CST [ PROCESS id ref* decl* seq-stat* opt-id ] (p)(u)(t)}
\quad \texttt{= let q = find-architecture-env(t)(p) in}
\quad \texttt{let labels = third(t(q)(*LAB* ))\ in}
\quad \texttt{(id \in labels \rightarrow error(cat("Duplicate process label: "))(|q(id))),}
\quad \texttt{let t1 = enter(t(q)(*LAB* )((e,cons(id,labels)))\ in}
\quad \texttt{(~null(opt-id)\land opt-id \neq id}
\quad \rightarrow \text{error}
\quad \quad \texttt{((\text{cat("PROCESS statement ")})(id))}
\quad \quad \text{(" ended with incorrect identifier: ")(opt-id)),}
\quad \texttt{let t2 = enter(t1(q)(id))(\langle e,\text{PROCESSNAME* },p,f,ref*\rangle)\ in}
\quad \texttt{let p1 = \%p(id)\ in}
\quad \texttt{let t3 = enter(extend(t2)(p1)(\langle e,\text{UNIT* }\rangle)(\langle e,\text{PROCESS* }\rangle)\ in}
\quad \texttt{let t4 = enter(t3)(p1)(\langle e,\text{SENSE* }\rangle)\ in}
\quad \texttt{let t5 = enter(t4)(p1)(\langle e,\text{opt-id} \rangle)\ in}
\quad \texttt{let t6 = enter(t5)(p1)(\langle e,\text{opt-id} \rangle)\ in}
\quad \texttt{SLT [ ref* ] (p1)(u2)(t)\ in}
\quad \texttt{\quad \text{where u2 = \lambda DT [ decl* ] (p1)(tt)(u1)(t)}}
\quad \texttt{\quad \text{where u1 = \lambda SST [ seq-stat* ] (p1)(u)(t))\ in}
\quad \texttt{\text{find-architecture-env(t)(p)}}
\quad \texttt{\quad \text{=} (null(p)\lor tag(t(p)(*UNIT* )))\Rightarrow \text{ARCHITECTURE* } \rightarrow p,}
\quad \texttt{\text{find-architecture-env(t)(rest(p)))\in}

\textit{(CST3) \texttt{CST [ SEL-SIGASSN delay-type id expr ref selected-waveform* ] (p)(u)(t)}
\quad \texttt{= let expr* = cons(expr,}
\quad \texttt{\quad \text{collect-expressions-from-selected-waveforms}
\quad \texttt{\quad \quad \text{(selected-waveform*)}}\ in}
\quad \texttt{\text{let ref* = delete-duplicates}
\quad \texttt{\quad \quad \text{(collect-signals-from-expr-list(expr*)}(t)(p)(e))\ in}
\quad \texttt{\text{let case-alt* = construct-case-alternatives}
\quad \texttt{\quad \quad \text{(ref)(delay-type)(selected-waveform*})\ in}
\quad \texttt{\text{let case-stat = (CASE \texttt{\quad expr,case-alt*})\ in}
\quad \texttt{\text{let process-stat = (PROCESS \texttt{\quad id,ref*},\texttt{\quad e,\texttt{(case-stat),id})\ in}
\quad \texttt{\text{CST [ process-stat ] (p)(u)(t))}}
\quad \texttt{\text{\text{collect-expressions-from-selected-waveforms(selected-waveform*)}}
\quad \texttt{\text{=} (null(selected-waveform*)\rightarrow \epsilon,}
\quad \texttt{\text{let selected-waveform = hd(selected-waveform*)\ in}
\quad \texttt{\text{let waveform = second(selected-waveform*}
\quad \texttt{\quad \text{and discrete-range* = third(selected-waveform*)}\ in}
\quad \texttt{\text{let transaction-exprs = collect-transaction-expressions(second(waveform))\ in}}
\quad \texttt{\text{nconc (transaction-exprs,}
\quad \texttt{\quad \text{cons(second(discrete-range*)),}
\quad \texttt{\quad \text{cons(third(discrete-range*),}
\quad \texttt{\quad \text{collect-expressions-from-selected-waveforms}
\quad \texttt{\quad \text{(tl(selected-waveform*))))))))}}
67
collect-trans-action-expressions(\(trans^*\))
= (null(\(trans^*\)) → \(\epsilon\),
    let transaction = hd(\(trans^*\)) in
    cons(second(transaction), collect-trans-action-expressions(tl(\(trans^*\))))))

collect-signals-from-expr-list(expr*) \((t)(p)(signal-refs)\)
= (null(expr*) → signal-refs,
    let expr = hd(expr*) in
    collect-signals-from-expr
    (expr)(t)(p)(collect-signals-from-expr-list(tl(expr*))(t)(p)(signal-refs)))

collect-signals-from-expr(expr)(t)(p)(signal-refs)
= (is-ref?(\(expr\))
   → let d = lookup-obj-desc(expr)(p)(t) in
      (is-sig?(type(d)) → cons(expr, signal-refs), signal-refs),
      is-paggr?(\(expr\)) → collect-signals-from-expr-list(second(\(expr\)))(t)(p)(signal-refs),
      is-unary-op?(hd(\(expr\))) → collect-signals-from-expr
      (second(\(expr\)))(t)(p)
      (collect-signals-from-expr(third(\(expr\)))(t)(p)(signal-refs))),
      is-binary-op?(hd(\(expr\))) ∨ is-relational-op?(hd(\(expr\)))
      → collect-signals-from-expr
      (second(\(expr\)))(t)(p)
      (collect-signals-from-expr(third(\(expr\)))(t)(p)(signal-refs)),
      signal-refs)

lookup-obj-desc(ref)(p)(t)
= let name = second(ref) in
    let id+ = (cons(last(name)) → rest(name), name) in
    let q = access(rest(id+))(p)(t) in
    lookup-desc(t)(q)(last(id+))

access(id*)(p)(t)
= (null(id*) → p,
    let id = hd(id*) in
    let d = lookup(t)(p)(id) in
    (d = *UNBOUND* → error(cat("Unbound identifier: ")(id)),
     access(tl(id*))(\(\%\)(path(d))(idf(d)))(t)))

lookup-desc(t)(p)(id)
= let d = t(p)(id) in
  (d = *UNBOUND* → lookup-desc(t)(rest(p))(id), d)

construct-case-alternatives(ref)(delay-type)(selected-waveform*)
= (null(selected-waveform*) → \(\epsilon\),
    let selected-waveform = hd(selected-waveform*) in
    let waveform = second(selected-waveform)
    and discrete-range+ = third(selected-waveform) in
    let sig-assn-stat = (SIGASSN,delay-type,ref,waveform) in
    let case-alt = (CASECHOICE,discrete-range+,sig-assn-stat) in
    cons(case-alt,
      construct-case-alternatives(ref)(delay-type)(tl(selected-waveform*))))

(CST4) \(\text{CST} \{ \text{COND-SIGASSN} \} \) delay-type id ref cond-waveform* waveform \(\{ \) (p)(u)(t)
= let expr* = aconc
  \(\{ \) collect-expressions-from-conditional-waveforms
  \(\) cond-waveform*),

68
collect-transaction-expressions(second(waveform))) in
let ref* = delete-duplicates
(collect-signals-from-expr-list(expr*)(t)(p)(e)) in
(null(cond-waveform*))
→ let sig-assn-stat = (SIGASSN,delay-type,ref,waveform) in
let process-stat = (PROCESS,id,ref*,e,(sig-assn-stat),id) in
CST[] process-stat[] (p)(u)(t),
let cond-part+ = construct-cond-parts
(ref)(delay-type)(cond-waveform*)
and else-part = ((SIGASSN,delay-type,ref,waveform)) in
let if-stat = (IF,cond-part+,else-part) in
let process-stat = (PROCESS,id,ref*,e,(if-stat),id) in
CST[] process-stat[] (p)(u)(t)
collect-expressions-from-conditional-waveforms(cond-waveform*)
= (null(cond-waveform*)→ e,
let cond-waveform = hd(cond-waveform*) in
let waveform = second(cond-waveform)
and condition = third(cond-waveform) in
let transaction-exprs = collect-transaction-expressions(second(waveform)) in
nconc
(transaction-exprs,
cons(condition,
collect-expressions-from-conditional-waveforms(tl(cond-waveform*)))))
construct-cond-parts(ref)(delay-type)(cond-waveform*)
= (null(cond-waveform*)→ e,
let cond-waveform = hd(cond-waveform*) in
let waveform = second(cond-waveform)
and condition = third(cond-waveform) in
let sig-assn-stat = (SIGASSN,delay-type,ref,waveform) in
let cond-part = (condition,(sig-assn-stat)) in
cons(cond-part,construct-cond-parts(ref)(delay-type)(tl(cond-waveform*)))))

6.5.7 Sensitivity Lists

(SLT0) SLT[] e[] (p)(u)(t) = u(t)
(SLT1) SLT[] ref*[] (p)(u)(t)
= SLT[] ref[] (p)(u)(t)
where u_1 = \lambda. SLT[] ref[] (p)(u)(t)

The refs in the sensitivity list of a PROCESS statement are checked in sequential order.

(SLT2) SLT[] REF name[] (p)(u)(t)
= let expr = ref in
ET[] expr[] (p)(k)(t)
where
k = \lambda(w,e).t.
let d = tdesc(w) in
(~is-sig?(w)
→ error
→ (cat("Non-signal in process sensitivity list: ")(ref)),
let d_1 = lookup(t)(p)(*SENS*) in
let t_1 = enter
(t)(p)(*SENS*)
(<e,(cons(SlX[] ref[] (p)(t),sensitivity(d_1)))) in
u(t_1))

69
6.5.8 Sequential Statements

(SST0) $\text{SST} [ \varepsilon ] (p)(c)(t) = c(t)$

(SST1) $\text{SST} [\text{seq-stat seq-stat*}] (p)(c)(t)$
   $= \text{SST} [\text{seq-stat}] (p)(c_1)(t)$
   where $c_1 = \lambda t.\text{SST} [\text{seq-stat*}] (p)(c)(t)$

Sequential statements are statically checked in the textual order of their appearance in the hardware description.

(SST2) $\text{SST} [\text{NULL}] (p)(c)(t) = c(t)$

NULL statements require no checking.

(SST3) $\text{SST} [\text{VARASSN ref expr}] (p)(c)(t)$
   $= \text{let expr}_0 = \text{ref in}$
   $\text{ET} [\text{expr}_0] (p)(k)(t)$
   where
   $k = \lambda (w,e),t.$
   let $d = \text{tdesc}(w)$ in
   $(-\text{is-var?(w)}$
   $\rightarrow \text{error}$
   $\text{(cat("Illegal target in variable assignment statement: ")})$
   (seq-stat)),
   $(-\text{is-writable?(w)}$
   $\rightarrow \text{error(cat("Read-only variable: ")(namef(d)))},$
   $\text{RT} [\text{expr}] (p)(k_1)(t)$
   where
   $k_1 = \lambda (w_1,e_1),t.$
   let $d_1 = \text{tdesc}(w_1)$ in
   (match-types($d,d_1$) $\rightarrow$ c(t),
   $\text{error(cat("Assignment type mismatch: ")(d)(d_1))))$

find-process-env(t)(p)
$= (\text{null}(p) \vee \text{tag}(t)(\text{*UNIT*}) = \text{*PROCESS*} \rightarrow p, \text{find-process-env}(t)(\text{rest}(p)))$

First the left part of a variable assignment statement is checked, and then the right part. The left part must be a variable of reference type (checked by is-var? and is-writable?), and the basic types of the left and right parts must be the same, as verified by match-types (refer to the definitions following semantic function DT5).

(SST4) $\text{SST} [\text{SIGASSN delay-type ref waveform}] (p)(c)(t)$
   $= \text{let expr = ref in}$
   $\text{ET} [\text{expr}] (p)(k)(t)$
   where
   $k = \lambda (w,e),t.$
   let $d = \text{tdesc}(w)$
   and $q = \text{find-process-env}(t)(p)$ in
   $(-\text{is-sig?(w)}$
   $\rightarrow \text{error}$
   $\text{(cat("Illegal target of signal assignment statement: ")})$
   (namef(d))).
-is-writable?(w) → error
  (cat("Read-only signal: ")(namef (d)),
null(q)
→ error
  (cat("Sequential signal assignment statement not in a process: ")
  (seq-stat)),
let d1 = lookup-obj-desc(ref)(p)(t) in
  (null(process(d1))
→ let t1 = enter (path(d1))(idf(d1))
  (<e,*OBJECT*,path(d1),exported(d1),type(d1),
  value(d1),last(q)>) in
c1(t1),
process(d1) = last(q) → c1(t),
error
  (cat("Target of signal assignments in multiple processes: ")
  (namef(d1))))
where
c1 = λt1,W (waveform) (p)(k1)(t)
  where
  k1 = λ(w1,t1).t.
let d1 = tdesc(w1) in
  (match-types(d,d1) → c(t),
  error
    (cat("Assignment type mismatch: ")
     (d)(d1))))

(SST5) SST [ IF cond-part+ else-part ] (p)(c)(t)
  = let seq-stat* = else-part in
  check-if(cond-part*)(p)(c1)(t)
  where c1 = λt.(null(seq-stat*) → c(t), SST [ seq-stat* ] (p)(c)(t))
check-if(cond-part*)(p)(c)(t)
  = (null(cond-part*) → c(t),
   let (expr,seq-stat*) = hd(cond-part*) in
   BLT [ expr ] (p)(k)(t)
   where
   k = λ(w,e,t).
   (is-boolean?(w)
    → SST [ seq-stat* ] (p)(c1)(t)
     where c1 = λ.check-if(tl(cond-part*)))(p)(c)(t),
     error(cat("Non-boolean condition in IF statement: ")(tdesc (w))))

A Stage 2 VHDL IF statement consists of one or more conditional parts (cond-parts) followed by a (possibly empty) else-part. Each cond-part consists of a test expression followed by sequential statements that are to be executed when the test expression is the first to evaluate to true; the sequential statements constituting the else-part are to be executed when none of the test expressions is true.

The cond-parts are first checked, in order, by auxiliary semantic function check-if, after which the else-part, if nonempty, is checked by SST. Checking each cond-part involves first ascertaining that the basic type of its test expression is boolean, and then invoking SST to check its sequential statements.
A Stage 2 VHDL CASE statement consists of a selector expression followed by one or more case alternatives, each consisting of sequential statements preceded either by a nonempty sequence of discrete ranges or by the reserved word OTHERS. This discrete range sequence defines a case selection set for the particular case alternative.

The Stage 2 VHDL concrete syntax allows the statements in a case alternative to be preceded by a list of discrete ranges and expressions; for uniformity, in the Phase 1 abstract syntax (generated by the Stage 2 VHDL parser) these expressions are converted into equivalent one-element discrete ranges.

A CASE statement must be checked for the following:

- The basic types of all the case selection sets (and thus of the expressions that define the discrete ranges) must be the same, which must match that of the selector expression. In Stage 2 VHDL, the only such basic types are BOOLEAN, BIT, INTEGER, and enumeration types (including CHARACTER).

- Every expression of every discrete range in a CASE statement must be static, i.e., must have a value defined by Phase 1. This enables the contents of each case selection set to be determined during Phase 1. The OTHERS alternative, if present, defines a case selection set that is the complement of the union of the other case selection sets with respect to the set of values associated with the basic type. The BOOLEAN basic type is associated with the set of truth values \{FALSE, TRUE\}, the BIT basic type with the set of bit values \{0, 1\}, the INTEGER basic type with the set of integers \{-2, -1, 0, 1, 2, ...\}, the CHARACTER basic type with the set \{(CHAR 0), ..., (CHAR 127)\} of ASCII-128 character representations, and an arbitrary enumeration type with the set of its enumeration literals.
- The selection sets for each case alternative must be *mutually disjoint*, and their union must be the set associated with the basic type of the selector expression. The case selection subsets defined by the discrete ranges within each case alternative need not be disjoint. Note that a CASE statement with a selection expression of basic type INTEGER must have an OTHERS alternative, as the set of integers cannot be covered by a finite number of case alternatives, each with only a finite number of (finite) discrete ranges.

The basic type of the selector expression is first determined. Then semantic function AT is invoked with this basic type to check the case alternatives. Refer to the discussion of AT, which returns the union of the case selection sets associated with all of the case alternatives, a union that must cover the set associated with the selector expression's basic type.

(SST7) $$\text{SST} [\text{LOOP id seq-stat* opt-id \{p\}(c)(t)}]$$

$$= \text{let } q = \text{find-looplabel-env}(t)(p) \text{ in}$$

$$\text{let labels = third}(t(q)(*\text{LAB*})) \text{ in}$$

$$(\text{id} \in \text{labels} \rightarrow \text{error}(\text{cat("Duplicate loop label: ")}(\$q)(\text{id})),$$

$$\text{let } t_1 = \text{enter}(t(q)(*\text{LAB*})((\epsilon,\text{cons}(\text{id},\text{labels}))) \text{ in}$$

$$(\text{null}(\text{opt-id}) \land \text{opt-id} \neq \text{id}$$

$$\rightarrow \text{error}$$

$$(\text{cat("Loop ")}(\text{id}))(\text{" ended with incorrect identifier: ")}(\text{opt-id}),$$

$$\text{let } t_2 = \text{enter}(t_1)(q)(\text{id})(<\epsilon,*\text{LOOPNAME*},p>) \text{ in}$$

$$\text{let } p_1 = \%/(p)(\text{id}) \text{ in}$$

$$\text{let } t_3 = \text{extend}(t_2)(p)(\text{id})(p_1)(*\text{UNIT*})(<\epsilon,*\text{LOOP*}>) \text{ in}$$

$$\text{let } t_4 = \text{enter}(t_3)(p_1)(*\text{LAB*})(<\epsilon,\text{e}>) \text{ in}$$

$$\text{let } t_5 = \text{enter}(t_4)(p)(\text{id})(<\epsilon,*\text{LOOPNAME*},p>) \text{ in}$$

$$\text{let } c_1 = \lambda.t.\text{SST}[\text{seq-stat* \{p\}(c)(t)} \text{ in}$$

$$c_1(t_5))$$

(SST8) $$\text{SST} [\text{WHILE id expr seq-stat* opt-id \{p\}(c)(t)}]$$

$$= \text{let } q = \text{find-looplabel-env}(t)(p) \text{ in}$$

$$\text{let labels = third}(t(q)(*\text{LAB*})) \text{ in}$$

$$(\text{id} \in \text{labels} \rightarrow \text{error}(\text{cat("Duplicate loop label: ")}(\$q)(\text{id})),$$

$$\text{let } t_1 = \text{enter}(t(q)(*\text{LAB*})((\epsilon,\text{cons}(\text{id},\text{labels}))) \text{ in}$$

$$(\text{opt-id} \neq \epsilon \land \text{opt-id} \neq \text{id}$$

$$\rightarrow \text{error}$$

$$(\text{cat("Loop ")}(\text{id}))(\text{" ended with incorrect identifier: ")}(\text{opt-id}),$$

$$\text{let } t_2 = \text{enter}(t_1)(q)(\text{id})(<\epsilon,*\text{LOOPNAME*},p>) \text{ in}$$

$$\text{let } p_1 = \%/(p)(\text{id}) \text{ in}$$

$$\text{let } t_3 = \text{extend}(t_2)(p)(\text{id})(p_1)(*\text{UNIT*})(<\epsilon,*\text{LOOP*}>) \text{ in}$$

$$\text{let } t_4 = \text{enter}(t_3)(p_1)(*\text{LAB*})(<\epsilon,\text{e}>) \text{ in}$$

$$\text{let } t_5 = \text{enter}(t_4)(p)(\text{id})(<\epsilon,*\text{LOOPNAME*},p>) \text{ in}$$

$$\text{let } c_1 = \lambda.t.\text{SST}[\text{seq-stat* \{p\}(c)(t)} \text{ in}$$

$$\text{RT}[\text{expr \{p\}(k)(t_5)] \text{ where}$$

$$k = \lambda(w,e),t.$$

$$(\text{is-boolean?}(w) \rightarrow c_1(t),$$

$$\text{error}$$

$$(\text{cat("Non-boolean condition in WHILE statement: ")})$$

$$(\text{tdesc}(w))))$$

(SST9) $$\text{SST} [\text{FOR id ref discrete-range seq-stat* opt-id \{p\}(c)(t)}]$$

$$= \text{let } q = \text{find-looplabel-env}(t)(p) \text{ in}$$

$$\text{let labels = third}(t(q)(*\text{LAB*})) \text{ in}$$

73
(id ∈ labels → error(cat("Duplicate loop label: ")(q)(id))),
let t₁ = enter(t)(q)(*LAB*)(e,cons(id,labels)) in
(¬null(opt-id)∧ opt-id ≠ id → error
cat("Loop ",(id)(" ended with incorrect identifier: ")(opt-id)),
let t₂ = enter(t₁)(q)(opt-id) in
let p₁ = % (p)(id) in
let t₃ = enter(extend(t₂)(p)(id))(p₁)(*UNIT*)(p, id) in
let t₄ = enter(t₃)(p₁)(*LAB*) in
let t₅ = enter(t₄)(p)(id)(<e,*LOOPNAME*,p>) in
let (direction,expr₁,expr₂) = discrete-range in
RT [ expr₁ ] (p)(k₁)(t)
where
k₁ = λ(w₁,e₁),t.
let d₁ = tdesc(w₁) in
RT [ expr₂ ] (p)(k₂)(t)
where
k₂ = λ(w₁,e₁),t.
let d₂ = tdesc(w₂) in
(match-types(d₁,d₂)
let decl = (DEC ,CONST ,
(hd(hd(tl(ref)))))
(hd(d₁)),
(hd(tl(discrete-range)))) in
DT [ decl ] (p₁)(tt)(u)(t₁),
error
cat("Bounds type mismatch in FOR statement: ")(seq-stat))
where
u = λ₆,c₁(t₆)
where c₁ = λ₇,SST [ seq-stat* ] (p₁)(c)(t₇))

find-looplabel-env(t)(p)
= let tg = tag(t)(p)(*UNIT* )) in
(null(p)V tg ∈ (*PROCESS* *PROCEDURE* *FUNCTION* *LOOP*) → p,
find-looplabel-env(t)(rest(p)))

In Stage 2 VHDL, entering a loop (i.e., a LOOP, WHILE or FOR statement) creates a new component environment of the TSE, just as in the case of entering a subprogram (see below). The identifier that is the loop's label must be checked for uniqueness among the identifiers used thus far as labels in the innermost enclosing program unit (process, procedure, function, or loop). If unique, the identifier is appended to the innermost enclosing unit's label identifier list (bound to the special identifier *LAB* of the corresponding environment).

A *LOOPNAME* descriptor is then entered into the current environment. The resulting TSE is extended to reflect loop entry; the *UNIT* entry in the extended TSE is set to *LOOP* to associate the extended TSE with the loop, and the *LOOPNAME* descriptor is also entered into the extended TSE. This latter descriptor is used by EXIT statements contained in this loop to validate the visibility of their loop names.

In the case of a WHILE loop, the basic type of the iteration control expression is checked to be BOOLEAN, and the loop body is also checked with SST.

In the case of a FOR loop, the basic types of the iteration bounds expressions are checked to
match, the implicit declaration of the iteration parameter is processed by semantic function \( DT \), and the loop body is checked with \( SST \).

\[(SST10)\] \( SST \begin{cases} \text{EXIT opt-dotted-name opt-expr } (p)(c)(t) \\ \quad \rightarrow \text{error(cat("EXIT statement not in a loop: "))(seq-stat)}, \\ \quad (null(opt-dotted-name) \rightarrow c_1(t), \\ \quad \text{name-type(opt-dotted-name)(e)(p)(t)(v)} \\ \quad \text{where} \\ \quad v = \lambda w.(tag(tdesc(w)) \neq *LOOPNAME* \\ \quad \rightarrow \text{error(cat("Not a loop name: ")(namef(tdesc(w)))),} \\ \quad c_1(t))) \\ \end{cases} \)

where

\[ c_1 = \lambda t.(null(opt-expr) \rightarrow c(t), \]

\[ \text{let expr = opt-expr in} \]

\[ ET \begin{cases} \text{expr } (p)(k)(t) \\ \quad \text{where} \\ \quad k = A(w,e),t. \\ \quad \text{(is-boolean?(w) \rightarrow c(t),} \\ \quad \text{error} \\ \quad \text{(cat("Non-boolean condition in EXIT statement: ")})} \\ \quad (tdesc(w)))) \\ \end{cases} \]

An EXIT statement must be contained within a loop; otherwise, an error is raised. If an exit control expression is present, its basic type is checked; if not \textbf{BOOLEAN}, an error is raised.

\[(SST11)\] \( SST \begin{cases} \text{CALL ref } (p)(c)(t) \\ \quad \rightarrow \text{let expr = ref in} \]

\[ ET \begin{cases} \text{expr } (p)(k)(t) \\ \quad \text{where} \\ \quad k = \lambda (w,e),t. \\ \quad \text{(is-boolean?(w) \rightarrow c(t),} \\ \quad \text{error} \\ \quad \text{("Invalid procedure call"))} \\ \end{cases} \]

A procedure call statement boils down to an expression that is a Stage 2 VHDL name. This expression is checked by \( ET \), and must have a \textbf{VOID} basic type.

\[(SST12)\] \( SST \begin{cases} \text{RETURN opt-expr } (p)(c)(t) \\ \quad \rightarrow \text{let d = context(t)(p) in} \]

\[ \text{let tg = tag(d) and cname = namef(d) in} \]

\[ \text{(null(opt-expr) \rightarrow (tg \neq *PROCEDURE*} \\ \quad \rightarrow \text{error} \\ \quad \text{(cat("Return without expression in context of non-procedure: ")})} \\ \quad \text{(cname)),} \\ \text{(tg \neq *FUNCTION*} \\ \quad \rightarrow \text{error} \\ \quad \text{(cat("Return with expression in context of non-function: ")})} \\ \quad \text{(cname)),} \\ \text{let expr = opt-expr in} \]

\[ ET \begin{cases} \text{expr } (p)(k)(t) \\ \end{cases} \]

75
where
\[ k = \lambda (w, e), t. \]
\[ (-\text{tdesc}(w) \in \text{extract-rtypes(signatures(d)))) \]
\[ \rightarrow \text{error} \]
\[ (\text{cat("Incorrect return expression type in function: ")}) \]
\[ (\text{cname}), \]
\[ c(t)) \]

\text{context}(t)(\text{path})
= \text{let } d = t(\text{path})(\text{*UNIT*}) \text{ in}
\[ (d = \text{*UNBOUND*} \rightarrow \text{context}(t)(\text{rest(path)}), \]
\[ \text{case tag}(d) \]
\[ (*\text{PROCEDURE*},*\text{FUNCTION*},*\text{PACKAGE*} \rightarrow t(\text{rest(path)})(\text{last(path)}), \]
\[ \text{OTHERWISE} \rightarrow \text{context}(t)(\text{rest(path)}))) \]

\text{extract-rtypes(signatures)}
= (\text{null(signatures)} \rightarrow \epsilon, \]
\[ \text{cons(second(rtype(hd(signatures))), extract-rtypes(tl(signatures))))} \]

\text{RETURN} statements have two forms, depending on the \text{PROCEDURE} or \text{FUNCTION} context in which they can appear. Auxiliary semantic function \text{context} returns the descriptor of the smallest subprogram or package enclosing the program text whose local environment is at the end of the current path. It is first determined whether the \text{RETURN} statement is in the proper context. If so, then if the \text{RETURN} statement has an expression, its basic type must be equal to the basic type of the result type of the function in which it appears.

\begin{align*}
\text{(SST13) SST} & \text{[WAIT ref opt-expr1 opt-expr2 ](p)(c)(t)} \\
= \text{let } c_1 = \lambda t. \text{let } d = \text{lookup}(t)(p)(\text{*SENS*}) \text{ in} \\
& (-\text{null(sensitivity(d))}) \\
& \rightarrow \text{error} \\
& (\text{cat("WAIT statement ")(seq-stat)} \\
& (\text{"illegal in process with sensitivity list: ")}) \\
& (\text{last(p)})), \\
& \text{let } c_2 = \lambda t. (\text{null(opt-expr1)} \rightarrow c(t), \\
& \text{let expr2 = opt-expr2 in} \\
& \text{RT [expr2 ](p)(k_2)(t)} \\
& \text{where} \\
& k_2 = \lambda (w_2, e_2), t_2. \\
& \text{(is-time?(w_2) \rightarrow c(t_2),} \\
& \text{error} \\
& (\text{cat("Ill-typed timeout clause in WAIT statement: ")}) \\
& (\text{seq-stat)))) \text{ in} \\
& (\text{null(opt-expr1)} \rightarrow c_2(t), \\
& \text{let expr1 = opt-expr1 in} \\
& \text{RT [expr1 ](p)(k_1)(t)} \\
& \text{where} \\
& k_1 = \lambda (w_1, e_1), t_1. \\
& \text{(is-boolean?(w_1) \rightarrow c_2(t_1),} \\
& \text{error} \\
& (\text{cat("Non-boolean condition clause in WAIT statement: ")}) \\
& (\text{seq-stat)))) \text{ in} \\
& \text{check-wait-refs(seq-stat)(ref*)(p)(c_1)(t)} \\
& = (\text{null( [ref* ]} \rightarrow c(t), \\
& \text{let ref = hd(ref*)} \\
& \text{and } c_1 = \lambda t. \text{check-wait-refs(seq-stat)(tl(ref*))(p)(c)(t) in} \\
& \text{check-wait-ref(seq-stat)(ref)(p)(c_1)(t))} \]
\end{align*}
Semantic equation SST13 specifies the static semantics of the WAIT statement, which consists of a sensitivity list \textit{ref*}, an optional condition \textit{opt-expr1}, and an optional timeout expression \textit{opt-expr2}. First, auxiliary semantic function \textit{check-wait-ref} recursively traverses the sensitivity list, checking that each \textit{ref} denotes a declared signal. Next, a descriptor for the special identifier \textit{*SENS*} is looked up, and if its \textit{sensitivity} field is nonempty, then the WAIT statement illegally appears inside a PROCESS statement with a sensitivity list. If present, the condition is checked to have basic type \textbf{BOOLEAN}. Finally, if present, the timeout expression is checked to have basic type \textbf{TIME}.

### 6.5.9 Case Alternatives

\textbf{(AT0)} \textbf{AT} [ ε ] (d)(p)(y)(t) = y(\text{emptyset})(t)

\textbf{(AT1)} \textbf{AT} [ case-alt* case-alt ] (d)(p)(y)(t) = \textbf{AT} [ case-alt* ] (d)(p)(y1)(t)

\textbf{where}

\textbf{AT} [ case-alt ] (d)(p)(y2)(t1)

\textbf{where}

\textbf{AT} [ case-overlap(d)((h1,h2))] → \textbf{error}

\textbf{(AT2)} \textbf{AT} [ CASECHOICE discrete-range+ seq-stat* ] (d)(p)(y)(t)

\textbf{where}

\textbf{AT} [ discrete-range+] (d)(p)(ya)(t)

\textbf{where}

\textbf{AT} [ CASEOTHERS seq-stat* ] (d)(p)(y)(t) = \textbf{SST} [ seq-stat* ] (p)(c)(t)

\textbf{where}

\textbf{SST} [ seq-stat* ] (p)(c)(t)

\textbf{where}

\textbf{SST} [ seq-stat* ] (p)(c)(t)

\textbf{where}

\textbf{SST} [ seq-stat* ] (p)(c)(t)

\textbf{where}

\textbf{SST} [ seq-stat* ] (p)(c)(t)
Semantic function AT processes each case alternative in turn, beginning with the last one. As the case selection set of each alternative is computed, it is checked for disjointness with the union of the selection sets of the preceding alternatives. If disjoint, then the union of these two case selection sets is returned; otherwise an error is raised.

Note that the case selection set of an OTHERS alternative (represented by CASEOTHERS in the abstract syntax) is always disjoint from the union of the selection sets of the preceding alternatives, because (1) a CASE statement can contain at most one such alternative; (2) if such an alternative is present, it must be the last alternative; and (3) the case selection set of an OTHERS alternative is the relative complement of the union of the case selection sets of the preceding alternatives.

AT invokes the semantic function DRT to compute the case selection set defined by the sequence of discrete ranges of a particular case alternative.

6.5.10 Discrete Ranges

(DRT0) \[ \text{DRT} \{ e \} (d)(p)(y)(t) = y(\text{emptyset})(t) \]

(DRT1) \[ \text{DRT} \{ \text{discrete-range} \} (d)(p)(y)(t) = \text{DRT} \{ \text{discrete-range} \} (d)(p)(y_1)(t) \]
\[ \text{where} \]
\[ y_1 = \lambda h_1, t_1. \]
\[ \text{DRT} \{ \text{discrete-range} \} (d)(p)(y_2)(t_1) \]
\[ \text{where} y_2 = \lambda h_2, t_2. y(h_1 \cup h_2)(t_2) \]

A sequence of discrete ranges is processed in order, from left to right.

(DRT2) \[ \text{DRT} \{ \text{discrete-range} \} (d)(p)(y)(t) = \text{let} \ (\text{direction}, \text{expr}_1, \text{expr}_2) = \text{discrete-range} \text{ in} \]
\[ \text{RT} \{ \text{expr} \} (p)(k_1)(t) \]
\[ \text{where} \]
\[ k_1 = \lambda (w_1, e_1), t_1. \]
\[ (\text{tdesc}(w_1) \neq d \rightarrow \text{error}("Case type mismatch"), \]
\[ e_1 = \ast \text{UNDEF} \rightarrow \text{error}("Non-static case expression"), \]
\[ \text{RT} \{ \text{expr} \} (p)(k_2)(t_1) \]
\[ \text{where} \]
\[ k_2 = \lambda (w_2, e_2), t_2. \]
\[ (\text{tdesc}(w_2) \neq d \rightarrow \text{error}("Case type mismatch"), \]
\[ e_2 = \ast \text{UNDEF} \rightarrow \text{error}("Non-static case expression"), \]
\[ y(mk-set(d)((\text{direction}, e_1, e_2)))(t_2)) \]
mk-set(d)(direction,e₁,e₂) = (case tag(d)
  *BOOL*
    → (e₁ = e₂ → {e₁},
       (direction = TO → (e₁ = FALSE ∧ e₂ = TRUE → {FALSE,TRUE }, emptyset),
        (e₁ = TRUE ∧ e₂ = FALSE → {TRUE,FALSE }, emptyset))),
  *BIT*
    → (e₁ = e₂ → {e₁},
       (direction = TO → (e₁ = 0 ∧ e₂ = 1 → {0,1}, emptyset), (e₁ = 1 ∧ e₂ = 0 → {1,0}, emptyset))),
  *INT*
    → (direction = TO → {FALSE ,TRUE }, emptyset),
       (ei = TRUE ∧ e₂ = FALSE → {TRUE ,FALSE }, emptyset)),
  *ENUMTYPE*
    → (direction = TO → mk-enum-set(literals(d))(e₁)(e₂),
       mk-enum-set(reverse(literals(d)))(e₁)(e₂)),
    OTHERWISE → error(cat("Illegal CASE expression type tag: ")(tag(d))))

mk-enum-set(id+)(id₁)(id₂)
= let n₁ = position(id₁)(id+)
  and n₂ = position(id₂)(id+)
  in
  (n₂ < n₁ → e₁,
   nth-tl(n₁)(reverse(nth-tl(length(id+)-(n₂+1))(reverse(id+)))))

nth-tl(n)(x) = (n = 0 → x, nth-tl(n-1)(tl(x)))

position(a)(x) = position-aux(a)(x)(0)

position-aux(a)(x)(n)
= (null(x)→ n, (a = hd(x)→ n, position-aux(a)(tl(x))(1+n)))

reverse(x) = reverse-aux(x)(ε)

reverse-aux(x)(y) = (null(x)→ y, reverse-aux(tl(x))(cons(hd(x),y)))

Semantic function DRT receives a case selector expression’s basic type from AT. DRT detects a mismatch between the basic type of a discrete range and that of the selector expression; it also detects the presence of nonstatic expressions in a discrete range. Case selection sets are constructed by the function mk-set ("make set"), which takes a type descriptor and a pair of translated static expressions that represent a discrete range (that the expressions are static is checked in Phase 1) and returns the corresponding set of values.

6.5.11 Waveforms and Transactions

(WT1) \textit{WT} [ \textit{WAVE \ transaction}^* ] (p)(k)(t) = \textit{TRT} [ \textit{transaction}^* ] (p)(k)(t)

(TRT1) \textit{TRT} [ \textit{transaction \ transaction}^* ] (p)(k)(t)
= \textit{TRT} [ \textit{transaction} ] (p)(k₁)(t)
  where
  k₁ = λ(w₁,e₁),t₁.
  let d₁ = \textit{tdesc}(w₁) in
  (null(transaction*) → k((w₁,e₁))(t₁),
   let transaction⁺ = transaction⁺ in
   \textit{TRT} [ transaction⁺ ] (p)(k₂)(t₁)
where
\[ k_2 = \lambda (w_2, e_2), t_2. \]

let \( d_2 = \text{tdesc}(w_2) \) in
\( \text{match-types}(d_1, d_2) \)
\( \rightarrow \text{error} \)
\( \text{cat} \(" \text{Type mismatch for waveform transactions: } \)\)
\( \text{cat} \(" \text{transaction}(\text{hd}(\text{transaction}_1))\), \)
\( e_1 \neq \star \text{UNDEF} \land e_2 \neq \star \text{UNDEF} \)
\( \rightarrow (e_1 \geq e_2) \)
\( \rightarrow \text{error} \)
\( \text{cat} \(" \text{Nonascending times for waveform transactions: } \)\)
\( \text{cat} \(" \text{transaction}(\text{hd}(\text{transaction}_1))\), \)
\( k((w_2, e_2))(t_2)), \)
\( k((w_2, e_1))(t_2)) \)

\( (\text{TRT}2) \)
\( \text{TRANS} \text{expr opt-expr } (p)(k)(t) \)
\( = \text{TRANS} \text{expr } (p)(k_1)(t) \)
\( \text{where} \)
\( k_1 = \lambda (w_1, e_1), t_1. \)
\( \text{null}(\text{opt-expr}) \rightarrow k((w_1, 0))(t_1), \)
\( \text{let expr}_2 = \text{opt-expr} \in \)
\( \text{TRANS} \text{expr}_2 \ (p)(k_2)(t_1) \)
\( \text{where} \)
\( k_2 = \lambda (w_2, e_2), t_2. \)
\( \text{is-time?}(w_2) \)
\( \rightarrow \text{error} \)
\( \text{cat} \(" \text{Transaction has ill-typed time expression: } \)\)
\( \text{cat} \(" \text{transaction}(\text{tdesc}(w_2))\), \)
\( e_2 \neq \star \text{UNDEF} \)
\( \rightarrow (e_2 < 0) \)
\( \rightarrow \text{error} \)
\( \text{cat} \(" \text{Transaction has negative time expression: } \)\)
\( \text{cat} \(" \text{transaction}(e_2)\), \)
\( k((w_2, e_2))(t_2)), \)
\( k((w_1, e_2))(t_2)) \)

6.5.12 Expressions

\( (\text{ET}0) \) \( \text{ET} [ \varepsilon ] (p)(k)(t) = k((\varepsilon, \varepsilon))(t) \)

\( (\text{ET1}) \) \( \text{ET} [ \text{FALSE} ] (p)(k)(t) = k((\text{mk-type}((\text{CONST VAL}))(\text{bool-type-desc}()), \text{FALSE}))(t) \)

\( (\text{ET2}) \) \( \text{ET} [ \text{TRUE} ] (p)(k)(t) = k((\text{mk-type}((\text{CONST VAL}))(\text{bool-type-desc}()), \text{TRUE}))(t) \)

\( (\text{ET3}) \) \( \text{ET} [ \text{BIT} \ bitlit ] (p)(k)(t) = k((\text{mk-type}((\text{CONST VAL}))(\text{bit-type-desc}()), \text{B [ bitlit ]}))(t) \)

\( (\text{ET4}) \) \( \text{ET} [ \text{NUM} \ constant ] (p)(k)(t) = k((\text{mk-type}((\text{CONST VAL}))(\text{int-type-desc}()), \text{N [ constant ]}))(t) \)

\( (\text{ET5}) \) \( \text{ET} [ \text{TIME} \ constant \ time-unit ] (p)(k)(t) = \text{let} \ \text{normalized-constant} = \text{case time-unit} \)
\( \text{FS} \rightarrow \text{N [ constant ]}, \)
\( \text{PS} \rightarrow 10000\times\text{N [ constant ]}, \)
\( \text{NS} \rightarrow 1000000\times\text{N [ constant ]}, \)
US → 1000000000×N [ constant ] ,
MS → 100000000000×N [ constant ] ,
SEC → 100000000000000×N [ constant ] ,
MIN → 60×(100000000000000×N [ constant ] ),
HR → 3600×(100000000000000×N [ constant ] ),
OTHERWISE
→ error
   (cat("Illegal unit name for physical type TIME: ")
   (time-unit))) in
k((mk-type((CONST VAL) )(time-type-desc()),normalized-constant))(t)

(ET6) ET [ CHAR constant ] (p)(k)(t)
= let expr = (CHAR ,constant) in
  let d = lookup(t)((STANDARD ) )(expr) in
  k((type(d),idf(d)))(t)

(ET7) ET [ BITSTR bit-lit* ] (p)(k)(t)
= let expr* = bit-lit* in
  (null(expr*)
   → k((mk-type((CONST VAL) )(lookup(t)(e)(BIT_VECTOR )),*UNDEF* ))(t),
   list-type(expr*)(p)(t)(vv)
     where vv = λw".array-type(BIT_VECTOR )(expr*)(w*)(t)(p)(k))

(ET8) ET [ STR char-lit* ] (p)(k)(t)
= let expr* = char-lit* in
  (null(expr*)→ k((mk-type((CONST VAL) )(lookup(t)(e)(STRING )),*UNDEF* ))(t),
   list-type(expr*)(p)(t)(vv)
     where vv = λw".array-type(STRING )(expr*)(w*)(t)(p)(k))

array-type(array-type-name)(expr*)(w*)(t)(p)(k)
= let d = tdesc(hd(w*)) in
  (chk-array-type(d)(tl(w*))
    → let array-type-desc = array-type-desc
      (new-array-type-name(array-type-name))(e)(p)(tt)
      (TO )((NUM 1) )((NUM ,length(w*)))(d) in
    k((mk-type(tmode(hd(w*)))(array-type-desc),*UNDEF* ))(t),
    error(cat("Array aggregate of inhomogeneous type: ")(expr*))
  )
chk-array-type(d)(w*)
= (null(w*)→ tt,
  match-types(d)(tdesc(hd(w*))→ chk-array-type(d)(tl(w*)),
  ff)

(ET9) ET [ REF name ] (p)(k)(t)
= name-type(name)(e)(p)(t)(v)
  where
  v = λw(let d = tdesc(w) in
    (second(tmode(w))= TYP
      → error(cat("Wrong context for a type: ")(namef(d))),
    tag(d)= *OBJECT* → k((type(d),value(d)))(t),
    tag(d)= *ENUMELT* → k((type(d),idf(d)))(t),
    k((w,*UNDEF* )))(t))
\( \text{(ET10)} \) \( \text{ET} [\text{PAGGR} \text{ expr* }] (p)(k)(t) = (\text{length}(\text{expr*})) = 1 \) 
\( \text{→ let expr = hd(\text{expr*}) in} \) 
\( \text{ET} [\text{expr}] (p)(k)(t), \) 
\( \text{list-type}(\text{expr*})(p)(t)(\text{vv}) \) 
\( \text{where vv = } \lambda w^*.\text{array-type}(\text{ANONYMOUS})*(\text{expr*})(w^*)(t)(p)(k) \)

\( \text{(ET11)} \) \( \text{ET} [\text{unary-op} \text{ expr}] (p)(k)(t) = \text{RT} [\text{expr}] (p)(k_1)(t) \) 
\( \text{where } k_1 = \lambda (w,e), t. \text{OT1} [\text{unary-op}] (k)((w,e))(t) \)

\( \text{(ET12)} \) \( \text{ET} [\text{binary-op} \text{ expr1} \text{ expr2}] (p)(k)(t) = \text{RT} [\text{expr1}] (p)(k_1)(t) \) 
\( \text{where} \) 
\( k_1 = \lambda (w_1, e_1), t. \) 
\( \text{RT} [\text{expr2}](p)(k_2)(t) \) 
\( \text{where} \) 
\( k_2 = \lambda (w_2, e_2), t. \) 
\( \text{OT2} [\text{binary-op}] (k)((w_1, e_1))((w_2, e_2))(t) \)

\( \text{(ET13)} \) \( \text{ET} [\text{relational-op} \text{ expr1} \text{ expr2}] (p)(k)(t) = \text{RT} [\text{expr1}](p)(k_1)(t) \) 
\( \text{where} \) 
\( k_1 = \lambda (w_1, e_1), t. \) 
\( \text{RT} [\text{expr2}](p)(k_2)(t) \) 
\( \text{where} \) 
\( k_2 = \lambda (w_2, e_2), t. \) 
\( \text{OT2} [\text{relational-op}] (k)((w_1, e_1))((w_2, e_2))(t) \)

\( \text{(RT1)} \) \( \text{RT} [\text{expr}](p)(k)(t) = \text{ET} [\text{expr}](p)(k_1)(t) \) 
\( \text{where} \) 
\( k_1 = \lambda (w, e), t. \) 
\( \text{let } tm = \text{tmode}(w) \) 
\( \text{and } d = \text{tdesc}(w) \text{ in} \) 
\( \text{(second}(tm)= \text{ACC} \) 
\( \text{→ error(cat("Non-value (an access): ")(namef}(d))), \) 
\( \text{second}(tm)= \text{OUT} \) 
\( \text{→ error} \) 
\( \text{(cat("Cannot dereference formal OUT parameter: "))} \) 
\( \text{(namef}(d))), \) 
\( \text{second}(tm)= \text{VAL} \wedge \text{is-void-tdesc?}(d) \) 
\( \text{→ error(cat("Void value: ")(namef}(d))), \) 
\( \text{let } w_1 = ((\text{second}(tm)= \text{AGR} \rightarrow \text{(DUMMY AGR) }, \text{(DUMMY VAL)} ),\text{tdesc}(w)) \text{ in} \) 
\( k((w_1, e))(t)) \)

\( \text{(OT1.1)} \) \( \text{OT1} [\text{unary-op}] (k)(w,e)(t) \) 
\( = \text{let } d = \text{tdesc}(w) \text{ in} \) 
\( \text{argtypes1}(\text{unary-op})(d) \) 
\( \text{→ k}((\text{restype1}(\text{unary-op})(d),\text{resval1}(\text{unary-op})(e)(d)))(t), \) 
\( \text{error(cat("Argument type mismatch for unary operator: ")(d)(\text{unary-op})))} \)

\( \text{argtypes1}(\text{unary-op})(d) = \text{(case unary-op} \) 
\( \text{NOT} \rightarrow \text{is-boolean-tdesc?}(d) \vee \text{is-bit-tdesc?}(d), \) 
\( \text{PLUS},\text{NEG},\text{ABS} \) 
\( \rightarrow \text{is-integer-tdesc?}(d) \vee \text{is-time-tdesc?}(d), \) 
\( \text{OTHERWISE} \rightarrow \text{error} \) 
\( \text{(cat("Unrecognized Stage 2 VHDL unary operator: ")(d)(\text{unary-op})))} \)
\[ \text{argtypes1-error}(\text{unary-op})(d) = \text{error(cat("Unary operator ")}(\text{unary-op}(" not implemented for type: ")(d)) \]

\[ \text{retype}(\text{unary-op})(d) = \text{mk-type}((\text{DUMMY VAL}) )(d) \]

\[ \text{resvall}(\text{unary-op})(e)(d) = (e = \text{UNDEF*} \rightarrow \text{UNDEF*}, \]

\[ \text{case unary-op} \]

\[ \text{NOT} \rightarrow (\text{is-boolean-tdesc?}(d) \rightarrow \neg e, \]

\[ \text{is-bit-tdesc?}(d) \rightarrow \text{invert-bit}(e), \]

\[ \text{UNDEF*}), \]

\[ \text{PLUS} \rightarrow e, \]

\[ \text{NEG} \rightarrow -e, \]

\[ \text{ABS} \rightarrow \text{abs}(e), \]

\[ \text{OTHERWISE} \rightarrow \text{UNDEF*} ) \]

\[ \text{invert-bit}(\text{bitlit}) = \text{mk-bit-simp-symbol}((-\text{bitlit})+1) \]

\[ \text{mk-bit-simp-symbol}(\text{bitlit}) = (\text{case bitlit} \]

\[ 0 \rightarrow (\text{BS} 0 1), \]

\[ 1 \rightarrow (\text{BS} 1 1), \]

\[ \text{OTHERWISE} \rightarrow \text{error(cat("Can't construct simp symbol for bit: ")(bitlit)))} \]

\[ (\text{OT2.1}) \text{OT2 } [\text{binary-op }] (k)(w_1,e_1)(w_2,e_2)(t) \]

\[ = \text{let } d_1 = \text{tdesc}(w_1), \]

\[ \text{and } d_2 = \text{tdesc}(w_2) \text{ in} \]

\[ (\text{argtypes2}(\text{binary-op})(d_1,d_2)) \]

\[ \rightarrow k((\text{retype2}(\text{binary-op})(d_1,d_2)), \]

\[ \text{resval2}((d_1,d_2))(\text{binary-op}((e_1,e_2)))(t), \]

\[ \text{error} \]

\[ (\text{cat("Argument type mismatch for binary operator: ")(binary-op)})) \]

\[ (\text{OT2.2}) \text{OT2 } [\text{relational-op }] (k)(w_1,e_1)(w_2,e_2)(t) \]

\[ = \text{let } d_1 = \text{tdesc}(w_1), \]

\[ \text{and } d_2 = \text{tdesc}(w_2) \text{ in} \]

\[ (\text{argtypes2}(\text{relational-op})(d_1,d_2)) \]

\[ \rightarrow k((\text{mk-type}((\text{DUMMY VAL}) )\text{bool-type-desc}()), \]

\[ \text{resval2}((d_1,d_2))(\text{relational-op}((e_1,e_2)))(t), \]

\[ \text{error} \]

\[ (\text{cat("Argument type mismatch for relational operator: ")(relational-op)})) \]

\[ \text{argtypes2}(\text{op})(d_1,d_2) \]

\[ = (\text{case op} \]

\[ \text{AND,NAND,OR,NOR,XOR} ) \]

\[ \rightarrow (\text{case hd}(d_1) \]

\[ \text{BOOLEAN} \rightarrow \text{is-boolean-tdesc?}(d_2) \lor \text{argtypes2-error}(\text{op})(d_1)(d_2), \]

\[ \text{BIT} \rightarrow \text{is-bit-tdesc?}(d_2) \lor \text{argtypes2-error}(\text{op})(d_1)(d_2), \]

\[ \text{OTHERWISE} \rightarrow \text{argtypes2-error}(\text{op})(d_1)(d_2), \]

\[ \text{(ADD, SUB) } \]

\[ \rightarrow (\text{case hd}(d_1) \]

\[ \text{(INTEGER, REAL, TIME) } \rightarrow d_1 = d_2 \lor \text{argtypes2-error}(\text{op})(d_1)(d_2), \]

\[ \text{OTHERWISE} \rightarrow \text{argtypes2-error}(\text{op})(d_1)(d_2), \]

\[ \text{MUL} \]

83
→ (case hd(d1)
   (INTEGER, REAL)
      → d1 = d2 ∨ is-time-tdesc?(d2),
      TIME
      → is-integer-tdesc?(d2) ∨ is-real-tdesc?(d2),
      OTHERWISE → argtypes2-error(op)(d1)(d2)),
DIV
→ (case hd(d1)
   (INTEGER,REAL) → d1 = d2 ∨ argtypes2-error(op)(d1)(d2),
   TIME
   → is-integer-tdesc?(d2) ∨ is-real-tdesc?(d2),
   OTHERWISE → argtypes2-error(op)(d1)(d2)),
(MOD, REM)
→ (case hd(d1)
   INTEGER → is-integer-tdesc?(d1)∨ argtypes2-error(op)(d1)(d2),
   OTHERWise → argtypes2-error(op)(d1)(d2)),
EXP
→ (case hd(d1)
   (INTEGER, REAL) → is-integer-tdesc?(d1)∨ argtypes2-error(op)(d1)(d2),
   OTHERWise → argtypes2-error(op)(d1)(d2)),
CONCAT
→ (is-bit-tdesc?(d1)
   → is-bit-tdesc?(d2) ∨ is-bitvector-tdesc?(d2),
   (is-bit-tdesc?(d2)
   → is-bit-tdesc?(d1) ∨ is-bitvector-tdesc?(d1),
   (is-array-tdesc?(d1) ∧ is-array-tdesc?(d2)
   → match-array-type-names(idf(d1), idf(d2))
   ∧ match-types(elty(d1), elty(d2)),
   argtypes2-error(op)(d1)(d2))),
(EQ, NE)
→ match-types(d1, d2) ∨ argtypes2-error(op)(d1)(d2),
(LT, LE, GT, GE)
→ (is-scalar-tdesc?(d1) ∧ is-scalar-tdesc?(d2)
   → match-types(d1)(d2) ∨ argtypes2-error(op)(d1)(d2),
   is-bitvector-tdesc?(d1) ∧ is-bitvector-tdesc?(d2) → tt,
   argtypes2-error(op)(d1)(d2)),
OTHERWISE → error(cat("Unrecognized Stage 2 VHDL operator: ")(00))

argtypes2-error(op)(d1)(d2)
= error(cat("Operator ")" not implemented for pair of types: ")(d1)(d2))

restype2(binary-op)(d1, d2)
= (case binary-op
   (AND, NAND, OR, NOR, XOR, ADD, SUB, MOD, REM, EXP ) → mk-type((DUMMY VAL))(d1),
   MUL
   → (case hd(d1)
      (INTEGER, REAL) → mk-type((DUMMY VAL))(d2),
      TIME → mk-type((DUMMY VAL))(d1),
      OTHERWISE → error("Shouldn’t happen!")),
DIV
→ (case hd(d1)
   (INTEGER, REAL) → mk-type((DUMMY VAL))(d2),
   TIME
   → (case hd(d2)
      (INTEGER, REAL) → mk-type((DUMMY VAL))(d1),
      TIME → mk-type((DUMMY VAL))(int-type-desc()),
      OTHERWISE → error("Shouldn’t happen!")),
OTHERWISE → error("Shouldn’t happen!")),

84
CONCAT → mk-type((DUMMY VAL) ) (mk-concat-tdesc(d1))(d2)),
OTHERWISE
→ error(cat("Unrecognized Stage 2 VHDL binary operator: ")(binary-op))

mk-concat-tdesc(d1)(d2)
= (is-bit-tdesc?(d1) v is-bitvector-tdesc?(d1))
→ array-type-desc
  (new-array-type-name(BIT_VECTOR ))(e)(tt)(direction(d1))(lb(d1))(e)
  (bit-type-desc()),
  let idf1 = idf(d1) in
array-type-desc
  (new-array-type-name((consp(idf1)→ hd(idf1), idf1)))(e)(tt)
  (direction(d1))(lb(d1))(e)(elty(d1))

resval2(d1,d2)(op)(e1,e2)
= (e1 = *UNDEF* v e2 = *UNDEF* → *UNDEF*,
let tg = tag(d1) in
  (case tg
   *BOOL* 
→ (case op
       AND → e1 ∧ e2,
       NAND → ~(e1 ∧ e2),
       OR → e1 V e2,
       NOR → ~(e1 V e2),
       XOR → (e1 = e2 → ff, tt),
       EQ → e1 = e2,
       NE → e1 ≠ e2,
       LT → ~e1 ∧ e2,
       LE → ~e1 V e2,
       GT → e1 ∧ ~e2,
       GE → e1 V ~e2,
       OTHERWISE
→ error
   )
  )
  (*BIT*,
→ (case op
    AND
→ (e1 = 1 ∧ e2 = 1 → mk-bit-simp-symbol(1), mk-bit-simp-symbol(0)),
    NAND
→ (e1 = 0 V e2 = 0 → mk-bit-simp-symbol(1), mk-bit-simp-symbol(0)),
    OR
→ (e1 = 1 V e2 = 1 → mk-bit-simp-symbol(1), mk-bit-simp-symbol(0)),
    NOR
→ (e1 = 0 ∧ e2 = 0 → mk-bit-simp-symbol(1), mk-bit-simp-symbol(0)),
    XOR
→ (e1 = e2 → mk-bit-simp-symbol(0), mk-bit-simp-symbol(1)),
    EQ
→ e1 = e2,
    NE
→ e1 ≠ e2,
    LT
→ e1 = 0 ∧ e2 = 1,
    LE
→ e1 = 0 V e2 = 1,
    GT
→ e1 = 1 ∧ e2 = 0,
    GE
→ e1 = 1 V e2 = 0,
    OTHERWISE
→ error
  )
  )
  (*INT*, *TIME*
→ (case op
    ADD → e1+e2,
**6.5.13 Primitive Semantic Equations**

(N1) N [ constant ] = constant

(B1) B [ bitlit ] = bitlit
7 Interphase Abstract Syntax Tree Transformation

Owing to the relative simplicity of the previous SDVS VHDL language subset, Stage 1 VHDL, Phases 1 and 2 of the Stage 1 VHDL translator were able to use the same abstract syntax.

Stage 2 VHDL is a considerably more sophisticated language subset. Consequently, it has become convenient to allow Phase 2 of the Stage 2 VHDL translator to employ a different abstract syntax for the language than does Phase 1, for reasons discussed below. Accordingly, as the final act of Phase 1 translation of a given Stage 2 VHDL hardware description, an “interphase” abstract syntax tree transformation is performed that yields a new abstract syntax tree (AST) for use by Phase 2. This transformation does not modify the original AST. Although the resulting transformed AST may resemble the original in many respects, there will also be substantial differences.

We should recall that in Phase 1, when abstract syntax trees are occasionally injected into the TSE, it is their transformed versions that are used; this occurs with array type descriptors created by functions process-slcdec and DT8, subprogram descriptors created by function process-subprog-body, and *SENS* (sensitivity list) descriptors updated with new refs by function SLT2.

7.1 Interphase Semantic Functions

The abstract syntax tree transformation is carried out by principal semantic functions DFX, ENX, ARX, PDX, DX, CSX, SLX, SSX, AX, DRX, WX, TRX, MEX, EX, and RX, with the aid of several important auxiliary semantic functions, most notably the function transform-name.

Following Phase 1 construction of the tree-structured environment (TSE), semantic function DFX is applied to the original AST to initiate the transformation, which uses (but does not modify) the TSE. Once the AST transformation is complete, Phase 1 auxiliary semantic function phase2 is invoked with the transformed AST and the TSE as syntactic and semantic arguments, respectively, to initiate Phase 2 translation (see Section 8).

Generally speaking, the AST-transforming semantic functions straightforwardly reconstruct their syntactic arguments from their transformed immediate syntactic constituents, with the following exceptions:

- transformation of PORT declarations into SIGNAL declarations
- “desugaring” of sensitivity lists in PROCESS statements: converting them into explicit final WAIT statements
- “desugaring” of concurrent signal assignment statements: converting them into equivalent PROCESS statements
- “desugaring” of secondary units of physical type TIME: converting them into the base unit FS (femtoseconds)
• disambiguation of refs as either array references or subprogram calls
• overload resolution between BOOLEAN and BIT operators
• overload resolution between INTEGER and REAL operators

7.2 Transformed Abstract Syntax of Names

An important case in point is the translation of names, e.g. refs, which are heavily overloaded: the Phase 1 semantic function name-type, which checks them and determines their type, is necessarily complex. Given the identical abstract syntax, a Phase 2 semantic function for refs would exhibit analogous complexity; instead, it was deemed preferable to transform the abstract syntax of refs into a form more suitable for Phase 2.

Thus, the abstract syntax of refs used in Phase 1 is:

```plaintext
ref ::= REF name
name ::= id | name id | name expr*
```

while the abstract syntax of refs used in Phase 2 is:

```plaintext
ref ::= REF basic-ref
basic-ref ::= modifier+
modifier ::= SREF id+ id
| INDEX expr
| SELECTOR id
| PARLIST expr*
```

Although not reflected in the syntax shown above, a basic-ref (basic reference) must begin with a simple reference SREF id+ id, which has for convenience been classified with the modifiers. The id is the root identifier, and id+ is the TSE access path for this ref. The structures following this root basic reference are called modifiers. An INDEX modifier denotes an array reference, a SELECTOR modifier denotes a record field access (not used in Stage 2 VHDL), and a PARLIST modifier denotes a subprogram call. This linear arrangement of a simple reference followed by zero or more modifiers makes the translation of refs in Phase 2 relatively straightforward, as the components of a ref are grouped from the left and thus a ref can be translated from left to right.
7.3 Interphase Semantic Equations

Most of the semantic equations for the interphase abstract syntax tree transformation, being straightforward, will be displayed without comment.

7.3.1 Stage 2 VHDL Design Files

(DFX1) \[ \text{DFX} \{ \text{DESIGN-FILE} \ \text{id} \ \text{pkg-decl*} \ \text{pkg-body*} \ \text{use-clause*} \ \text{ent-decl} \ \text{arch-body} \} (t) \]
\[ = \text{let} \ p_0 = \%((\text{id})) \ \text{in} \]
\[ \{ \text{DESIGN-FILE} \ \text{id}, \ \text{DX} \{ \text{pkg-decl*} \} (p_0)(t), \]
\[ \text{DX} \{ \text{use-clause*} \} (p_0)(t), \]
\[ \text{ENX} \{ \text{ent-decl} \} (p_0)(t), \]
\[ \text{ARX} \{ \text{arch-body} \} (p_0)(t) \]

7.3.2 Entity Declarations

(ENX1) \[ \text{ENX} \{ \text{ENTITY} \ \text{id} \ \text{port-decl*} \ \text{decl*} \ \text{opt-id} \} (p)(t) \]
\[ = \{ \text{ENTITY} \ \text{id}, \ \text{PDX} \{ \text{port-decl*} \} (p)(t), \]
\[ \text{DX} \{ \text{decl*} \} (p)(t), \]
\[ \text{opt-id} \}

7.3.3 Architecture Bodies

(ARX1) \[ \text{ARX} \{ \text{ARCHITECTURE} \ \text{id}_1 \ \text{id}_2 \ \text{decl*} \ \text{con-stat*} \ \text{opt-id} \} (p)(t) \]
\[ = \text{let} \ p_1 = \%((\text{p}(\text{id}_2))(\text{id}_1)) \ \text{in} \]
\[ \{ \text{ARCHITECTURE} \ \text{id}_1, \ \text{id}_2, \ \text{DX} \{ \text{decl*} \} (p_1)(t), \]
\[ \text{CSX} \{ \text{con-stat*} \} (p_1)(t), \]
\[ \text{opt-id} \}

7.3.4 Port Declarations

(PDX0) \[ \text{PDX} \{ \varepsilon \} (p)(t) = \varepsilon \]

(PDX1) \[ \text{PDX} \{ \text{port-decl port-decl*} \} (p)(t) \]
\[ = \text{cons}(\text{PDX} \{ \text{port-decl} \} (p)(t), \text{PDX} \{ \text{port-decl*} \} (p)(t)) \]

(PDX2) \[ \text{PDX} \{ \text{DEC PORT} \ \text{id}^* \ \text{mode type-mark opt-expr} \} (p)(t) \]
\[ = \{ \text{DEC} , \text{SIG} , \text{id}^* , \text{type-mark}, \]
\[ \text{let} \ \text{expr} = \\text{opt-expr} \ \text{in} \]
\[ \text{second}(\text{EX} \{ \text{expr} \} (p)(t)) \]

(PDX3) \[ \text{PDX} \{ \text{SLCDEC PORT} \ \text{id}^* \ \text{mode slice-name opt-expr} \} (p)(t) \]
\[ = \{ \text{SLCDEC} , \text{SIG} , \text{id}^* , \]
\[ \text{let} \ (\text{type-mark}, \text{discrete-range}) = \text{slice-name} \ \text{in} \]
\[ (\text{type-mark}, \text{DX} \{ \text{discrete-range} \} (p)(t)), \]
\[ \text{let} \ \text{expr} = \\text{opt-expr} \ \text{in} \]
\[ \text{second}(\text{EX} \{ \text{expr} \} (p)(t)) \]
7.3.5 Declarations

(DX0) $\mathbf{DX} \ [\ e \ ] (p)(t) = e$

(DX1) $\mathbf{DX} \ [\ \text{decl} \ \text{decl}^* \ ] (p)(t) = \text{cons}(\mathbf{DX} \ [\ \text{decl} \ ] (p)(t), \mathbf{DX} \ [\ \text{decl}^* \ ] (p)(t))$

(DX2) $\mathbf{DX} \ [\ \text{pkg-decl} \ \text{pkg-decl}^* \ ] (p)(t) = \text{cons}(\mathbf{DX} \ [\ \text{pkg-decl} \ ] (p)(t), \mathbf{DX} \ [\ \text{pkg-decl}^* \ ] (p)(t))$

(DX3) $\mathbf{DX} \ [\ \text{pkg-body} \ \text{pkg-body}^* \ ] (p)(t) = \text{cons}(\mathbf{DX} \ [\ \text{pkg-body} \ ] (p)(t), \mathbf{DX} \ [\ \text{pkg-body}^* \ ] (p)(t))$

(DX4) $\mathbf{DX} \ [\ \text{use-clause} \ \text{use-clause}^* \ ] (p)(t) = \text{cons}(\mathbf{DX} \ [\ \text{use-clause} \ ] (p)(t), \mathbf{DX} \ [\ \text{use-clause}^* \ ] (p)(t))$

(DX5) $\mathbf{DX} \ [\ \text{DEC} \ \text{object-class} \ \text{id}^+ \ \text{type-mark} \ \text{opt-expr} \ ] (p)(t) = (\text{DEC}, \text{object-class}, \text{id}^+, \text{type-mark},$

  let expr = opt-expr in

  second($\mathbf{EX} \ [\ expr \ ] (p)(t))$

(DX6) $\mathbf{DX} \ [\ \text{SLCDEC} \ \text{object-class} \ \text{id}^+ \ \text{slice-name} \ \text{opt-expr} \ ] (p)(t) = (\text{SLCDEC}, \text{object-class}, \text{id}^+,$

  let (type-mark, discrete-range) = slice-name in

  (type-mark, $\mathbf{DX} \ [\ \text{discrete-range} \ ] (p)(t)),$

  let expr = opt-expr in

  second($\mathbf{EX} \ [\ expr \ ] (p)(t))$

(DX7) $\mathbf{DX} \ [\ \text{ETDEC} \ \text{id}^+ \ ] (p)(t) = (\text{ETDEC}, \text{id}, \text{id}^+)$

(DX8) $\mathbf{DX} \ [\ \text{ATDEC} \ \text{id} \ \text{discrete-range} \ \text{type-mark} \ ] (p)(t) = (\text{ATDEC}, \text{id}, \mathbf{DX} \ [\ \text{discrete-range} \ ] (p)(t), \text{type-mark})$

(DX9) $\mathbf{DX} \ [\ \text{PACKAGE} \ \text{id} \ \text{decl}^* \ \text{opt-id} \ ] (p)(t) = (\text{PACKAGE}, \text{id}, \mathbf{DX} \ [\ \text{decl}^* \ ] ((\%p(id))(t)), \text{opt-id})$

(DX10) $\mathbf{DX} \ [\ \text{PACKAGEBODY} \ \text{id} \ \text{decl}^* \ \text{opt-id} \ ] (p)(t) = \text{let} \ d = t(p)(id) \ \text{in}$

  let q = $\%\text{(path}(d)))(id) \ \text{in}$

  (\text{PACKAGEBODY}, \text{id}, \mathbf{DX} \ [\ \text{decl}^* \ ] (q)(t), \text{opt-id})$

(DX11) $\mathbf{DX} \ [\ \text{PROCEDURE} \ \text{id} \ \text{proc-par-spec}^* \ \text{type-mark} \ ] (p)(t) = (\text{PROCEDURE}, \text{id}, \text{proc-par-spec}^*, \text{type-mark})$

(DX12) $\mathbf{DX} \ [\ \text{FUNCTION} \ \text{id} \ \text{func-par-spec}^* \ \text{type-mark} \ ] (p)(t) = (\text{FUNCTION}, \text{id}, \text{func-par-spec}^*, \text{type-mark})$

(DX13) $\mathbf{DX} \ [\ \text{SUBPROGBODY} \ \text{subprog-decl}^* \ \text{seq-stat}^* \ \text{opt-id} \ ] (p)(t) = \text{let} \ (\text{tg}, \text{id}, \text{par-spec}^*, \text{type-mark}) = \text{subprog-spec} \ \text{in}$

  let $p_1 = ((\%p(id))(t)$ \ \text{in}$

  (\text{SUBPROGBODY},$

  let decl = subprog-spec in$

  $\mathbf{DX} \ [\ \text{decl} \ ] (p)(t), \mathbf{DX} \ [\ \text{decl}^* \ ] (p_1)(t), \mathbf{SSX} \ [\ \text{seq-stat}^* \ ] (p_1)(t), \text{opt-id}$

(DX14) $\mathbf{DX} \ [\ \text{USE} \ \text{dotted-name}^+ \ ] (p)(t) = (\text{USE}, \text{dotted-name}^+)$
7.3.6 Concurrent Statements

(CSX0) \texttt{CSX }\texttt{[} \varepsilon \texttt{]} \texttt{(p)(t) = \varepsilon}

(CSX1) \texttt{CSX }\texttt{[} \texttt{con-stat con-stat* }\texttt{]} \texttt{(p)(t)} \\
    \texttt{= cons(CSX }\texttt{[} \texttt{con-stat }\texttt{]} \texttt{(p)(t)}, \texttt{CSX }\texttt{[} \texttt{con-stat* }\texttt{]} \texttt{(p)(t)})

(CSX2) \texttt{CSX }\texttt{[} \texttt{PROCESS id ref* decl* seq-stat* opt-id }\texttt{]} \texttt{(p)(t)} \\
    \texttt{= let } p_1 = \% \texttt{(p)(id) in} \\
    \texttt{(PROCESS ,id,DX }\texttt{[} \texttt{decl* }\texttt{]} \texttt{(p_1)(t),} \\
    \texttt{let seq-stat* }\texttt{= (null(seq-stat*) }\texttt{-> ((WAIT ,ref*,\varepsilon,\varepsilon)),} \\
    \texttt{(null(ref*) }\texttt{-> seq-stat*,} \\
    \texttt{append(seq-stat*,((WAIT ,ref*,\varepsilon,\varepsilon)))) }\texttt{in} \\
    \texttt{SSX }\texttt{[} \texttt{seq-stat* }\texttt{]} \texttt{(p_1)(t),opt-id)}

(CSX3) \texttt{CSX }\texttt{[} \texttt{SEL-SIGASSN delay-type id expr ref selected-waveform+ }\texttt{]} \texttt{(p)(t)} \\
    \texttt{= let expr* }\texttt{= cons(expr,} \\
    \texttt{collect-expressions-from-selected-waveforms} \\
    \texttt{(selected-waveform+)} \texttt{in} \\
    \texttt{let ref* }\texttt{= delete-duplicates} \\
    \texttt{(collect-signals-from-expr-list(expr*))(t)(p)(s)) \texttt{in} \\
    \texttt{let case-alt+ }\texttt{= construct-case-alternatives} \\
    \texttt{(ref)(delay-type)(selected-waveform+)} \texttt{in} \\
    \texttt{let case-stat }\texttt{= (CASE ,expr,case-alt+)} \texttt{in} \\
    \texttt{let process-stat }\texttt{= (PROCESS ,id,ref*,\varepsilon,(case-stat),id)} \texttt{in} \\
    \texttt{CSX }\texttt{[} \texttt{process-stat }\texttt{]} \texttt{(p)(t)}

(CSX4) \texttt{CSX }\texttt{[} \texttt{COND-SIGASSN delay-type id ref cond-waveform* waveform }\texttt{]} \texttt{(p)(t)} \\
    \texttt{= let expr* }\texttt{= nconc} \\
    \texttt{(collect-expressions-from-conditional-waveforms} \\
    \texttt{(cond-waveform*),} \\
    \texttt{collect-transaction-expressions(second(waveform))) }\texttt{in} \\
    \texttt{let ref* }\texttt{= delete-duplicates} \\
    \texttt{(collect-signals-from-expr-list(expr*))(t)(p)(e)) \texttt{in} \\
    \texttt{(null(cond-waveform*))} \\
    \texttt{-> let sig-assn-stat }\texttt{= (SIGASSN ,delay-type,ref,waveform)} \texttt{in} \\
    \texttt{let process-stat }\texttt{= (PROCESS ,id,ref*,\varepsilon,(sig-assn-stat),id)} \texttt{in} \\
    \texttt{CSX }\texttt{[} \texttt{process-stat }\texttt{]} \texttt{(p)(t)}, \\
    \texttt{let cond-part+ }\texttt{= construct-cond-parts} \\
    \texttt{(ref)(delay-type)(cond-waveform*)} \\
    \texttt{and else-part }\texttt{= ((SIGASSN ,delay-type,ref,waveform)) }\texttt{in} \\
    \texttt{let if-stat }\texttt{= (IF ,cond-part+,else-part)} \texttt{in} \\
    \texttt{let process-stat }\texttt{= (PROCESS ,id,ref*,\varepsilon,(if-stat),id)} \texttt{in} \\
    \texttt{CSX }\texttt{[} \texttt{process-stat }\texttt{]} \texttt{(p)(t)}

7.3.7 Sensitivity Lists

(SLX0) \texttt{SLX }\texttt{[} \varepsilon \texttt{]} \texttt{(p)(t) = \varepsilon}

(SLX1) \texttt{SLX }\texttt{[} \texttt{ref* }\texttt{]} \texttt{(p)(t) = cons(SLX }\texttt{[} \texttt{ref }\texttt{]} \texttt{(p)(t)}, \texttt{SLX }\texttt{[} \texttt{ref* }\texttt{]} \texttt{(p)(t))}

(SLX2) \texttt{SLX }\texttt{[} \texttt{REF name }\texttt{]} \texttt{(p)(t)} \\
    \texttt{= let expr = ref in} \\
    \texttt{second(EX }\texttt{[} \texttt{expr }\texttt{]} \texttt{(p)(t))}

91
### 7.3.8 Sequential Statements

(SSX1) \[ \text{SSX} \{ \text{seq-stat seq-stat*} \} (p)(t) = \text{cons}(\text{SSX} \{ \text{seq-stat} \} (p)(t), \text{SSX} \{ \text{seq-stat*} \} (p)(t)) \]

(SSX2) \[ \text{SSX} \{ \text{NULL} \} (p)(t) = (\text{NULL}) \]

(SSX3) \[ \text{SSX} \{ \text{VARASSN ref expr} \} (p)(t) = (\text{VARASSN}, \text{let} \ exp_0 = \text{ref} \text{ in} \second(\text{EX} \{ \text{expr} \} (p)(t)), \text{second}(\text{EX} \{ \text{expr} \} (p)(t))) \]

(SSX4) \[ \text{SSX} \{ \text{SIGASSN delay-type ref waveform} \} (p)(t) = (\text{SIGASSN}, \text{delay-type}, \text{let} \ expr = \text{ref} \text{ in} \second(\text{EX} \{ \text{expr} \} (p)(t)), \text{WX} \{ \text{waveform} \} (p)(t)) \]

(SSX5) \[ \text{SSX} \{ \text{IF cond-part+ else-part} \} (p)(t) = \text{let} \ seq-stat* = \text{else-part} \text{ in} \ (\text{IF}, \text{transform-if}(\text{cond-part+})(p)(t), \text{SSX} \{ \text{seq-stat*} \} (p)(t)) \]

(transform-if(\text{cond-part*})(p)(t) = (null(\text{cond-part*}) \rightarrow \epsilon, \text{let} \ expseq-stat* = \text{hd}(\text{cond-part*}) \text{ in} \ \text{cons}(\text{second}(\text{EX} \{ \text{expr} \} (p)(t)), \text{SSX} \{ \text{seq-stat*} \} (p)(t)), \ 
\text{transform-if}(\text{tl}(\text{cond-part*}))(p)(t)) \]

(SSX6) \[ \text{SSX} \{ \text{CASE expr case-alt+} \} (p)(t) = (\text{CASE}, \text{second}(\text{EX} \{ \text{expr} \} (p)(t)), \text{AX} \{ \text{case-alt+} \} (p)(t)) \]

(SSX7) \[ \text{SSX} \{ \text{LOOP id seq-stat* opt-id} \} (p)(t) = (\text{LOOP}, \text{id}, \text{SSX} \{ \text{seq-stat*} \} (%(p)(id))(t), \text{opt-id}) \]

(SSX8) \[ \text{SSX} \{ \text{WHILE id expr seq-stat* opt-id} \} (p)(t) = (\text{WHILE}, \text{id}, \text{second}(\text{EX} \{ \text{expr} \} (p)(t)), \text{SSX} \{ \text{seq-stat*} \} (%(p)(id))(t), \text{opt-id}) \]

(SSX9) \[ \text{SSX} \{ \text{FOR id ref discrete-range seq-stat* opt-id} \} (p)(t) = (\text{FOR}, \text{id}, \text{second}(\text{EX} \{ \text{ref} \} (%(p)(id))(t)), \text{DRX} \{ \text{discrete-range} \} (%(p)(id))(t), \text{SSX} \{ \text{seq-stat*} \} (%(p)(id))(t), \text{opt-id}) \]

(SSX10) \[ \text{SSX} \{ \text{EXIT opt-dotted-name opt-expr} \} (p)(t) = (\text{EXIT}, \text{opt-dotted-name}, \text{let} \ expr = \text{opt-expr} \text{ in} \ \text{second}(\text{EX} \{ \text{expr} \} (p)(t))) \]

(SSX11) \[ \text{SSX} \{ \text{CALL ref} \} (p)(t) = (\text{CALL}, \text{let} \ expr = \text{ref} \text{ in} \ \text{second}(\text{EX} \{ \text{expr} \} (p)(t))) \]

(SSX12) \[ \text{SSX} \{ \text{RETURN opt-expr} \} (p)(t) = (\text{RETURN}, \text{let} \ expr = \text{opt-expr} \text{ in} \ \text{second}(\text{EX} \{ \text{expr} \} (p)(t))) \]

(SSX13) \[ \text{SSX} \{ \text{WAIT ref* opt-expr1 opt-expr2} \} (p)(t) = \text{let} \ expr_1 = \text{opt-expr1} \text{ and} \ expr_2 = \text{opt-expr2} \text{ in} \ (\text{WAIT}, \text{MEX} \{ \text{ref*} \} (p)(t), \text{second}(\text{EX} \{ \text{expr1} \} (p)(t)), \text{second}(\text{EX} \{ \text{expr2} \} (p)(t))) \]
7.3.9 Case Alternatives

\[(AX0) \text{AX} \ [\ varepsilon ] \ (p)(t) = \varepsilon \]

\[(AX1) \text{AX} \ [\ \text{case-alt} \ \text{case-alt}^* ] \ (p)(t) = \text{cons(AX} \ [\ \text{case-alt} ] \ (p)(t),\text{AX} \ [\ \text{case-alt}^* ] \ (p)(t)) \]

\[(AX2) \text{AX} \ [\ \text{CASECHOICE} \ \text{discrete-range}^+ \ \text{seq-stat}^* ] \ (p)(t) = \text{(CASECHOICE,DRX} \ [\ \text{discrete-range}^+ ] \ (p)(t),\text{SSX} \ [\ \text{seq-stat}^* ] \ (p)(t)) \]

\[(AX3) \text{AX} \ [\ \text{CASEOTHERS} \ \text{seq-stat}^* ] \ (p)(t) = \text{(CASEOTHERS,SSX} \ [\ \text{seq-stat}^* ] \ (p)(t)) \]

7.3.10 Discrete Ranges

\[(DX0) \text{DRX} \ [\ varepsilon ] \ (p)(t) = \varepsilon \]

\[(DX1) \text{DRX} \ [\ \text{discrete-range} \ \text{discrete-range}^* ] \ (p)(t) = \text{cons(DRX} \ [\ \text{discrete-range} ] \ (p)(t),\text{DRX} \ [\ \text{discrete-range}^* ] \ (p)(t)) \]

\[(DX2) \text{DRX} \ [\ \text{discrete-range} ] \ (p)(t) = \text{let} \ (\text{direction,expr1,expr2}) = \text{discrete-range} \text{ in}
\text{(direction,second(\text{EX} \ [\ expr1 ] \ (p)(t)),second(\text{EX} \ [\ expr2 ] \ (p)(t)))} \]

7.3.11 Waveforms and Transactions

\[(WX1) \text{WX} \ [\ \text{WAVE} \ \text{transaction}^+ ] \ (p)(t) = \text{(WAVE,TRX} \ [\ \text{transaction}^+ ] \ (p)(t)) \]

\[(TRX1) \text{TRX} \ [\ \text{transaction} \ \text{transaction}^* ] \ (p)(t) = \text{(null(transaction}^*) \rightarrow \text{(TRX} \ [\ \text{transaction} ] \ (p)(t)),
\text{let} \ \text{transaction}^* = \text{transaction}^* \text{ in}
\text{cons(\text{TRX} \ [\ \text{transaction} ] \ (p)(t),\text{TRX} \ [\ \text{transaction}^+ ] \ (p)(t))}) \]

\[(TRX2) \text{TRX} \ [\ \text{TRANS} \ \text{expr} \ \text{opt-expr} ] \ (p)(t) = \text{(TRANS,second(\text{EX} \ [\ expr ] \ (p)(t)),
\text{let} \ expr1 = \text{opt-expr} \text{ in}
\text{second(\text{EX} \ [\ expr1 ] \ (p)(t)))} \]

7.3.12 Expressions

\[(MEX0) \text{MEX} \ [\ varepsilon ] \ (p)(t) = \varepsilon \]

\[(MEX1) \text{MEX} \ [\ \text{ref} \ \text{ref}^* ] \ (p)(t) = \text{cons(second(\text{EX} \ [\ \text{ref} ] \ (p)(t)),MEX} \ [\ \text{ref}^* ] \ (p)(t)) \]

\[(EX0) \text{EX} \ [\ varepsilon ] \ (p)(t) = \text{(void-type-desc(),}\varepsilon) \]

\[(EX1) \text{EX} \ [\ \text{FALSE} ] \ (p)(t) = \text{(bool-type-desc(),(FALSE )}) \]

\[(EX2) \text{EX} \ [\ \text{TRUE } ] \ (p)(t) = \text{(bool-type-desc(),(TRUE )}) \]

\[(EX3) \text{EX} \ [\ \text{BIT} \ \text{bitlit} ] \ (p)(t) = \text{(bit-type-desc(),(BIT ,bitlit))} \]

\[(EX4) \text{EX} \ [\ \text{NUM} \ \text{constant} ] \ (p)(t) = \text{(int-type-desc(),(NUM ,constant))} \]
EX [ TIME constant time-unit ] (p)(t)
= let normalized-constant = (case time-unit
    FS -> N [ constant ] ,
    PS -> 1000xN [ constant ] ,
    NS -> 1000000xN [ constant ] ,
    US -> 1000000000xN [ constant ] ,
    MS -> 100000000000xN [ constant ] ,
    SEC -> 100000000000000xN [ constant ] ,
    MIN -> 60x(1000000000000000xN [ constant ] ),
    HR -> 3600x(1000000000000000xN [ constant ] ),
    OTHERWISE
    -> error
    (cat("Illegal unit name for physical type TIME: "))
  )
(time-type-desc(),(TIME ,normalized-constant,FS ))

EX [ CHAR constant ] (p)(t)
= let d = lookup(t)((STANDARD ) )(expr) in
  (type(d), (CHAR ,constant))

EX [ BITSTR bit-lit* ] (p)(t) = (e,(BITSTR ,bit-lit*))

EX [ STR char-lit* ] (p)(t) = (e,(STR ,char-lit*))

EX [ REF name ] (p)(t) = transform-name(name)(e)(e)(p)(t)

transform-name(name)(w)(ast?)(p)(t)
= (null(w))
  -> let w1 = lookup2(t)(p)(e)(hd(name)) in
  transform-name
  (tl(name))(w1)(((SREF ,path(tdesc(w1)),idf(tdesc(w1)))))(p)(t),
  let d = tdesc(w) in
  let tg = tag(d) in
  let x = hd(name) in
  and tm = tmode(w) in
  (consp(x))
  -> let ast? = transform-list(x)(p)(t) in
  (second(tm)= OBJ ^ is-array-tdesc?(d)
    - transform-name
      (tl(name))((tm,elty(d)))
      (nconc(ast?,((INDEX ,hd(ast?)))))(p)(t),
      (second(tm)= OBJ ^ is-array?((type(d))))
    V (second(tm)= (REF VAL) ^ is-array-tdesc?(d))
    - transform-name
      (tl(name))
      ((second(tm)= OBJ
        -> mk-type(tmode(type(d)))(elty(tdesc(type(d)))),
        mk-type(tm)(elty(d))))
      (nconc(ast?,((INDEX ,hd(ast?)))))(p)(t),
      transform-name
      (tl(name))(extract-rtype(d))
      (nconc(ast?,((PARLIST ,ast?))))(p)(t),
    ((second(tm)= OBJ ^ is-record?(type(d))))
    V (second(tm)= (REF VAL) ^ is-record-tdesc?(d))
    -> let d1 = (second(tm)= OBJ -> tdesc(type(d)), d) in
    let d2 = lookup-record-field(components(d1))(x) in

94
transform-name
   (tl(name))(mk-type(tm)(d_2))(aconc(ast\_x,((SELECTOR ,x))))
   (p)(t),
second(tm)= OBJ ∧ is-record-tdesc?(d)
→ let d_2 = lookup-record-field(components(d))(x) in
   transform-name
   (tl(name))(mk-type(tm)(d_2))(aconc(ast\_x,((SELECTOR ,x))))
   (p)(t),
let w_1 = lookup-local(t)(%(path(d))(idf(d)))(x) in
   transform-name
   (tl(name))(w_1)(((SREF ,path(tdesc(w_1)),idf(tdesc(w_1)))))(p)
   (t))))

transform-name-aux(tg)(d)(ast)
= (case tgi
    "OBJECT" → (second(type(d)),(REF ,ast)),
    "ENUMELT" → (second(type(d)),(ENUMLIT ,idf(d))),
    "PROCEDURE" * "FUNCTION"
      → (second(type(d)),(REF ,aconc(ast,((PARLIST ,e))))),
    OTHERWISE → (d,(REF ,ast)))

transform-list(x)(p)(t)
= (null(x) → ε,
   let expr = hd(x) in
   cons(second(EX [ expr ] (p)(t)),transform-list(tl(x))(p)(t)))

The functions transform-name, transform-name-aux, and transform-list produce the
linear form of the basic references discussed above.

(EX10) EX [ PAGGR expr* ] (p)(t)
= (length(expr*)= 1
   → let expr = hd(expr*) in
   EX [ expr ] (p)(t),
   (ε,(PAGGR ,ex-paggr(expr*)(p)(t))))

(EX11) EX [ unary-op expr ] (p)(t)
= let (d,e) = EX [ expr ] (p)(t) in
   (case unary-op
      PLUS → (d,e),
      NOT → (d,(scalar-op(unary-op)(d),e)),
      NEG → (d,(scalar-op(unary-op)(d),e)),
      ABS → (d,(scalar-op(unary-op)(d),e)),
      OTHERWISE → error
        → error
          (cat("Unrecognized Stage 2 VHDL unary operator: "))(unary-op)))

(EX12) EX [ binary-op expri expr2 ] (p)(t)
= let (d_1,e_1) = EX [ expr1 ] (p)(t) in
   let (d_2,e_2) = EX [ expr2 ] (p)(t) in
   (d_1,(scalar-op(binary-op)(d_1),e_1,e_2))

(EX13) EX [ relational-op expri expr2 ] (p)(t)
= let (d_1,e_1) = EX [ expr1 ] (p)(t) in
   let (d_2,e_2) = EX [ expr2 ] (p)(t) in
   (bool-type-desc(),(scalar-op(relational-op)(d_1),e_1,e_2))
scalar-op(op)(d)
= (is-bit-tdesc?(d)V is-vector-tdesc?(d)→ bits-op(op),
is-real-tdesc?(d)→ real-op(op),
op)

bits-op(op)
= (case op
   EQ → EQ ,
   NE → NE ,
   LT → LT ,
   LE → LE ,
   GT → GT ,
   GE → GE ,
   NOT → BNOT ,
   AND → BAND ,
   NAND → BNAND ,
   OR → BOR ,
   NOR → BNOR ,
   XOR → BXOR ,
   OTHERWISE → error(cat(Undefined bitwise operator: )(op)))

real-op(op)
= (case op
   EQ → EQ ,
   NE → NE ,
   LT → RLT ,
   LE → RLE ,
   GT → RGT ,
   GE → RGE ,
   NEG → RNEG ,
   ABS → RABS ,
   ADD → RPLUS ,
   SUB → RMINUS ,
   MUL → RTIMES ,
   DIV → RDIV ,
   EXP → REXPT ,
   OTHERWISE → error(cat(Undefined 'real' operator: )(op)))

The functions scalar-op, bits-op, and real-op do overload resolution between INTEGER, BIT, and REAL operators.

(RX1) RX [ expr ] (p)(t) = EX [ expr ] (p)(t)
8 Phase 2: State Delta Generation

If Phase 1 of the Stage 2 VHDL translator completes without error, then after the interphase abstract syntax tree transformation has been accomplished (see Section 7), Phase 2, state delta generation, can proceed. Several kinds of checks have already been performed on the hardware description in Phase 1, the most significant being the detection of missing prior declarations of items such as variables and labels, the improper use of names, and static type checking. Thus, these checks do not have to be duplicated in Phase 2.

Phase 2 receives from Phase 1 the transformed abstract syntax tree (AST) for the hardware description, together with the tree-structured environment (TSE) — a complete record of the name/attribute associations corresponding to the hardware description's declarations and whose structure reflects that of the description. The TSE remains fixed throughout Phase 2. It contains all definitions needed to execute its corresponding Stage 2 VHDL hardware description, and Phase 1 has ensured that only that portion of the TSE visible at any given textual point of the description can be accessed during Phase 2. With the aid of the TSE, Phase 2 incrementally generates SDVS Simplifier assertions and state deltas (SDs).

8.1 Phase 2 SemanticDomains and Functions

The formal description of Phase 2 translation consists of semantic domains and semantic functions, the latter being functions from syntactic to semantic domains. Compound semantic domains are defined in terms of primitive semantic domains. Similarly, primitive semantic functions are unspecified (their definitions being understood implicitly) and the remaining semantic functions are defined (by syntactic cases) via semantic equations.

The principal Phase 2 semantic functions (and corresponding Stage 2 VHDL language constructs to which they assign meanings) are: DF (design files), EN (entity declarations), AR (architecture bodies), D (declarations), CS (concurrent statements), SS (sequential statements), W (waveforms), TRM and TR (transactions), ME and MR (expression lists), E and R (expressions), T (expression types), B (bit literals), and N (numeric literals).

Each of the principal semantic functions requires an appropriate syntactic argument — an abstract syntactic object (tree) produced by the interphase abstract syntax tree transformation (see Section 7). Most of the semantic functions take (at least) the following additional arguments:

- the tree-structured environment (TSE) generated in Phase 1;
- a path, indicating the currently “visible” portion of the TSE;
- a continuation, specifying which Phase 2 semantic function to invoke next;
- a universe structure; and
- an execution stack.
In the absence of errors, the Phase 2 semantic functions return a list of Simplifier assertions and SDs. Moreover, E and R also return a translated expression and list of guard formulas. Guard formulas are inserted in the precondition of generated SDs to ensure that certain conditions are met in the proof in which the SDs appear. For example, if an array name is indexed by an expression, then Phase 2 generates a guard formula asserting that the index value is not out of range.

The execution state manipulated by Phase 2 translation involves two components: a universe structure (see Section 8.2.2) and an execution stack (see Section 8.2.3). An analogy with conventional denotational semantics can be applied: the execution state corresponds to the store, translated expressions and guard formulas correspond to expression values, and SD/assertion lists correspond to non-error final answers.

When SDs are generated by a semantic function, the continuation that is input to that function plays a slightly unconventional role: the result of applying to an execution state the continuation, or other continuations derived from the continuation, is appended to the postconditions of the generated SDs. In the absence of errors, the item appended represents a list of SDs. Such a continuation is evaluated and applied only when the SD in whose postcondition it appears is applied.

For example, an IF statement having no ELSE part generates two state deltas: one for the case in which its condition evaluates to true, the other for the false case. The continuation for the true case represents the execution of the body of the IF statement succeeded by the execution of the statement following the IF statement. The continuation for the false case skips the body, and proceeds directly to the statement following the IF statement. Whichever of these two SDs is applied determines which continuation is evaluated and applied to an execution state, and therefore which additional state deltas are subsequently generated.

Figures 3 and 4 depict, respectively, the semantic domains and function types for Phase 2 of the Stage 2 VHDL translator.
Primitive Semantic Domains

*Bool* = \{FALSE, TRUE\}  
Simplifier propositional (boolean) constants

*Bit* = \{(BS 0 1), (BS 1 1)\}  
Simplifier bit constants (length 1 bitstrings)

*Char* = \{(CHAR 0), ..., (CHAR 127)\}  
Simplifier character constants

n : N = \{0, 1, 2, ...\}  
Simplifier natural number constants

id : Id  
identifiers

SysId  
system-generated identifiers (disjoint from Id)

t : TEnv  
tree-structured environments (TSEs)

d : Desc  
descriptors (see Section 6.2)

v : UStruct  
universe structures (see Section 8.2.2)

stk : Stk  
execution stacks (see Section 8.2.3)

e : TExpr  
translated expressions

trans : TTrans  
translated transactions

f, guard : GForm  
lists of guard formulas

sd : SD  
state deltas

Assert  
SDVS Simplifier assertions

Error  
error messages

Compound Semantic Domains

elbl : Elbl = Id + SysId  
TSE edge labels

p, q: Path = Elbl*  
TSE paths

qname: Name = Elbl (. Elbl)*  
qualified names

d : Dv = Desc  
denotable values (descriptors)

r : Env = Id → (Dv + {*UNBOUND*})  
environments

Tmode = \{PATH\} × Id* +  
(type modes)

\({\{\text{CONST, VAR, SIG, DUMMY}\} ×} \{\text{VAL, OUT, REF, OBJ, ACC, TYP}\}\)

w : Type = Tmode × Desc  
types

u : Dc = UStruct → Stk → Ans  
declaration & concurrent statement continuations

c : Sc = Dc  
sequential statement continuations

k : Ec = (TExpr × GForm) → Sc  
expression continuations

h : Mc = (TExpr* × GForm*) → Sc  
expression list continuations

wave-cont : Wc = (TTrans* × GForm*) → Sc  
waveform continuations

trans-cont : Tc = (TTrans × GForm) → Sc  
transaction continuations

Ans = (SD + Assert)* + Error  
final answers

Figure 3: Phase 2 Semantic Domains
<table>
<thead>
<tr>
<th></th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF</td>
<td>Design $\rightarrow$ TEnv $\rightarrow$ Ans</td>
</tr>
<tr>
<td>EN</td>
<td>Ent $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Dc $\rightarrow$ Dc</td>
</tr>
<tr>
<td>AR</td>
<td>Arch $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Dc $\rightarrow$ Dc</td>
</tr>
<tr>
<td>D</td>
<td>Dec$^*$ $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Dc $\rightarrow$ Dc</td>
</tr>
<tr>
<td>CS</td>
<td>CStat$^*$ $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Dc $\rightarrow$ Dc</td>
</tr>
<tr>
<td>SS</td>
<td>SStat$^*$ $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Sc $\rightarrow$ Sc</td>
</tr>
<tr>
<td>W</td>
<td>Wave $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Wc $\rightarrow$ Sc</td>
</tr>
<tr>
<td>TRM</td>
<td>Trans$^*$ $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Wc $\rightarrow$ Sc</td>
</tr>
<tr>
<td>TR</td>
<td>Trans $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Tc $\rightarrow$ Sc</td>
</tr>
<tr>
<td>ME</td>
<td>Expr$^*$ $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Mc $\rightarrow$ Sc</td>
</tr>
<tr>
<td>MR</td>
<td>Expr$^*$ $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Mc $\rightarrow$ Sc</td>
</tr>
<tr>
<td>E</td>
<td>Expr $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Ec $\rightarrow$ Sc</td>
</tr>
<tr>
<td>R</td>
<td>Expr $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Ec $\rightarrow$ Sc</td>
</tr>
<tr>
<td>T</td>
<td>Expr $\rightarrow$ TEnv $\rightarrow$ Path $\rightarrow$ Desc</td>
</tr>
<tr>
<td>B</td>
<td>BitLit $\rightarrow$ Bit</td>
</tr>
<tr>
<td>N</td>
<td>NumLit $\rightarrow$ N</td>
</tr>
</tbody>
</table>

**Figure 4: Phase 2 Semantic Functions**

- Design file dynamic semantics
- Entity declaration dynamic semantics
- Architecture body dynamic semantics
- Declaration dynamic semantics
- Concurrent statement dynamic semantics
- Sequential statement dynamic semantics
- Waveform dynamic semantics
- Transaction list dynamic semantics
- Transaction dynamic semantics
- Expression list dynamic semantics ($l$-values)
- Expression list dynamic semantics ($r$-values)
- Expression dynamic semantics ($l$-values)
- Expression dynamic semantics ($r$-values)
- Expression types
- Bit values of bit literals (primitive)
- Integer values of numeric literals (primitive)
8.2 Phase 2 Execution State

As mentioned in Section 8.1, the execution state manipulated by Phase 2 translation consists of a universe structure and an execution stack. The purpose of this section is to elucidate the nature and role of these aspects of the execution state.

8.2.1 Unique Name Qualification

Except for quantification, the language of state deltas has no scoping, i.e., it is "flat." Even with quantification, the state deltas generated by the Stage 2 VHDL translator certainly do not have a scoping structure that naturally parallels the scopes of their corresponding Stage 2 VHDL hardware description. Furthermore, even if there were such a correspondence between source (Stage 2 VHDL) and target (state deltas) scopes, it would still be convenient to generate unique names for the SDVS user to use in proofs.

For example, a PROCESS statement may contain a declaration of a variable $x$ of the same name as a signal in the enclosing architecture body. The inner instance of $x$ can be distinguished from the outer instance by prefixing or qualifying it with the name (user-supplied or system-generated) of the process in which the inner instance is declared. We shall call such a qualified name, derived from the static structure of the Stage 2 VHDL hardware description, a *statically uniquely qualified name* or SUQN. At the beginning of Phase 2 translation (after the interphase AST transformation — see Section 7), the SUQN of any object (for which such a name makes sense) is recorded in the qid field associated with the object in the TSE.

Another important kind of unique name qualification is based on the dynamic execution of a Stage 2 VHDL description. A program unit can be reentered, either by repetition or recursion, and local declarations in the reentered program will be re-elaborated, creating new dynamic instances of entities that cannot be distinguished on the basis of static program structure. In this case new names that are distinct dynamic instances of the same statically uniquely qualified name are sufficient to enable the SDVS user to distinguish all instances of names for use in proofs. The separate dynamic instances of a name are indicated by appending $!n$ to it, where $n$ is a *dynamic instance index* for that name (e.g., $a.x$, $a.x!2$, $a.x!3$, ..., where $a.x!1$ is simply denoted $a.x$). These names are called *dynamically uniquely qualified names* (DUQNs).

Only statically and dynamically uniquely qualified names appear in the state deltas generated by Phase 2 translation.

8.2.2 Universe Structure for Unique Dynamic Naming

Given that there may be several dynamic instances of the same SUQN in a Stage 2 VHDL hardware description, Phase 2 translation employs a mechanism called a *universe structure* (together with functions that access and manipulate it) to manage the creation of new dynamic instances of each distinct SUQN, as well as to ensure that the correct dynamic instance of each SUQN is available at any given time.
A universe structure consists of four components:

**universe name**:
The name of the current universe. A universe name has the form $z\backslash u\backslash n$, where $z$ is the name of the main program and $n$ is the current universe’s ordinal number ($n = 1, 2, \ldots$).

**universe counter**:
The current universe’s ordinal number.

**universe stack**:
A stack of universe names used to save and restore prior universes in accordance with the changes of environment in a Stage 2 VHDL hardware description.

**universe variables**:
The current universe’s environment of statically and dynamically uniquely qualified names. This is a list of entries of the form $(SUQN, ordinal-number, ordinal-stack)$, one for each distinct SUQN. The ordinal number denotes the most recently created dynamic instance of that SUQN. The ordinal stack is a stack of this SUQN’s ordinal numbers, whose top element denotes the current dynamic instance of this SUQN. This stack is used to save and restore prior dynamic instances of this SUQN in accordance with the changes of environment in a Stage 2 VHDL hardware description.

```plaintext
mk-initial-universe(z) = let uname = catenate(z, "\u", 1) in make-universe-data(uname, 1, (uname), ((z, l, (l))))

make-universe-data(uname, ucounter, ustack, uvars) = (uname, ucounter, ustack, uvars)

universe-name(v) = hd(v)

universe-counter(v) = second(v)

universe-stack(v) = third(v)

universe-vars(v) = fourth(v)

push-universe(v, z, suqn*) = let ucounter = 1 + universe-counter(v) in let uname = catenate(z, "\u", ucounter) in let ustack = cons(uname, universe-stack(v)) in make-universe-data
(uname, ucounter, ustack, push-universe-vars(suqn*, universe-vars(v)))

push-universe-vars(suqn*, vars) = (null(suqn*) → vars, let suqn = hd(suqn*) in let v = assoc(suqn, vars) in (null(v) → push-universe-vars(tl(suqn*), cons(init-var(suqn), vars)), push-universe-vars(tl(suqn*), cons(push-var(v), vars))))
```

102
Currently, the only part of the universe structure that is actually used for dynamic name qualification is the universe variables component. Each time a program unit that may have a declarative part (packages, entities, architectures, processes, subprogram bodies) is entered, the current universe is saved and an updated universe structure is created by push-universe. The universe structure's counter (ordinal) is incremented by one, a corresponding new universe name is created, and the old universe name is pushed onto the universe stack. In the universe variables component of the universe structure, the triple for each SUQN corresponding to each name declared in the unit's declarative part (except types) is updated: the value of its ordinal is incremented by one and this new ordinal value is pushed onto the ordinal stack of the SUQN's triple. Whenever any SUQN needs to be dynamically uniquely
qualified, the top element of its ordinal stack is used to find the index of the current dynamic instance of that SUQN.

When such a program unit is exited, pop-universe restores the universe name by popping it from the universe stack. The ordinal stack of the triple of the SUQN of each (non-type) name declared in this unit is popped, restoring the current dynamic qualification of that SUQN to a former value.

The functions get-qualified-ids, qualified-id, and name-qualified-id accomplish the dynamic qualification of SUQNs relative to a universe structure.

8.2.3 Execution Stack

The elements of the execution stack are descriptors that contain information to control normal returns and exits from program units, as well as the undeclaration of objects, packages, subprograms, and formal parameters.

There are several kinds of execution stack descriptors, and more detailed explanations of their roles will be provided at the points in the semantics where they are used. For now, we note that each descriptor has four components: an identifying tag; an identifier, identifier sequence, or fully qualified name that associates the descriptor with some program unit; a path that may replace the current path to effect a change of environment; and a function, which may be a continuation or continuation transformer, that will effect a change of control and environment corresponding to the descriptor's purpose.

stack bottom:

< *STKBOTTOM*, id, ε, ε >

This descriptor is the execution stack “bottom marker,” used to terminate model execution and to prevent execution stack underflow. The identifier id is the name of the Stage 2 VHDL design file.

package body exit:

< *PACKAGE-BODY-EXIT*, id, p, u >

This descriptor is pushed onto the execution stack just prior to the elaboration of a package body. The identifier id is the package name, and u: Dc is a declaration continuation that will continue execution (most likely elaboration) at the package body's successor in the environment denoted by p.

subprogram return:

< *SUBPROGRAM-RETURN*, id, p, c >

This descriptor is pushed onto the execution stack after a subprogram (procedure or function) is entered, but just before the elaboration of the subprogram's local declarations. The identifier id is the subprogram name, and c: Sc is a continuation that will continue execution at the successor of the subprogram call in the environment denoted by p.
loop exit :

\[ \langle *\text{LOOP-EXIT}*, \text{id}, \text{p}, \text{c} > \]

This descriptor is pushed onto the execution stack when a loop statement (\text{LOOP}, \text{WHILE}, or \text{FOR}) is entered. The identifier \text{id} is the loop label, and \text{c} : \text{Sc} is a continuation that will continue execution at the loop's successor in the environment denoted by \text{p}.

block exit :

\[ \langle *\text{BLOCK-EXIT}*, \text{id}, \text{p}, \text{c} > \]

This descriptor is pushed onto the execution stack just before the elaboration of a \text{FOR} loop's iteration parameter, which implicitly establishes a block scope. The identifier \text{id} is the \text{FOR} loop label, and \text{c} : \text{Sc} is a continuation that will continue execution at the \text{FOR} loop's successor statement in the environment denoted by \text{p}.

begin marker :

\[ \langle *\text{BEGIN}*, \text{id}, \text{p}, \text{c} > \]

This descriptor is pushed onto the execution stack immediately after the local declarations of a subprogram, or the iteration parameter of a \text{FOR} loop, have been elaborated.

undeclaration :

\[ \langle *\text{UNDECLARE}*, \text{id}^+, \text{p}, \text{g} > \]

This descriptor, pushed onto the execution stack when a subprogram is called, enables the eventual explicit undeclaration (upon subprogram exit) of the subprogram's formal parameters and other locally declared objects. The identifier list \text{id}^+ names the objects to be undeclared, and \text{g} : \text{Sc} \rightarrow \text{Sc} is a continuation transformer which, after carrying out the explicit undeclaration specified in \text{g} (thereby popping this *UNDECLARE* descriptor from the execution stack), continues execution via its continuation argument.
8.3 Special Functions

Certain functions appearing in the semantic specification of Phase 2 translation are not defined denotationally, for either of two reasons: (1) their denotational description is too cumbersome or not well understood, or (2) they are used to construct SDVS-dependent representations of expressions or formulas.

These functions, implemented directly in Common Lisp, are described below.

8.3.1 Operational Semantic Functions

To understand Phase 2 translation, it is important to recognize that in defining the semantics of the VHDL simulation cycle, the VHDL translator involves a significant operational component. This is to be distinguished from the semantics of sequential statements within processes, which the translator defines in a primarily denotational manner.

We are referring here to our strategy, explained in Section 2, of designing aspects of a simulator kernel into the Stage 2 VHDL translator. After application of the state deltas specifying the behavior of one execution cycle for the active processes, the translator is responsible for:

- determining the next VHDL clock time at which a driver becomes active or a process resumes;
- advancing the SDVS state to this new time; and
- generating the state delta that specifies the next sequential statement in the first resuming process for the new execution cycle.

After a given resuming process suspends, its continuation is the textually next resuming process.

It is the internal translator machinery to perform these tasks that is operationally defined — much of it embodied in a portion of the translator that is directly coded in Common Lisp, rather than described by semantic equations. The names of the Common Lisp functions serving this purpose are listed below.

make-vhdl-process-elaborate
make-vhdl-begin-model-execution
make-vhdl-try-resume-next-process
make-vhdl-process-suspend
find-signal-structure
name-driver
init-scalar-signal
8.3.2 Constructing State Deltas

The construction of SDs is specified via functions \texttt{mk-sd(z)(pre, comod, mod, post)} and \texttt{mk-sd-decl(z)(pre, comod, mod, post)}, which take five arguments: the design file name \texttt{z} (if \texttt{p} is the current path, this is always \texttt{hd(p)}) and representations of the precondition, comodification list, modification list, and postcondition of the SD to be constructed.

These functions are used to represent the construction of SDs without specifying their exact representation, which is SDVS-dependent and not given here. The pre- and postconditions of an SD are \textit{lists} of formulas, each of which represents a formula that is the logical \textit{conjunction} of the formulas in this list. If the precondition and comod list arguments of \texttt{mk-sd} and \texttt{mk-sd-decl} are \texttt{e}, then the precondition and comod list of the constructed SD are \texttt{(TRUE)} and \texttt{(ALL)}, respectively. Otherwise, the given arguments are used directly in the SD. The postcondition may contain an SD, which is usually represented as a statement continuation applied to an execution stack.

\texttt{mk-sd} and \texttt{mk-sd-decl} are almost the same, the only difference being that an SD created by \texttt{mk-sd-decl} is given a special tag that identifies its association with declaration elaboration rather than statement execution.

For technical reasons, the comod list of every SD is \texttt{(ALL)} and the mod list of every SD must be \textit{nonempty}. To ensure that an SD's mod list is never empty, \texttt{mk-sd(z)( ... )} always prefix \texttt{z\textbackslash pc} to its mod list argument, where \texttt{z\textbackslash pc} is a unique place (represented by a system identifier) in which \texttt{z} is the name of the Stage 2 VHDL hardware description being translated. This unique place is the name of a \textit{program counter} whose value implicitly changes when \textit{any} SD is applied. This program counter place does not make any other kind of appearance in a translated Stage 2 VHDL hardware description.

The notation of state deltas requires that certain symbols sometimes be prefixed to uniquely qualified names: the \texttt{dot (.)} and \texttt{pound (#)} symbols. The functions \texttt{dot} and \texttt{pound}, applied to uniquely qualified names, accomplish this.

\texttt{dot(placename) = (DOT ,placename)}

\texttt{pound(placename) = (POUND ,placename)}

Finally, the two functions \texttt{fixed-characterized-sds} and \texttt{subst-vars} are employed by the Phase 2 semantics of procedure calls to implement the SDVS \textit{affine characterization} mechanism [16, 17], which will be incorporated in Stage 3 VHDL.
8.3.3 Error Reporting

The few kinds of errors that can occur in Phase 2 are reported by the functions `impl-error` and `execution-error`.

The function `impl-error` is used, for example, to report invalid arguments passed to the low-level utility functions `mk-scalar-rel`, `mk-exp1`, and `mk-exp2`, although this should never occur.

The function `execution-error` is used to report execution errors such as an empty execution stack, although again, such errors should never occur if Phase 1 has done its job.
8.4 Phase 2 Semantic Equations

This section constitutes the heart of the present report. It documents the semantic equations and auxiliary semantic functions in terms of which Phase 2 of the Stage 2 VHDL translator — state delta generation — is denotationally specified.

8.4.1 Stage 2 VHDL Design Files

(DF1) \(DF \llbracket \text{DESIGN-FILE id pkg-decl* pkg-body* use-clause* ent-decl arch-body} \rrbracket (t)\)

\[\begin{align*}
&\text{let } p_0 = \%e(id) \text{ in} \\
&\text{let } id_1 = \text{hd}(\text{id(ent-decl)}) \text{ in} \\
&\text{let } p_1 = \%p_0(id_1) \text{ in} \\
&\text{let } v = \text{mk-initial-universe(id)} \\
&\text{and } \text{stk} = (\langle \text{STKBOTTOM* ,id,e,e} \rangle) \text{ in} \\
&\text{(mk-disjoint(id,(dot(id)))),} \\
&\text{mk-cover} \\
&\quad (\dot{id}) \text{(concat(id,"pc"),VHDLTIME ,VHDLTIME\text{-}\text{PREVIOUS })),} \\
&\text{mk-scalar-decl(VHDLTIME ,(TYPE VHDLTIME )},} \\
&\text{mk-scalar-decl(VHDLTIME\text{-}\text{PREVIOUS ,(TYPE VHDLTIME )},} \\
&\text{mk-rel(vhdltime-type-desc())((EQ ,dot(VHDLTIME ),mk-vhdltime(0)(0))),} \\
&\text{mk-rel} \\
&\quad (vhdltime-type-desc())((EQ ,dot(VHDLTIME\text{-}\text{PREVIOUS }),mk-vhdltime(0)(0))),} \\
&\text{mk-decl-sd(id)(e}(e)(e)(u_1(v)(stk))) \\
&\text{where } u_1 = \lambda v,stk.D \llbracket \text{pkg-decl* } (t)(p_0)(u_2)(v)(stk) \\
&\text{where } u_2 = \lambda v,stk.D \llbracket \text{pkg-body* } (t)(p_0)(u_3)(v)(stk) \\
&\text{where } u_3 = \lambda v,stk.D \llbracket \text{use-clause* } (t)(p_0)(u_4)(v)(stk) \\
&\text{where } u_4 = \lambda v,stk.\text{EN} \llbracket \text{ent-decl } (t)(p_0)(u_5)(v)(stk) \\
&\text{where } u_5 = \lambda v,stk.\text{AR} \llbracket \text{arch-body } (t)(p_1)(u_6)(v)(stk) \\
&\text{where } u_6 = \lambda v,stk.\text{block-exit}(v)(stk)
\end{align*}\]

\text{mk-disjoint(id,lst) = cons(ALLDISJOINT ,cons(id,lst))}

\text{mk-cover(id,lst) = cons(COVERING ,cons(id,lst))}

\text{mk-scalar-decl(placename,place-type) = (DECLARE ,placename,place-type)}

\text{vhdltime-type-desc()} = \langle \text{VHDLTIME },e,*\text{VHDLTIME* ,(STANDARD) },tt\rangle

\text{mk-rel(d)(op,e_1,e_2)}

\[\begin{align*}
&\text{let } t_\text{g} = \text{tag(d)} \text{ in} \\
&\text{(case } t_\text{g} \\
&\quad (\text{*/BOOL* ,*BIT* ,*INT* ,*REAL* ,*TIME* ,*VHDLTIME* ,*ENUMTYPE* ,*POLY* }) \\
&\quad \rightarrow \text{mk-scalar-rel(tg)((op,e_1,e_2))}, \\
&\quad \text{*WAVE* } \rightarrow (\text{EQ ,e_1(e_2)}, \\
&\quad \text{*ARRAYTYPE* } \\
&\quad \rightarrow (\text{is-bitvector-tdesc?}(d)) \\
&\quad \rightarrow (\text{case op} \\
&\quad \text{EQ } \\
&\quad \rightarrow (\text{is-constant-bitvector?}(e_1) \land \text{is-constant-bitvector?}(e_2)) \\
&\quad \rightarrow (\text{EQ ,cons(USSCONC ,e_1),cons(USSCONC ,e_2))}, \\
&\quad \text{is-constant-bitvector?}(e_2) \rightarrow (\text{EQ ,e_1,cons(USSCONC ,e_2))}, \\
&\quad \text{is-constant-bitvector?}(e_1) \rightarrow (\text{EQ ,cons(USSCONC ,e_1),e_2}), \\
&\quad (\text{EQ ,e_1,e_2)}), \\
&\quad \text{NE } \\
&\quad \rightarrow (\text{is-constant-bitvector?}(e_1) \land \text{is-constant-bitvector?}(e_2))
\end{align*}\]
\[\begin{align*}
&\rightarrow (\text{NEQ}, \text{cons}([\text{USCONC}], e_1), \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_2) \rightarrow (\text{NEQ}, e_1, \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_1) \rightarrow (\text{NEQ}, \text{cons}([\text{USCONC}], e_1), e_2), \\
&(\text{NEQ}, e_1, e_2)), \\
&\text{LT} \quad \rightarrow (\text{EQ}, (\text{BS}, 1, 1), \\
&\quad \text{is-constant-bitvector}(e_1) \land \text{is-constant-bitvector}(e_2) \\
&\quad \rightarrow (\text{USLSS}, \text{cons}([\text{USCONC}], e_1), \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_2) \rightarrow (\text{USLSS}, e_1, \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_1) \rightarrow (\text{USLSS}, \text{cons}([\text{USCONC}], e_1), e_2)), \\
&(\text{USLSS}, e_1, e_2)), \\
&\text{LE} \quad \rightarrow (\text{EQ}, (\text{BS}, 1, 1), \\
&\quad \text{is-constant-bitvector}(e_1) \land \text{is-constant-bitvector}(e_2) \\
&\quad \rightarrow (\text{USLEQ}, \text{cons}([\text{USCONC}], e_1), \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_2) \rightarrow (\text{USLEQ}, e_1, \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_1) \rightarrow (\text{USLEQ}, \text{cons}([\text{USCONC}], e_1), e_2)), \\
&(\text{USLEQ}, e_1, e_2)), \\
&\text{GT} \quad \rightarrow (\text{EQ}, (\text{BS}, 1, 1), \\
&\quad \text{is-constant-bitvector}(e_1) \land \text{is-constant-bitvector}(e_2) \\
&\quad \rightarrow (\text{USGTR}, \text{cons}([\text{USCONC}], e_1), \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_2) \rightarrow (\text{USGTR}, e_1, \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_1) \rightarrow (\text{USGTR}, \text{cons}([\text{USCONC}], e_1), e_2)), \\
&(\text{USGTR}, e_1, e_2)), \\
&\text{GE} \quad \rightarrow (\text{EQ}, (\text{BS}, 1, 1), \\
&\quad \text{is-constant-bitvector}(e_1) \land \text{is-constant-bitvector}(e_2) \\
&\quad \rightarrow (\text{USGEQ}, \text{cons}([\text{USCONC}], e_1), \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_2) \rightarrow (\text{USGEQ}, e_1, \text{cons}([\text{USCONC}], e_2)), \\
&\text{is-constant-bitvector}(e_1) \rightarrow (\text{USGEQ}, \text{cons}([\text{USCONC}], e_1), e_2)), \\
&(\text{USGEQ}, e_1, e_2)), \\
&\text{OTHERWISE} \quad \rightarrow \text{impl-error}("Shouldn't happen!"), \\
&\text{is-string-tdesc}(d) \\
&\rightarrow (\text{case op EQ}) \\
&\rightarrow (\text{is-constant-string}(e_1) \land \text{is-constant-string}(e_2) \\
&\quad \rightarrow (\text{EQ}, \text{cons}([\text{ACONC}], e_1), \text{cons}([\text{ACONC}], e_2)), \\
&\text{is-constant-string}(e_2) \rightarrow (\text{EQ}, e_1, \text{cons}([\text{ACONC}], e_2)), \\
&\text{is-constant-string}(e_1) \rightarrow (\text{EQ}, \text{cons}([\text{ACONC}], e_1), e_2)), \\
&(\text{EQ}, e_1, e_2)), \\
&\text{NE} \quad \rightarrow (\text{is-constant-string}(e_1) \land \text{is-constant-string}(e_2) \\
&\quad \rightarrow (\text{NEQ}, \text{cons}([\text{ACONC}], e_1), \text{cons}([\text{ACONC}], e_2)), \\
&\text{is-constant-string}(e_2) \rightarrow (\text{NEQ}, e_1, \text{cons}([\text{ACONC}], e_2)), \\
&\text{is-constant-string}(e_1) \rightarrow (\text{NEQ}, \text{cons}([\text{ACONC}], e_1), e_2)), \\
&(\text{NEQ}, e_1, e_2)), \\
&\text{OTHERWISE} \quad \rightarrow \text{impl-error}("Shouldn't happen!"), \\
&\text{(case op EQ}) \\
&\rightarrow (\text{not-dotted-expr-p}(e_2) \rightarrow \text{impl-error}("Shouldn't happen!"), \\
&(\text{EQ}, e_1, e_2)), \\
&\text{NE} \quad \rightarrow (\text{not-dotted-expr-p}(e_2) \rightarrow \text{impl-error}("Shouldn't happen!"), \\
&(\text{NEQ}, e_1, e_2)), \\
&\text{OTHERWISE} \quad \rightarrow \text{impl-error}("Shouldn't happen!")), \\
\end{align*}\]
→ (not-dotted-expr-p(e2) → impl-error("Shouldn’t happen!")),
(EQ ,e1,e2)),
OTHERWISE → impl-error("Shouldn’t happen!"))
is-constant-bitvector?(expr*)
= null(expr*)
 ∨ (consp(expr*))
 ∧ let expr1 = hd(expr*) in
 consp(expr1) ∧ hd(expr1) = BS )
is-constant-string?(expr*)
= null(expr*)
 ∨ (consp(expr*))
 ∧ let expr1 = hd(expr*) in
 consp(expr1) ∧ hd(expr1) = CHAR )
not-dotted-expr-p(expr) = ¬(consp(expr) ∧ hd(expr) = DOT )

mk-scalar-rel(type-tag)(relational-op,e1,e2)
= (case type-tag
 *BOOL*
 → (case relational-op
 EQ → mk-bool-eq(type-tag,e1,e2),
 NE → mk-bool-neq(type-tag,e1,e2),
 LT → (AND ,(EQ ,e1,FALSE ),(EQ ,e2,TRUE )),
 LE → (IMPLIES ,e1,e2),
 GT → (AND ((USLSS ,e1,e2),(BS ,1,1)),
 LE → (EQ ,USLEQ ,e1,e2),(BS ,1,1)),
 GE → (EQ ,USGTR ,e1,e2),(BS ,1,1)),
 OTHERWISE
 → impl-error
 ("Unrecognized Stage 2 VHDL ‘boolean’ relational operator: “a”,
 relational-op)),

 *BIT*
 → (case relational-op
 EQ → (EQ ,e1,e2),
 NE → (NEQ ,e1,e2),
 LT → (EQ ,(USLSS ,e1,e2),(BS ,1,1)),
 LE → (EQ ,USLEQ ,e1,e2),(BS ,1,1)),
 GT → (EQ ,(USGTR ,e1,e2),(BS ,1,1)),
 GE → (EQ ,USGEQ ,e1,e2),(BS ,1,1)),
 OTHERWISE
 → impl-error
 ("Unrecognized Stage 2 VHDL ‘bit’ relational operator: “a”,
 relational-op)),

 *INT*
 → (case relational-op
 EQ → (EQ ,e1,e2),
 NE → (NEQ ,e1,e2),
 LT → (LT ,e1,e2),
 LE → (LE ,e1,e2),
 GT → (GT ,e1,e2),
 GE → (GE ,e1,e2),
 OTHERWISE
 → impl-error
 ("Unrecognized Stage 2 VHDL ‘integer’ relational operator: “a”,
 relational-op)),

 *REAL*
→ (case relational-op
  EQ → (EQ ,e1,e2),
  NE → (NEQ ,e1,e2),
  (RLT ,RLE,RGT ,RGE ) → (relational-op,e1,e2),
  OTHERWISE → impl-error
  ("Unrecognized Stage 2 VHDL ‘real’ relational operator: "a", relational-op)),

*ENUMTYPE*
→ (case relational-op
  EQ → (EQ ,e1,e2),
  NE → (NEQ ,e1,e2),
  LT → (ELT ,e1,e2),
  LE → (ELE ,e1,e2),
  GT → (EGT ,e1,e2),
  GE → (EGE ,e1,e2),
  PRED → (EPRED ,e1,e2),
  SUCC → (ESUCC ,e1,e2),
  OTHERWISE → impl-error
  ("Unrecognized Stage 2 VHDL ‘enumeration’ relational operator: "a", relational-op)),

*POLY*
→ (case relational-op
  EQ → (EQ ,e1,e2),
  NE → (NEQ ,e1,e2),
  OTHERWISE → impl-error
  ("Unrecognized Stage 2 VHDL ‘polymorphic’ relational operator: "a", relational-op)),

*VHDLTIME*
→ (case relational-op
  EQ → (EQ ,e1,e2),
  NE → (NEQ ,e1,e2),
  LT → (TIMELT ,e1,e2),
  LE → (TIMELE ,e1,e2),
  GT → (TIMEGT ,e1,e2),
  GE → (TIMEGE ,e1,e2),
  OTHERWISE → impl-error
  ("Unrecognized Stage 2 VHDL ‘vhdltime’ relational operator: "a", relational-op)),

OTHERWISE → impl-error("Unsupported Stage 2 VHDL basic type "a.",type-tag))

mk-bool-eq(type-tag,e1,e2)
= (type-tag = *BOOL*
  → (simple-term(e1)
    → (simple-term(e2)→ (EQ ,e1,e2), (EQ ,e1,(COND ,e2,TRUE ,FALSE ))),
    simple-term(e2)→ (EQ ,e2,(COND ,e1,TRUE ,FALSE )),
    (COND ,e1,e2,(NOT ,e2))),
  (EQ ,e1,e2))

mk-bool-neq(type-tag,e1,e2)
= (type-tag = *BOOL*
  → (simple-term(e1)
    → (simple-term(e2)→ (NEQ ,e1,e2), (NEQ ,e1,(COND ,e2,TRUE ,FALSE ))),
    simple-term(e2)→ (NEQ ,e2,(COND ,e1,TRUE ,FALSE )),
    (EQ ,e1,e2))
\[ (\text{COND} ,e_1,e_2,(\text{NOT} ,e_2)) , \]
\[ (\text{NEQ} ,e_1,e_2)) \]

\[ \text{simple-term(term)} = \text{let operators} = (\text{DOT POUND}) \text{ in} \]
\[ \neg\text{consp(term)}\forall \text{hd(term)}\subseteq \text{operators} \]

\[ \text{mk-vhdltime(global)(delta)} = (\text{VHDLTIME} \ ,\text{global},\text{delta}) \]

\[ \text{block-exit(v)(stk)} = \text{let} \ <\text{tg,qname,pth,g}> = \text{hd(stk)} \text{ in} \]
\[ \begin{align*}
\text{(case} & \text{tg) } \\
*\text{STKBOTTOM}* & \to \text{model-execution-complete(qname)}, \\
*\text{UNDECLARE}* & \to g(\lambda v, s. \text{block-exit}(v)(s))(v)(\text{stk}), \\
(*\text{BLOCK-EXIT*} , *\text{SUBPROGRAM-RETURN*} ) & \to g(v)(\text{stk-pop(stk)}), \\
(*\text{BEGIN*} , *\text{LOOP-EXIT*} , *\text{PACKAGE-BODY-EXIT*} ) & \to \text{block-exit(v)(stk-pop(stk))}, \\
\text{OTHERWISE} & \to \text{impl-error("Unknown execution stack descriptor with tag: "a",tg))} \\
\end{align*} \]

\[ \text{model-execution-complete(id)} = (\text{mk-sd(id)(e)(e)(e)(((VHDL\text{MODEL}\_\text{EXECUTION}\_\text{COMPLETE} ,id))))} \]

A Stage 2 VHDL design file has a name, and consists of some (possibly none) package declarations, package bodies, and USE clauses, followed by an entity declaration and an architecture body.

The semantics of the design file has as its sole semantic argument the TSE \( t \) constructed by Phase 1. The design file name \( id \) denotes a special place, whose value \( .id \) is itself a place that will represent, at any given point during the translation, the current universe of visible places. This name is available to most of the Phase 2 semantic functions as the first edge label in the current path.

Translation of a design file commences by generating some top-level assertions and declarations for the SDVS Simplifier:

- **A disjointness assertion**, required for technical reasons.

  The function \( \text{mk-disjoint(place-list)} \) generates an SDVS assertion stating that the places in \( \text{place-list} \) are mutually disjoint.

- **A covering assertion** that the initial universe of visible places \( .id \) consists of certain predefined places: the program counter place \( \text{id}\backslash\text{pc} \) as well as the places \( \text{vhdltime} \) and \( \text{vhdltime.previous} \).

  The function \( \text{mk-cover(place, place-list)} \) generates an SDVS covering assertion that \( \text{place} \) covers all the places in \( \text{place-list} \) and that all of the places in \( \text{place-list} \) are mutually disjoint.

---

\[^{2}\text{The function}\ \text{mk-cover}\ \text{has in some instances been superseded by}\ \text{mk-cover-already};\ \text{it implements an experimental new naming scheme for VHDL variables. The scheme is available only when the SDVS function}\ \text{new-declarations}\ \text{is defined to return non-NIL. In SDVS Version 11, this new scheme is not available, so we will not discuss the actions of this function here.} \)
• Declarations of the places vhdltime and vhdltime-previous. The function mk-scalar-decl(placename,place-type) (make scalar declaration) generates an SDVS declaration of a scalar-value place of the indicated type.

• Assertions that the places vhdltime and vhdltime-previous have as their initial value the time object vhdltime(O,O) of the Simplifier VHDL Time domain.

The function mk-rel(type-desc)(relation,accessed-place,expression) (make relation) constructs an SDVS typed relation that asserts that the value of a place at pre- or postcondition time stands in a certain relation to the value of an expression.

Then an SD that defines the execution of the hardware description is generated. The application of this SD leads to further usable SDs, whose generation in the absence of errors is accomplished by continuations. With respect to the TSE t, an initial path consisting of the design file's name, an initial universe, and an initial execution stack containing a *STKBOTTOM* descriptor to terminate model execution (see Section 8.2), these SDs symbolically elaborate the design file's package declarations, package bodies, USE clauses, entity declaration, and architecture body.

8.4.2 Entity Declarations

(EN1) EN [ ENTITY id decl1 decld opt-id ] (t)(p)(u)(v)(stk)
  = let p1 = %(p)(id) in
    D [ decl ] (t)(p1)(u1)(v)(stk)
    where u1 = \nu1,stk1.D[ decl ] (t)(p1)(u)(v)(stk)

Phase 2 translation of an entity declaration effects the elaboration, via semantic function D, first of its port declarations, and then of any other declarations local to the entity. The interphase abstract syntax tree transformation has arranged for the Phase 2 abstract syntax of port declarations to be identical to that for other objects of class SIGNAL.

8.4.3 Architecture Bodies

(AR1) AR [ ARCHITECTURE id1 id2 decl* con-stat* opt-id ] (t)(p)(u)(v)(stk)
  = let p1 = %(p)(id1) in
    D [ decl* ] (t)(p1)(u1)(v)(stk)
    where
      u1 = \nu1,stk1.
      CS [ con-stat* ] (t)(p1)(u2)(v1)(stk1)
      where
        u2 = \nu2,stk2.
        cons((VHDL.MODEL.ELABORATION_COMPLETE ,hd(p)),
             (mk-sd
              (hd(p))(\varepsilon)(\varepsilon)
              ((make-vhdl-begin-model-execution
                (hd(p))(u)(v2)(stk2))))))

Phase 2 translation of an architecture body first effects the elaboration, via semantic function D, of the architecture's local declarations, and then initiates the translation, via semantic function CS, of its concurrent statements (which have been uniformly converted to

114
PROCESS statements by the interphase abstract syntax tree transformation at the end of Phase 1; see Section 7). The continuation of concurrent statement elaboration returns a Simpifier assertion to the effect that the VHDL model's elaboration is complete, as well as a state delta, constructed by special function make-vhdl-begin-model-execution, that initiates symbolic execution of the model.

8.4.4 Declarations

(D0) $D \{ \text{dec} \} (t)(p)(u)(v)(stk) = u(v)(stk)$

(D1) $D \{ \text{decl} \} \{ \text{decl} \} (t)(p)(u)(v)(stk)$
$$= D \{ \text{dec} \} (t)(p)(u_1)(v)(stk)$$

$$\text{where } u_1 = \lambda v_1, stk_1, D \{ \text{decl} \} (t)(p)(u_1)(v_1)(stk_1)$$

(D2) $D \{ \text{pkg-decl} \} \{ \text{pkg-decl} \} (t)(p)(u)(v)(stk)$
$$= D \{ \text{pkg-decl} \} (t)(p)(u_1)(v)(stk)$$

$$\text{where } u_1 = \lambda v_1, stk_1, D \{ \text{pkg-decl} \} (t)(p)(u_1)(v_1)(stk_1)$$

(D3) $D \{ \text{pkg-body} \} \{ \text{pkg-body} \} (t)(p)(u)(v)(stk)$
$$= D \{ \text{pkg-body} \} (t)(p)(u_1)(v)(stk)$$

$$\text{where } u_1 = \lambda v_1, stk_1, D \{ \text{pkg-body} \} (t)(p)(u_1)(v_1)(stk_1)$$

(D4) $D \{ \text{use-clause} \} \{ \text{use-clause} \} (t)(p)(u)(v)(stk)$
$$= D \{ \text{use-clause} \} (t)(p)(u_1)(v)(stk)$$

$$\text{where } u_1 = \lambda v_1, stk_1, D \{ \text{use-clause} \} (t)(p)(u_1)(v_1)(stk_1)$$

The Phase 2 processing of declarations proceeds sequentially, from first to last.

(D5) $D \{ \text{DEC} \} \{ \text{object-class} \} \{ \text{id} \} \{ \text{type-mark} \} \{ \text{opt-expr} \} (t)(p)(u)(v)(stk)$
$$= \text{let } d = \text{lookup-type-desc(type-mark)}(t)(p) \text{ in}$$

$$\text{case } \text{tag}(d)$$

$(*\text{BOOL}*, *\text{BIT}*, *\text{INT}*, *\text{REAL}*, *\text{TIME}*, *\text{ENUMTYPE}*)$

$\rightarrow \text{gen-scalar-decl}$

$$\text{(decl)(object-class)}(\text{id}+)(d)(opt-expr)(t)(p)(u)(v)(stk),$$

$*\text{ARRAYTYPE}*$

$\rightarrow \text{gen-array-decl}$

$$\text{(decl)(object-class)}(\text{id}+)(d)(\text{direction}(d))(\text{real-lb}(d))$$

$$\text{(real-ub}(d))(\text{elty}(d))(\text{opt-expr})(t)(p)(u)(v)(stk),$$

$*\text{RECORDTYPE}*$

$\rightarrow \text{gen-record-decl}$

$$\text{(decl)(object-class)}(\text{id}+)(d)(\text{opt-expr})(t)(p)(u)(v)(stk),$$

$\text{OTHERWISE} \rightarrow u(v)(stk))$

(D6) $D \{ \text{SLCDEC} \} \{ \text{object-class} \} \{ \text{id}+ \} \{ \text{slice-name} \} \{ \text{opt-expr} \} (t)(p)(u)(v)(stk)$
$$= \text{let } d = \text{lookup}(t)(p)(\text{hd}(\text{id}+)) \text{ in}$$

$$\text{let } \text{anom-array-type-desc} = \text{second}($d$) \text{ in}$$

$$\text{gen-array-decl}$$

$$\text{(decl)(object-class)}(\text{id}+)(\text{anom-array-type-desc})$$

$$\text{(direction}(\text{anom-array-type-desc}))(\text{lb}(\text{anom-array-type-desc}))$$

$$\text{(ub}(\text{anom-array-type-desc}))(\text{elty}(\text{anom-array-type-desc}))(\text{opt-expr})(t)(p)$$

$\text{(u)(v)(stk)}$

115
lookup-type-desc(id\*)\(t(p)\)
\(= (null(id\*) \rightarrow \text{void-type-desc()},\)
\hspace{1em} \text{let } q = \text{access(rest(id\*))(t)(p) in}
\hspace{1em} \text{lookup-desc(t)(q)(last(id\*))}\)

access(id\*)\(t(p)\)
\(= (null(id\*) \rightarrow p,\)
\hspace{1em} \text{let } d = \text{lookup(t)(p)(hd(id\*)) in}
\hspace{1em} \text{access(tl(id\*))}(t)(\%\text{(path}(d))(\text{idf}(d))))\)

lookup-desc(t)(p)(id)
\(= \text{let } d = t(p)(id) \text{ in}
\hspace{1em} (d = \text{*UNBOUND*} \rightarrow \text{lookup-desc(t)(rest(p))(id), d})\)

\(= (null(expr)\)
\hspace{1em} \rightarrow \text{gen-scalar-decl-id+}(decl)(object-class)(id\+)(d)(expr)(t)(p)(u)(v)(stk),\)
\hspace{1em} \text{gen-scalar-decl-id\*}(decl)(object-class)(id\*)(d)(expr)(t)(p)(u)(v)(stk))\)

\(= \text{object-class } \text{SIG}\)
\hspace{1em} \rightarrow \text{gen-scalar-signal-decl-id+}(decl)(id\+)(d)(expr)(t)(p)(u)(v)(stk),\)
\hspace{1em} \text{gen-scalar-nonsignal-decl-id+}(decl)(id\+)(d)(expr)(t)(p)(u)(v)(stk))\)

get-qualified-ids(suqn\*)(v)
\(= (null(suqn\*) \rightarrow e, \text{cons(qid(t)(p)(hd(id\*))}, \text{get-qualified-ids(tl(id\*))(t)(p)))}\)

get-qualified-ids(suqn\*)(v)
\(= (null(suqn\*) \rightarrow e, \text{cons(qid(hd(suqn\*))(v)}, \text{get-qualified-ids(tl(suqn\*))(v)))}\)
qualified-id(suqn)(v) = let vars = universe-vars(v) in
    let suqn-triple = assoc(suqn,vars) in
        (suqn-triple
            → let n = hd(third(suqn-triple)) in
                name-qualified-id(suqn)(n),
                name-qualified-id(suqn)(1))

name-qualified-id(suqn)(n) = (new-declarations() → (PLACELEMENT ,suqn,n),
    (n = 1 → suqn, catenate(suqn,"!*",n)))

already-qualified-id(suqn)(v) = ¬null(assoc(suqn,universe-vars(v)))

qualified-id-decls(suqn*)
    = (null(suqn*)→ e,
        let suqn = hd(suqn*) in
        cons((DECLARE ,suqn,(TYPE PLACEARRAY ) ),qualified-id-decls(t1(suqn*))))

mk-exists-already(duqn+ )(suqnl )(v) (sd) = (new-declarations()
    → (already-qualified-id(hd(suqn+ ))(v) → sd, mk-exists(suqn+ )(sd)),
    mk-exists(duqn+ )(sd))

mk-exists(suqn* )(sd) = sd

mk-qual-id-coverings(suqn+ )(duqn+ )(z)(v)
    = (new-declarations()
        → (already-qualified-id(hd(suqn+ ))(v)
            → (mk-rel(int-type-desc())((EQ ,pound(z),dot(z)))),
                nconc
                    (mk-disjoint(z,cons(dot(z),suqn+ )),mk-cover(pound(z),cons(dot(z),suqn+ ))),
                    qualified-id-decls(suqn+ )))

    = let type-spec = mk-type-spec(d)(t)(p) in
        (null(e)
            → nconc
                (mk-scalar-nonsignal-dec-post-declare(duqn* )(type-spec),
                    u(v)(stk)),
                nconc
                    (mk-scalar-nonsignal-dec-post-declare(duqn* )(type-spec),
                        u1(v)(stk))
                where
                    u1 = λv1,stki.
                        (mk-decl-sd (hd(p))(e)(e)(duqn* )
                            (nconc
                                (mk-scalar-nonsignal-dec-post-init(duqn* )(e)(d),
                                    u1(v1)(stki)))))

mk-type-spec(d)(t)(p)
    = (case tag(d)
        *BOOL* → (TYPE BOOLEAN ) ,
        *BIT* → (TYPE BIT ) ,
        *INT* → (TYPE INTEGER ) ,
        117
*REAL* → (TYPE FLOAT),
*TIME* → (TYPE INTEGER),
*VHDLTIME* → (TYPE VHDLTIME),
*VOID* → (TYPE VOID),
*POLY* → (TYPE POLYMORPHIC),
*WAVE* → (TYPE ,WAVEFORM ,mk-type-spec(hd(type(d)))(t)(p)),
*ENUMTYPE* → (idf(d)= CHARACTER → (TYPE CHARACTER),
       cons(TYPE ,cons(ENUMERATION ,literals(d)))),
*RECORDTYPE* → cons(TYPE ,cons(RECORD ,record-to-type(components(d))(t)(p))),
*ARRAYTYPE* → let expres = lb(d) in
       R [ expr1 ] (t)(p)(k1)(v1)(stk1).
       where
       ki = A(ei,fi),vi,stkj.
       let expres = ub(d) in
       where
       k2 = A(e2,f2),v2,stk2.
       cons(TYPE,
       (ARRAY ,e1,e2 ,mk-type-spec(elty(d))(t)(p))),
OTHERWISE — impl-error("Unrecognized Stage 2 VHDL type: -a7,tag(d)))
record-to-type(record-components)(t)(p)
= (null(record-components)→ ε,
   let (id,d) = hd(record-components) in
   cons((id,mk-type-spec(d)(t)(p)),
       record-to-type(tl(record-components))(t)(p)))
mk-scalar-nonsignal-dec-post-declare(duqn*)(type-spec)
= (null(duqn*)→ ε,
   let duqn = hd(duqn*) in
   cons(mk-scalar-decl(duqn,type-spec),
       mk-scalar-nonsignal-dec-post-declare(tl(duqn*))(type-spec)))
mk-scalar-decl(placename,place-type) = (DECLARE ,placename,place-type)
mk-scalar-nonsignal-dec-post-init(duqn*)(e)(d)
= (null(duqn*)→ ε,
   let duqn = hd(duqn*) in
   nconc
   (assign(d)((duqn,e)),mk-scalar-nonsignal-dec-post-init(tl(duqn*))(e)(d)))
assign(d)(target,value)
= (case tag(d)
   (*BOOL*, *BIT*, *INT*, *REAL*, *ENUMTYPE*, *POLY*, *VHDLTIME*, *WAVE*)
   → (mk-rel(d)((EQ ,pound(target),value))),
   *TIME* → (mk-rel(int-type-desc()((EQ ,pound(target),value)))),
   *ARRAYTYPE* → (is-bitvector-tdesc?(d)
     → (is-constant-bitvector?(value)
       → (case direction(d)
         TO
         → assign-array-to
         (target)(value)(elty(d))(ORIGIN ,target))(0),
DOWNTO
       → assign-array-downto
       )
   ))
(target)(value)(elty(d))
(mk-exp2
  (SUB ,
   (mk-exp2(ADD ,(ORIGIN ,target),(RANGE ,target)),1))(0),
OTHERWISE → impl-error("Illegal direction: "a",direction (d))),

(mk-rel(d)((EQ ,pound(target),value))),(d)

(is-string-tdesc?(d)
  → (is-constant-string?(value)
    → (case direction(d)
      TO   → assign-array-to
             (target)(value)(elty(d))(ORIGIN ,target))(0),
      DOWNTO
        → assign-array-downto
           (target)(value)(elty(d))
           (mk-exp2
            (SUB ,
             (mk-exp2(ADD ,(ORIGIN ,target),(RANGE ,target)),1))(0),
OTHERWISE → impl-error("Illegal direction: "a",direction (d))),

(mk-rel(d)((EQ ,pound(target),value))),(d))

(is-constant-bitvector?(expr*)
 = null(expr*)
 ∨ (consp(expr*)
    ∧ let expr1 = hd(expr*) in
     consp(expr1)∧ hd(expr1)= BS )

(is-constant-string?(expr*)
 = null(expr*)
 ∨ (consp(expr*)
    ∧ let expr1 = hd(expr*) in
     consp(expr1)∧ hd(expr1)= CHAR )

not-dotted-expr-p(expr) = ∀(consp(expr)∧ hd(expr)= DOT )

assign-array-to(target)(aggregate)(element-type-desc)(start-index)(m)
 = (null(aggregate)→ ε,
   nconc
   119
(assign
 element-type-desc)
 (((ELEMENT ,target,mk-exp2(ADD ,start-index,m)),hd(aggregate))),
 assign-array-to
 (target)(tl(aggregate))(element-type-desc)(start-index)(m+1))

 assign-array-downto(target)(aggregate)(element-type-desc)(start-index)(m)
 = (null(aggregate)→ e,
 nconc
 (assign
 element-type-desc)
 (((ELEMENT ,target,mk-exp2(SUB ,start-index,m)),hd(aggregate))),
 assign-array-downto
 (target)(tl(aggregate))(element-type-desc)(start-index)(m+1))

 mk-exp2(binary-op,e1,e2)
 = (case binary-op
 AND → (AND ,e1,e2),
 NAND → (NAND ,e1,e2),
 OR → (OR ,e1,e2),
 NOR → (NOR ,e1,e2),
 XOR → (XOR ,e1,e2),
 BAND → (USAND ,e1,e2),
 BNAND → (USNAND ,e1,e2),
 BOR → (USOR ,e1,e2),
 BNOR → (USNOR ,e1,e2),
 BXOR → (USXOR ,e1,e2),
 ADD → (PLUS ,e1,e2),
 SUB → (MINUS ,e1,e2),
 MUL → (MULT ,e1,e2),
 DIV → (DIV ,e1,e2),
 MOD → (MOD ,e1,e2),
 REM → (REM ,e1,e2),
 EXP → (EXPT ,e1,e2),
 (RPLUS ,RMINUS ,RTIMES ,RDIV ,REXPT ) → (binary-op,e1,e2),
 CONCAT → (ACONC ,e1,e2),
 OTHERWISE
 → impl-error("Unrecognized Stage 2 VHDL binary operator: "a",binary-op))

 assign-record(comp*)(e1,e2)
 = (null(comp*)→ e,
 let (id,d) = hd(comp*) in
 nconc
 (assign(d)((mk-recelt(e1,id),second(assoc(id,e2)))),
 assign-record(tl(comp*)((e1,e2))))

 mk-recelt(e)(id) = (RECORD ,e,id)

 = R [ expr ] (t)(p)(k)(v)(stk)
 where
 k = λ(e,f),v1,stk1.
 let z = hd(p)
 and signal-suqn+ = get-qids(id+)(t)(p) in
 let driver-suqn+ = name-drivers(signal-suqn+) in
 let suqn+ = append(signal-suqn+,driver-suqn+) in

 120
let $v_2 = \text{push-universe}(v_1)(v)(\text{suqn}^+)$ in
let signal-duqn$^+$ = get-qualified-ids(signal-suqn$^+')(v_2)$
and driver-duqn$^+$ = get-qualified-ids(driver-suqn$^+')(v_2)$ in
let duqn$^+$ = append(signal-duqn$^+',$driver-duqn$^+')$ in
(mk-exists-already
 (duqn$^+')(\text{suqn}^+')(v)
 (mk-decl-sd
 (z)(f)(e)(z))
(nconc
 (mk-qual-id-coverings(\text{suqn}^+')(duqn$^+')(z)(v),
 mk-scalar-signal-dec-post
 (decl)((duqn$^+',$signal-duqn$^+',$driver-duqn$^+',$e,d))(t)(p)(u)
 (v_2)(\text{stk})))

name-drivers(signal-names)
= (null(signal-names) → ε,
 cons(name-driver(hd(signal-names)),name-drivers(tl(signal-names))))

= let type-spec = mk-type-spec(d)(t)(p) in
 nconc
 (mk-scalar-signal-dec-post-declare
 (signal-duqn$^*$(driver-duqn$^*$)(type-spec)(waveform-type-spec),
 u_1(v)(\text{stk}))
 where
 u_1 = \lambda v_1,\text{stk}_1.
 (mk-decl-sd
 (hd(p))(e)(duqn$^*$)
 (nconc
 (mk-scalar-signal-dec-post-init
 (signal-duqn$^*$(driver-duqn$^*$)(e)(d)(waveform-type-desc(d)),
 u(v_1)(\text{stk}_1))))

mk-scalar-signal-dec-post-declare(signal-duqn$^*$)(driver-duqn$^*$)(type-spec)(waveform-type-spec)
= (null(signal-duqn$^*') → ε,
 let signal-duqn = hd(signal-duqn$^*$)
 and driver-duqn = hd(driver-duqn$^*$) in
 nconc
 (mk-scalar-signal-decl
 ((signal-duqn,driver-duqn))((type-spec,waveform-type-spec)),
 mk-scalar-signal-dec-post-declare
 (tl(signal-duqn$^*$))(tl(driver-duqn$^*$))(type-spec)(waveform-type-spec)))

mk-scalar-signal-dec(signal-name,driver-name)(type-spec,waveform-type-spec)
= (mk-scalar-decl(signal-name,type-spec),
 mk-scalar-decl(driver-name,waveform-type-spec),
 mk-scalar-signal-fn-decl(signal-name,driver-name))

mk-scalar-signal-fn-decl(signal-name,driver-name)
= (DECLARE ,signal-name,(TYPE ,FN ,(VAL ,dot(driver-name),dot(VHDLTIME )))

waveform-type-desc(type-desc) = <\text{WAVEFORM}, ε,*\text{WAVE*} ,(\text{STANDARD}), tt,type-desc>
mk-scalar-signal-dec-post-init(signal-duqn*) (driver-duqn*) (e) (type-desc) (waveform-type-desc)
= (null(signal-duqn*) → ε,
    let signal-duqn = hd(signal-duqn*)
    and driver-duqn = hd(driver-duqn*) in
    let initial-signal-val = (null(e) → eval-expr(dot(signal-duqn)), e) in
    let initial-waveform = init-scalar-signal
        (signal-duqn) (driver-duqn) (type-desc)
        (initial-signal-val) in
    nconc
        (assign(waveform-type-desc)((driver-duqn, initial-waveform)),
        mk-scalar-signal-dec-post-init
            (tl(signal-duqn*)) (tl(driver-duqn*)) (e) (type-desc) (waveform-type-desc)))

= (null(expr)
    → gen-array-decl-id+
        (element-type-desc)(expr)(t)(p)(u)(v)(stk),
    gen-array-decl-id*
        (element-type-desc)(expr)(t)(p)(u)(v)(stk))
real-lb(d)
= let bound = lb(d) in
    (consp(bound) ∧ hd(bound) = NUM → bound,
    (REF ,((SREF ,path(d),mk-tick-low(idf(d))))))
mk-tick-low(id) = catenate(id,"'LOW")
real-ub(d)
= (path(d) = (STANDARD) ∧ idf(d) ∈ (STRING BIT-VECTOR) → ε,
    let bound = ub(d) in
    (consp(bound) ∧ hd(bound) = NUM → bound,
    (REF ,((SREF ,path(d),mk-tick-high(idf(d))))))
mk-tick-high(id) = catenate(id,"'HIGH")
= (object-class = SIG
    → gen-array-signal-decl-id+
        (decl)(id+)(type-desc)(direction)(lower-bound)(upper-bound)
        (element-type-desc)(expr)(t)(p)(u)(v)(stk),
    gen-array-nonsignal-decl-id+
        (t)(p)(u)(v)(stk))
= (null(id+ → u(v, stk),
    let id+ = (hd(id+)) in
    gen-array-decl-id+
        where
        u₁ = λv₁ stk₁.
    gen-array-decl-id*
        (decl)(object-class)(tl(id+))(type-desc)(direction)(lower-bound)
        (upper-bound)(element-type-desc)(expr)(t)(p)(u)(v)(stk$_{1}$))
\[
\text{gen-array-nonsignal-decl+}(\text{id}^+)(\text{direction})(\text{expr}_1)(\text{expr}_2)(\text{element-type-desc})(\text{expr})(t)(p)(u)(v)(\text{stk})
\]

\[
= \text{R} \mid \text{expr} \mid (t)(p)(k)(v)(\text{stk})
\]

where
\[
k = \lambda (e_1,f_1,v_1,\text{stk}_1).
\]

\[
\text{R} \mid \text{expr}_1 \mid (t)(p)(k_1)(v_2)(\text{stk}_1)
\]

where
\[
k_1 = \lambda (e_1,f_1),v_2,\text{stk}_2.
\]

\[
\text{R} \mid \text{expr}_2 \mid (t)(p)(k_2)(v_3)(\text{stk}_2)
\]

where
\[
k_2 = \lambda (e_2,f_2),v_3,\text{stk}_3.
\]

let \( z = \text{hd}(p) \)

and \( \text{len} = \text{length-expr}(\text{expr}) \)

and \( \text{suqn}^+ = \text{get-gids}(\text{id}^+)(t)(p) \) in

let \( v_4 = \text{push-universe}(v_2)(z)(\text{suqn}^+) \) in

let \( \text{duqn}^+ = \text{get-qualified-ids}(\text{suqn}^+)(v_4) \) in

let \( g_1 = (e_1 \wedge e_2) \)

\( \rightarrow \text{mk-rel}(\text{int-type-desc})((\text{LE},e_1,e_2)), \)

TRUE \)

and \( g_2 = (e_1 \wedge e_2) \)

\( \rightarrow \text{mk-rel} \)

\( (\text{int-type-desc}) \)

\( ((\text{GE}, \text{mk-exp}^2(\text{ADD}, \text{mk-exp}^2(\text{SUB}, e_2, e_1)), 1), \text{len})) \)

TRUE \) in

(mk-exists-already)

\( (\text{duqn}^+)(\text{suqn}^+)(v) \)

(mk-decl-sd)

\( (z) \)

(nconc)

\( (t_1,f_2,(g_1), \)

\( (\text{len} = 0 \rightarrow f, \text{nconc}((g_2,f)))((e))(\text{le}) \)

(nconc)

(mk-qual-id-coverings)

\( (\text{suqn}^+)(\text{duqn}^+)(z)(v), \)

mk-array-nonsignal-dec-post

\( (\text{decl}) \)

\( ((\text{duqn}^+,e,\text{direction},e_1,e_2,\text{element-type-desc})) \)

\( (t)(p)(u)(v_4)(\text{stk}_3)))))))

length-expr(\text{expr})

\( = (\text{null}(\text{expr}) \rightarrow 0, \)

\( \text{hd}(\text{expr}) \in (\text{BITSTR STR PAGGR}) \rightarrow (\text{length}(\text{second}(\text{expr})), \)

1) \)

mk-array-nonsignal-dec-post(\text{decl})(\text{duqn}^*,e,\text{direction},\text{lower-bound},\text{upper-bound},\text{element-type-desc})(t)(p)(u)(v)(\text{stk})

\( = \text{let element-type-spec} = \text{mk-type-spec}(\text{element-type-desc})(t)(p) \) in

\( (\text{null}(e) \)

\( \rightarrow \text{nconc} \)

(mk-array-nonsignal-dec-post-declare)

\( (\text{duqn}^*)(\text{element-type-spec})(\text{direction})(\text{lower-bound})(\text{upper-bound}), \)

\( u(v)(\text{stk})), \)

\( \text{nconc} \)

(mk-array-nonsignal-dec-post-declare)

\( (\text{duqn}^*)(\text{element-type-spec})(\text{direction})(\text{lower-bound})(\text{upper-bound}), \)

\( u_1(v)(\text{stk}))))

123
where
\[ u_1 = \lambda v_1, \text{stk}_1. \]

\[
\begin{align*}
\text{(mk-decl-sd)} & \quad \text{(hd(p))(e)(duqn*)} \\
\text{(nconc)} & \quad (\text{direction} = \text{TO}) \\
& \quad \rightarrow \text{mk-array-nonsignal-dec-post-init-to} \\
& \quad \text{(duqn*)}(e)(\text{element-type-desc})(\text{lower-bound}), \\
& \quad \text{mk-array-nonsignal-dec-post-init-downto} \\
& \quad \text{(duqn*)}(e)(\text{element-type-desc})(\text{upper-bound}), \\
& \quad u(v_1)(\text{stk}_1))))) \\
\end{align*}
\]

\[
\begin{align*}
\text{mk-array-nonsignal-dec-post-declare}(\text{duqn*})(\text{element-type-spec})(\text{direction})(\text{lower-bound})(\text{upper-bound}) &= (\text{null}(\text{duqn*}) \rightarrow \epsilon, \\
& \quad \text{let duqn = hd(duqn*) in} \\
& \quad \text{nconc} \\
& \quad \text{(mk-vhdl-array-decl)} \\
& \quad (\text{duqn})(\text{element-type-spec})(\text{direction})(\text{lower-bound}) \\
& \quad ((\text{null}(\text{upper-bound}) \\
& \quad \rightarrow (\text{lower-bound} = 1 \rightarrow (\text{RANGE}, \text{duqn}), \\
& \quad \text{mk-exp2}(\text{SUB}, \text{mk-exp2}(\text{ADD}, (\text{RANGE}, \text{duqn}), \text{lower-bound}), 1)), \\
& \quad \text{upper-bound})), \\
& \quad \text{mk-array-nonsignal-dec-post-declare} \\
& \quad (\text{tl}(\text{duqn*}))(\text{element-type-spec})(\text{direction})(\text{lower-bound})(\text{upper-bound})) \\
\end{align*}
\]

\[
\begin{align*}
\text{mk-vhdl-array-decl}(\text{id})(\text{element-type-spec})(\text{direction})(\text{lower-bound})(\text{upper-bound}) &= \text{case second}(\text{element-type-spec}) \\
& \quad \text{BIT} \\
& \quad \rightarrow (\text{mk-array-decl}(\text{id})(\text{element-type-spec})(\text{lower-bound})(\text{upper-bound}), \\
& \quad \text{mk-bitvec-fn-decl}(\text{id})(\text{direction})(\text{lower-bound})(\text{upper-bound})), \\
\text{CHARACTER} & \quad \rightarrow (\text{mk-array-decl}(\text{id})(\text{element-type-spec})(\text{lower-bound})(\text{upper-bound}), \\
& \quad \text{mk-string-fn-decl}(\text{id})(\text{direction})(\text{lower-bound})(\text{upper-bound})), \\
\text{OTHERWISE} & \quad \rightarrow (\text{mk-array-decl}(\text{id})(\text{element-type-spec})(\text{lower-bound})(\text{upper-bound}))) \\
\end{align*}
\]

\[
\begin{align*}
\text{mk-array-decl}(\text{id})(\text{element-type-spec})(\text{lower-bound})(\text{upper-bound}) &= (\text{DECLARE}, \text{id}, (\text{TYPE}, \text{ARRAY}, \text{lower-bound}, \text{upper-bound}, \text{element-type-spec})) \\
\text{mk-bitvec-fn-decl}(\text{bitvec-name})(\text{direction})(\text{lower-bound})(\text{upper-bound}) &= \text{let bitvec-elt-names} = (\text{direction} = \text{TO} \\
& \quad \rightarrow \text{mk-slice-elt-names-to} \\
& \quad \text{(bitvec-name)}(\text{lower-bound})(\text{upper-bound}), \\
& \quad \text{mk-slice-elt-names-downto} \\
& \quad \text{(bitvec-name)}(\text{lower-bound})(\text{upper-bound})) \text{ in} \\
& \quad (\text{DECLARE}, \text{bitvec-name}, (\text{TYPE}, \text{FN}, \text{concatenate-bits}(\text{bitvec-elt-names}))) \\
\text{mk-string-fn-decl}(\text{string-name})(\text{direction})(\text{lower-bound})(\text{upper-bound}) &= \text{let string-elt-names} = (\text{direction} = \text{TO} \\
& \quad \rightarrow \text{mk-slice-elt-names-to} \\
& \quad \text{(string-name)}(\text{lower-bound})(\text{upper-bound}), \\
& \quad \text{mk-slice-elt-names-downto} \\
& \quad \text{(string-name)}(\text{lower-bound})(\text{upper-bound})) \text{ in} \\
& \quad (\text{DECLARE}, \text{string-name}, (\text{TYPE}, \text{FN}, \text{concatenate-characters}(\text{string-elt-names}))) \\
\text{mk-slice-elt-names-to}(\text{slice-name})(\text{lower-bound})(\text{upper-bound}) &= (\text{lower-bound} > \text{upper-bound} \rightarrow \epsilon, \\
& \quad \text{cons}(\text{mk-array-elt}(\text{slice-name})(\text{lower-bound}), \\
& \quad \text{mk-slice-elt-names-to}(\text{slice-name})(\text{lower-bound}+1)(\text{upper-bound}))) \\
\end{align*}
\]
\( \text{mk-slice-elt-names-downto}(\text{slice-name})(\text{lower-bound})(\text{upper-bound}) = (\text{upper-bound} < \text{lower-bound} \rightarrow \varepsilon, \text{cons}(\text{mk-array-elt}(\text{slice-name})(\text{upper-bound}), \text{mk-slice-elt-names-downto}(\text{slice-name})(\text{lower-bound})(\text{upper-bound} - 1))) \)

\( \text{mk-array-elt}(id)(e) = (\text{ELEMENT}, id, e) \)

\( \text{concatenate-bits}(\text{bit-names}) = \text{cons}((\text{USCONC}, \text{mk-dotted-names}(\text{bit-names})) \)

\( \text{concatenate-characters}(\text{char-names}) = \text{cons}((\text{ACONC}, \text{mk-dotted-names}(\text{char-names})) \)

\( \text{mk-dotted-names}(\text{names}) = (\text{null}(\text{names}) \rightarrow \varepsilon, \text{cons}(\text{dot}(\text{hd}(\text{names})), \text{mk-dotted-names}(\text{tl}(\text{names})))) \)

\( \text{mk-array-nonsignal-dec-post-init-to}(\text{duqn}^*)(e)(\text{element-type-desc})(\text{lower-bound}) = (\text{null}(\text{duqn}^*) \rightarrow \varepsilon, \text{nconc} (\text{assign-array-to}(\text{hd}(\text{duqn}^*))(e)(\text{element-type-desc})(\text{lower-bound})(0), \text{mk-array-nonsignal-dec-post-init-to}(\text{tl}(\text{duqn}^*))(e)(\text{element-type-desc})(\text{lower-bound}))) \)

\( \text{mk-array-nonsignal-dec-post-init-downto}(\text{duqn}^*)(e)(\text{element-type-desc})(\text{upper-bound}) = (\text{null}(\text{duqn}^*) \rightarrow \varepsilon, \text{nconc} (\text{assign-array-downto}(\text{hd}(\text{duqn}^*))(e)(\text{element-type-desc})(\text{upper-bound})(0), \text{mk-array-nonsignal-dec-post-init-downto}(\text{tl}(\text{duqn}^*))(e)(\text{element-type-desc})(\text{upper-bound}))) \)

\( \text{gen-array-signal-decl-id}(\text{decl})(id^+)(\text{type-desc})(\text{direction})(\text{expr}_1)(\text{expr}_2)(\text{element-type-desc})(\text{expr})(t)(p)(u)(v)(\text{stk}) = \text{R}(\text{expr}) \) where \
\( k = \lambda(e, f), \text{v}_1, \text{v}_2, \text{stk}_1 \).
\( \text{R}((\text{expr})_1)(t)(p)(k_1)(\text{v}_1)(\text{v}_2)(\text{stk}_1) \) where \
\( k_1 = \lambda(e_1, f_1), \text{v}_2, \text{stk}_2 \).
\( \text{R}((\text{expr})_2)(t)(p)(k_2)(\text{v}_1)(\text{v}_2)(\text{stk}_2) \) where \
\( k_2 = \lambda(e_2, f_2, \text{v}_3, \text{stk}_3) \).
\( \text{let} \ z = \text{hd}(p) \)
\( \text{and} \ \text{len} = \text{length-expr}(\text{expr}) \)
\( \text{and} \ \text{signal-suqn}^+ = \text{get-qids}(id^+)(t)(p) \) in \
\( \text{let} \ \text{driver-suqn}^+ = \text{name-drivers}(\text{signal-suqn}^+) \) in \
\( \text{let} \ \text{suqn}^+ = \text{append}(\text{signal-suqn}^+, \text{driver-suqn}^+) \) in \
\( \text{let} \ \text{v}_4 = \text{push-universe}(\text{v}_3)(z)(\text{suqn}^+) \) in \
\( \text{let} \ \text{signal-duqn}^+ = \text{get-qualified-ids} (\text{signal-suqn}^+)(\text{v}_4) \)
\( \text{and} \ \text{driver-duqn}^+ = \text{get-qualified-ids} (\text{driver-suqn}^+)(\text{v}_4) \) in \
\( \text{let} \ \text{duqn}^+ = \text{append} (\text{signal-suqn}^+, \text{driver-suqn}^+) \) in \
\( \text{let} \ g_1 = (e_1 \wedge e_2) \rightarrow \text{mk-rel} (\text{int-type-desc})(\text{LE}, e_1, e_2) \)
\( \text{and} \ g_2 = (e_1 \wedge e_2) \rightarrow \text{mk-rel} \)

125
((int-type-desc())
  ((GE,
    mk-exp2
    (ADD,
     mk-exp2(SUB,e2,e1),1),len)),
   TRUE) in
  (mk-exists-already
   (duqn+)(suqn+)(v)
   (mk-decl-sd
    (e)
    (nconc
     (f1,f2,g1),
     (len = 0 → f, nconc(f, (g2))))(e)((z))
    (nconc
     (mk-qual-id-coverings
      (suqn+)(duqn+)(z)(v),
      mk-array-signal-dec-post
      (decl)
      ((duqn+,signal-duqn+,driver-duqn+,e,type-desc,direction,
        e1,e2,element-type-desc))(t)(p)
      (u)(v)(stk))))

mk-array-signal-dec-post(decl)(duqn*,signal-duqn*,driver-duqn*,e,type-desc,direction,
= let element-type-spec = mk-type-spec(element-type-desc)(t)(p) in
  let element-waveform-type-spec = mk-waveform-type-spec(element-type-spec) in
  nconc
    (mk-array-signal-dec-post-declare
     (signal-duqn*)(driver-duqn*)(element-type-spec)
     (element-waveform-type-spec)(element-type-desc)(direction)(lower-bound)
     (upper-bound)(t)(p)(v)(stk),u1(v)(stk))
  where
    u1 = λv1,stk1.
    (mk-decl-sd
     (hd(p))(e)(e)(duqn*)
     (nconc
      (mk-array-signal-dec-post-init
       (signal-duqn*)(driver-duqn*)(e)(type-desc)(direction)
       (lower-bound)(upper-bound)(element-type-desc)
       (waveform-type-desc(element-type-desc))(t)(p)(v)(stk),
       u(v1)(stk1))))

mk-waveform-type-spec(type-spec)
= (case second(type-spec)
    ARRAY → append(rest(type-spec),(mk-waveform-type-spec(last(type-spec)))),
    OTHERWISE — (TYPE,WAVEFORM,type-spec))

mk-array-signal-dec-post-declare(signal-duqn*)(driver-duqn*)(element-type-spec)
= (null(signal-duqn*) → ε,
  let signal-duqn = hd(signal-duqn*)
  and driver-duqn = hd(driver-duqn*) in
  nconc
    (mk-array-signal-decl
     (signal-duqn)(driver-duqn)(element-type-spec)
     (element-waveform-type-spec)(element-type-desc)(direction)(lower-bound)
     (upper-bound)(t)(p)(v)(stk),

126
mk-array-signal-dec-post-declare
((tl(signal-duqn*))(tl(driver-duqn*))(element-type-spec)
(element-waveform-type-spec)(element-type-desc)(direction)(lower-bound)
(upper-bound)(t)(p)(v)(stk)))

mk-array-signal-dec(signal-name)(driver-name)(element-type-spec)(element-waveform-type-spec)
= nconc
  (mk-vhdl-array-decl
   (signal-name)(element-type-spec)(direction)(lower-bound)
   ((null(upper-bound)
     (-> (lower-bound = 1 -> (RANGE ,signal-name),
     mk-exp2(SUB ,mk-exp2(ADD ,((RANGE ,signal-name),lower-bound),1)),
     upper-bound)),
   (mk-array-decl
    (driver-name)(element-waveform-type-spec)(lower-bound)
    ((null(upper-bound)
      (-> (lower-bound = 1 -> (RANGE ,driver-name),
      mk-exp2(SUB ,mk-exp2(ADD ,((RANGE ,driver-name),lower-bound),1)),
      upper-bound))
    ,
  mk-array-signal-elt-in-decls

(t)(p)(v)(stk)
= (is-array-tdesc?(element-type-desc)
  -> let signal-elts = mk-slice-elt-names-to
  and driver-elts = mk-slice-elt-names-to
  (driver-duqn)(lower-bound)(upper-bound) in
  let expr₁ = real-lb(element-type-desc) in
  R. [ expr₁ ] (t)(p)(k₁)(v₁)(stk)
  where
  k₁ = λ(e₁,f₁),v₁,stk₁.
  let expr₂ = real-ub(element-type-desc) in
  R. [ expr₂ ] (t)(p)(k₂)(v₂)(stk₂)
  where
  k₂ = λ(e₂,f₂),v₂,stk₂.
  mk-array-signal-elt-in-decls-aux
  (signal-elts)(driver-elts)(elty(element-type-desc))
  (e₁)(e₂)(t)(p)(v₁)(v₂)(stk₁)
  let scalar-signal-elts = mk-slice-elt-names-to
  (signal-duqn)(lower-bound)(upper-bound)
  and scalar-driver-elts = mk-slice-elt-names-to
  (driver-duqn)(lower-bound)(upper-bound) in
  mk-scalar-signal-elt-in-decls(scalar-signal-elts,scalar-driver-elts))

(t)(p)(v)(stk)
= (null(signal-duqn*) -> e,
  let signal-duqn = hd(signal-duqn*)
  and driver-duqn = hd(driver-duqn*) in
  nconc
  (mk-array-signal-elt-in-decls
   (t)(p)(v)(stk)),
mk-array-signal-elt-fn-decls-aux
  (tl(signal-duqn*))(tl(driver-duqn*))((element-type-desc)(lower-bound))
  (upper-bound)(t)(p)(v)(stk)))

mk-scalar-signal-fn-decls(signal-names,driver-names)
  = (null(signal-names) → e,
      cons((mk-scalar-signal-fn-decl(hd(signal-names),hd(driver-names)),
      mk-scalar-signal-fn-decls(tl(signal-names),tl(driver-names))))

  ((element-type-desc)((element-waveform-type-desc)(t)(p)(v)(stk)))
  = (direction = TO
      → mk-array-signal-dec-post-init-to
      (signal-duqn*)(driver-duqn*)(e)(type-desc)(lower-bound)(upper-bound)
      ((element-type-desc)((element-waveform-type-desc)(t)(p)(v)(stk)),
      mk-array-signal-dec-post-init-downto
      (signal-duqn*)(driver-duqn*)(e)(type-desc)(lower-bound)(upper-bound)
      ((element-type-desc)((element-waveform-type-desc)(t)(p)(v)(stk)))

mk-array-signal-dec-post-init-to(signal-duqn*)(driver-duqn*)(e)(type-desc)((lower-bound)(upper-bound)
  ((element-type-desc)((element-waveform-type-desc)(t)(p)(v)(stk)))
  = (is-array-tdesc?(element-type-desc)
      → let expr1 = real-lb(element-type-desc) in
      R [ expr1 ] (t)(p)(k1)(v)(stk)
      where
      k1 = λ(ei,fi),v1,stk1.
      let expr2 = real-ub(element-type-desc) in
      R [ expr2 ] (t)(p)(k2)(v1)(stk1)
      where
      k2 = λ(e2,f2),v2,stk2.
      mk-array-signal-dec-post-init-elt-arrays-to
      (signal-duqn*)(driver-duqn*)(e)(type-desc)
      ((lower-bound)(upper-bound)(element-type-desc)
      (direction(element-type-desc))(e1)((e2)(t)(p)(v2)(stk2),
      mk-array-signal-dec-post-init-elt-scalars-to
      (signal-duqn*)(driver-duqn*)(e)(type-desc)(lower-bound)(upper-bound)
      ((element-type-desc)((element-waveform-type-desc)(t)(p)(v)(stk)))

  ((element-type-desc)((element-waveform-type-desc)(t)(p)(v)(stk)))
  = (is-array-tdesc?(element-type-desc)
      → let expr1 = real-lb(element-type-desc) in
      R [ expr1 ] (t)(p)(k1)(v)(stk)
      where
      k1 = λ(ei,fi),v1,stk1.
      let expr2 = real-ub(element-type-desc) in
      R [ expr2 ] (t)(p)(k2)(v1)(stk1)
      where
      k2 = λ(e2,f2),v2,stk2.
      mk-array-signal-dec-post-init-elt-arrays-downto
      (signal-duqn*)(driver-duqn*)(e)(type-desc)
      ((lower-bound)(upper-bound)(element-type-desc)
      (direction(element-type-desc))(e1)((e2)(t)(p)(v2)(stk2),
      mk-array-signal-dec-post-init-elt-scalars-downto
      (signal-duqn*)(driver-duqn*)(e)(type-desc)(lower-bound)(upper-bound)
      ((element-type-desc)((element-waveform-type-desc)(t)(p)(v)(stk)))

128
mk-array-signal-dec-post-init-elt-arrays-to(signal-duqn*)(driver-duqn*)(e)(type-desc)
= (null(signal-duqn*) → e,
    let signal-duqn = hd(signal-duqn*)
    and driver-duqn = hd(driver-duqn*) in
    nconc
    (let signal-elts = mk-slice-elt-names-to
       (signal-duqn)(lower-bound)(upper-bound)
       and driver-elts = mk-slice-elt-names-to
       (driver-duqn)(lower-bound)(upper-bound) in
    mk-array-signal-dec-post-init-aux
    (signal-elts)(driver-elts)(e)(elt-type-desc)(elt-direction)
    (elt-lower-bound)(elt-upper-bound)(elt-waveform-type-desc)
    (t)(p)(v)(stk),
    mk-array-signal-dec-post-init-elt-arrays-downto
    (tl(signal-duqn*))(tl(driver-duqn*))(tl(e))(type-desc)(tl(lower-bound)
    (upper-bound)(elt-type-desc)(elt-direction)(elt-lower-bound)
    (elt-upper-bound)(t)(p)(v)(stk)),
    mk-array-signal-dec-post-init-aux(signal-duqn*)(driver-duqn*)(e)(type-desc)
= (null(signal-duqn*) → e,
    let signal-duqn = hd(signal-duqn*)
    and driver-duqn = hd(driver-duqn*) in
    nconc
    (let signal-elts = mk-slice-elt-names-downto
       (signal-duqn)(lower-bound)(upper-bound)
       and driver-elts = mk-slice-elt-names-downto
       (driver-duqn)(lower-bound)(upper-bound) in
    mk-array-signal-dec-post-init-aux
    (signal-elts)(driver-elts)(e)(elt-type-desc)(elt-direction)
    (elt-lower-bound)(elt-upper-bound)(elt-waveform-type-desc)
    (t)(p)(v)(stk),
    mk-array-signal-dec-post-init-elt-arrays-downto
    (tl(signal-duqn*))(tl(driver-duqn*))(tl(e))(type-desc)(tl(lower-bound)
    (upper-bound)(elt-type-desc)(elt-direction)(elt-lower-bound)
    (elt-upper-bound)(t)(p)(v)(stk)))

mk-array-signal-dec-post-init-aux(signal-duqn*)(driver-duqn*)(e)(type-desc)(direction)
= (null(signal-duqn*) → e,
    let signal-duqn = hd(signal-duqn*)
    and driver-duqn = hd(driver-duqn*) in
    nconc
    (mk-array-signal-dec-post-init
     ((signal-duqn))(driver-duqn).hd(e)(type-desc)(direction)
     (lower-bound)(upper-bound)(element-type-desc)(element-waveform-type-desc)
     (t)(p)(v)(stk),
    mk-array-signal-dec-post-init-aux
     (tl(signal-duqn*))(tl(driver-duqn*))(tl(e))(type-desc)(direction)
     (lower-bound)(upper-bound)(element-type-desc)(element-waveform-type-desc)
     (t)(p)(v)(stk)),
    mk-array-signal-dec-post-init-elt-scalars-to(signal-duqn*)(driver-duqn*)(e)(type-desc)
= (null(signal-duqn*) → e,
    let signal-duqn = hd(signal-duqn*)

129
\( \text{and } \text{driver-duqn} = \text{hd}(\text{driver-duqn}^*) \text{ in } \)

\[
\begin{align*}
\text{let initial-waveforms} &= \text{init-array-signal-to} \\
&\quad (\text{signal-duqn})(\text{driver-duqn})(e)(\text{type-desc}) \\
&\quad (\text{element-type-desc})(\text{lower-bound})(\text{upper-bound}) \text{ in } \\
\text{nconc} \\
&\quad (\text{assign-array-to}) \\
&\quad (\text{driver-duqn})(\text{initial-waveforms})(\text{element-waveform-type-desc}) \\
&\quad (\text{lower-bound})(0), \\
\text{mk-array-signal-dec-post-init-elt-scalars-to} \\
&\quad (\text{tl}(\text{signal-duqn}^*))((\text{tl}(\text{driver-duqn}^*)))(e)(\text{type-desc})(\text{lower-bound}) \\
&\quad (\text{upper-bound})(\text{element-type-desc})(\text{element-waveform-type-desc})(t)(p)(v) \\
&\quad (\text{stk})) \end{align*}
\]

\(\text{mk-array-signal-dec-post-init-elt-scalars-downto}(\text{signal-duqn}^*)(\text{driver-duqn}^*)(e)(\text{type-desc}) \\
(\text{lower-bound})(\text{upper-bound})(\text{element-type-desc})(\text{element-waveform-type-desc})(t)(p)(v)(\text{stk}) \\
= (\text{null}(\text{signal-duqn}^*) \rightarrow \epsilon, \\
\text{let } \text{signal-duqn} = \text{hd}(\text{signal-duqn}^*) \\
\text{and } \text{driver-duqn} = \text{hd}(\text{driver-duqn}^*) \text{ in } \\
\text{let initial-waveforms} = \text{init-array-signal-downto} \\
&\quad (\text{signal-duqn})(\text{driver-duqn})(e)(\text{type-desc}) \\
&\quad (\text{element-type-desc})(\text{lower-bound})(\text{upper-bound}) \text{ in } \\
\text{nconc} \\
&\quad (\text{assign-array-downto}) \\
&\quad (\text{driver-duqn})(\text{initial-waveforms})(\text{element-waveform-type-desc}) \\
&\quad (\text{upper-bound})(0), \\
\text{mk-array-signal-dec-post-init-elt-scalars-downto} \\
&\quad (\text{tl}(\text{signal-duqn}^*))((\text{tl}(\text{driver-duqn}^*)))(e)(\text{type-desc})(\text{lower-bound}) \\
&\quad (\text{upper-bound})(\text{element-type-desc})(\text{element-waveform-type-desc})(t)(p)(v) \\
&\quad (\text{stk})) \end{align*} \]

\[(D7) \text{ D } \text{ETDEC } \text{id }^+ \text{ } (t)(p)(u)(v)(\text{stk}) \]
\[
= (\text{mk-decl-sd} \\
(\text{hd}(p))(e)(e)(e) \\
(\text{nconc}((\text{mk-etdec-post}((\text{id}))(t)(p),u(v)(\text{stk})))))
\]

\[
\text{mk-etdec-post}(\text{type-mark})(t)(p) \\
= \text{let } d = \text{lookup-type-desc}(\text{type-mark})(t)(p) \text{ in } \\
\text{mk-enumlit-rels}(d)(\text{literals}(d))
\]

\[
\text{mk-enumlit-rels}(d)(\text{id}^*) \\
= (\text{null}(\text{tl}(\text{id}^*)) \rightarrow \epsilon, \\
\text{let } \text{id}_1 = \text{hd}(\text{id}^*) \\
\text{and } \text{id}_2 = \text{hd}(\text{tl}(\text{id}^*)) \text{ in } \\
\text{cons}(\text{mk-rel}(d)((\text{PRED}, \text{id}_1, \text{id}_2)), \text{mk-enumlit-rels}(d)(\text{tl}(\text{id}^*)))
\]

The translation of an enumeration type declaration emits an SDVS declaration of the enumeration type.

\[(D8) \text{ D } \text{ATDEC } \text{id }^+ \text{ discrete-range type-mark } (t)(p)(u)(v)(\text{stk}) \]
\[
= \text{let } (\text{direction}, \text{expr}_1, \text{expr}_2) = \text{discrete-range } \text{ in } \\
\text{let } \text{lower-bound} = (\text{direction} = \text{TO} \rightarrow \text{expr}_1, \text{expr}_2) \\
\text{and } \text{upper-bound} = (\text{direction} = \text{TO} \rightarrow \text{expr}_2, \text{expr}_1) \text{ in } \\
\text{attributes-low-high} \\
((\text{id}, \text{INTEGER}), \text{lower-bound}, \text{upper-bound})(t)(p)(u)(v)(\text{stk})
\]
attributes-low-high(id,attribute-type-mark,lower-bound,upper-bound)(t)(p)(u)(v)(stk)

\[ \text{let } \text{decl}_1 = (\text{DEC}, \text{SYSGEN}, (\text{mk-tick-low(id)}), \text{attribute-type-mark}, \text{lower-bound}) \text{ in } \]
\[ \text{let } \text{decl}_2 = (\text{DEC}, \text{SYSGEN}, (\text{mk-tick-high(id)}), \text{attribute-type-mark}, \text{upper-bound}) \text{ in } \]
\[ \text{let } \text{decl}^+ = (\text{decl}_1, \text{decl}_2) \text{ in } \]
\[ \text{D} \text{[ decl}^+ \text{ ]} (t)(p)(u)(v)(stk) \]

\[ \text{mk-tick-low(id)} = \text{catenate(id, "LOW")} \]
\[ \text{mk-tick-high(id)} = \text{catenate(id, "HIGH")} \]

An array type declaration declares and initializes the 'low and 'high array type attributes.

(D9) \[ \text{D} \text{[ PACKAGE id decl}^* \text{ opt-id ]} (t)(p)(u)(v)(stk) \]
\[ = \text{D} \text{[ decl}^* \text{ ]} (t)(\%(p)(id))(u)(v)(stk) \]

The declarations contained within a package are translated as usual, but in the package's context in the TSE, via the extended path \%(p)(id).

(D10) \[ \text{D} \text{[ PACKAGEBODY id decl}^* \text{ opt-id ]} (t)(p)(u)(v)(stk) \]
\[ = \text{D} \text{[ decl}^* \text{ ]} (t)(\%(p)(id))(u)(v)(stk) \]

\[ \text{pkg-body-exit(v)(stk)} \]
\[ = \text{let } <\text{tg}, \text{qname}, \text{pth}, \text{g}> = \text{hd(stk)} \text{ in } \]
\[ \text{case } \text{tg} \]
\[ *\text{STKBOTTOM}* \rightarrow \text{model-execution-complete(qname)}, \]
\[ *\text{UNDECLARE}* \rightarrow g(\lambda v,s.s.pkg-body-exit(v)(s))(v)(stk), \]
\[ (*\text{BEGIN}* \rightarrow \text{pkg-body-exit(v)(stk-pop(stk))}, \]
\[ (*\text{PACKAGE-BODY-EXIT* ,*LOOP-EXIT* ,*SUBPROGRAM-RETURN* )} \rightarrow g(v)(stk-pop(stk)), \]
\[ \text{OTHERWISE} \]
\[ \rightarrow \text{impl-error("Unknown execution stack descriptor with tag: "a",tg))} \]

The declarations contained in a package body are translated in the package's context in the TSE, via the extended path \%(p)(id). A *PACKAGE-BODY-EXIT* descriptor is first pushed onto the execution stack to prevent the package's declarations from being unelaborated when the package body is exited.

(D11) \[ \text{D} \text{[ PROCEDURE id proc-par-spec* type-mark ]} (t)(p)(u)(v)(stk) = u(v)(stk) \]

(D12) \[ \text{D} \text{[ FUNCTION id func-par-spec* type-mark ]} (t)(p)(u)(v)(stk) = u(v)(stk) \]

(D13) \[ \text{D} \text{[ SUBPROGBODY subprog-spec decl* seq-stat* opt-id ]} (t)(p)(u)(v)(stk) \]
\[ = u(v)(stk) \]

Subprogram declarations need no Phase 2 translation, nor do subprogram bodies.

(D14) \[ \text{D} \text{[ USE dotted-name* ]} (t)(p)(u)(v)(stk) = u(v)(stk) \]

The effect of USE clauses has already been recorded in the TSE during Phase 1; no further Phase 2 translation is necessary.
8.4.5 Concurrent Statements

\( CS \[ e \] (t)(p)(u)(v)(stk) = u(v)(stk) \)

\( CS \[ \text{con-stat con-stat} \] (t)(p)(u)(v)(stk) = CS \[ \text{con-stat} \] (t)(p)(u1)(v)(stk) \)
where \( u_1 = \lambda v, stk. CS \[ \text{con-stat} \] (t)(p)(u)(v)(stk) \)

A list of concurrent statements is translated in order, from first to last.

\( CS \[ \text{PROCESS id decl^* seq-stat^* opt-id} \] (t)(p)(u)(v)(stk) = let \( p_1 = \% (p)(id) \) in
  (mk-decl-sd
   (hd(p))(e)(e)
   ((make-vhdl-process-elaborate(id)(t)(p1)(seq-stat^*)(u1)(v)(stk))))
where \( u_2 = \lambda v, stk. D \[ \text{decl}^* \] (t)(p2)(u)(v)(stk) \)

8.4.6 Sequential Statements

\( SS \[ e \] (t)(p)(c)(v)(stk) = c(v)(stk) \)

\( SS \[ \text{seq-stat seq-stat} \] (t)(p)(c)(v)(stk) = SS \[ \text{seq-stat} \] (t)(p)(c1)(v)(stk) \)
where \( c_1 = \lambda v, stk. SS \[ \text{seq-stat} \] (t)(p)(c)(v)(stk) \)

A list of sequential statements is translated in order, from first to last.

\( SS \[ \text{NULL} \] (t)(p)(c)(v)(stk) = c(v)(stk) \)

NULL statements have no effect.

\( SS \[ \text{VARASSN ref expr} \] (t)(p)(c)(v)(stk) = let \( d = T \[ \text{ref} \] (t)(p) \) in
  E \[ \text{ref} \] (t)(p)(k1)(v)(stk)
where \( k_1 = \lambda (e1,f1), v1, stk1 \).
  R \[ expr \] (t)(p)(k2)(v1)(stk1)
where \( k_2 = \lambda (e2,f2), v2, stk2 \).
  (mk-sd
   (hd(p))(nconc(f1,f2))(e)(e1))
  nconc
   (assign(d)((e1,e2)),
    c(v2)(stk2)))

assign(d)(target,value)
= (case tag(d)
   (*BOOL*, *BIT*, *INT*, *REAL*, *ENUM*, *TYPE*, *POLY*, *VHDLTIME*, *WAVE*)
   \( \rightarrow (mk-rel(d)((EQ, pound(target), value))) \),
   *TIME* \( \rightarrow (mk-rel(int-type-desco)((EQ, pound(target), value))) \),

132
The translation of a variable assignment statement first translates its left and right parts, obtaining translated expressions and guard formulas. Note that the left part is translated by E and is therefore not dereferenced (by application of the dot function), as it would be if R were used instead. The precondition of the generated SD consists of the combined lists of guard formulas, and its mod list is the translated left part. Its postcondition asserts the new value of the left part place, and then asserts succeeding SDs by appropriately using the
c. Assignments in Stage 2 VHDL can be scalar or can assign entire arrays. Entire array assignments are asserted element by element via auxiliary semantic function array-signal-assignment.

(SS4) \[ \text{SIGASSN} \] delay-type ref waveform \[ (t)(p)(c)(v)(stk) \]
= \( \lambda \) d = \( \text{T} \) [ ref ] \[ (t)(p) \] in
(case tag(d)
   (*BOOL*, *BIT*, *INT*, *REAL*, *TIME*, *ENUMTYPE*)
   \( \rightarrow \) scalar-signal-assignment
   *ARRAYTYPE*
   \( \rightarrow \) array-signal-assignment
   (delay-type)(ref)(waveform)(t)(p)(c)(v)(stk),
   OTHERWISE
   \( \rightarrow \) impl-error
   ("Signal assignment not implemented for object ", ref,
   " of type ", d))

= \( \lambda \) ref \[ (t)(p)(k)(v)(stk) \]
where
\( k = \lambda \) (signal-name,guard),v,stk.
let driver-name = name-driver(signal-name) in
\[ \text{W} \] [ waveform ] \[ (t)(p)(wave-cont)(v)(stk) \]
where
wave-cont = \( \lambda \) (trans*,guard*),v,stk.
let all-guards = nconc(guard,guard*) in
(delay-type = TRANSPORT
   \( \rightarrow \) (mk-sd
      (hd(p))(all-guards)(e)((driver-name))
      nconc
      (assign
      (waveform-type-desc(d))
      ((driver-name, mk-transport-update
      (dot(driver-name))(trans*))),
      c(v)(stk)))),
let earliest-new-transaction = hd(trans*) in
(mk-sd
   (hd(p))
   (cons(mk-preemption
      (dot(driver-name))
      (earliest-new-transaction),all-guards))(e)((driver-name))
   nconc
   (assign
      (waveform-type-desc(d))
      ((driver-name, mk-inertial-update
      (dot(driver-name))(trans*))),
      c(v)(stk)))),
mk-sd
   (hd(p))
   (cons(mk-not
      (mk-preemption
      (dot(driver-name))
      (earliest-new-transaction)),
      c(v)(stk))))

134
all-guards)((e)(((driver-name)))
(nconc
(assign
(waveform-type-desc(d))
((driver-name,
mk-inertial-update
(dot(driver-name))(trans*))),
c(v)(stk))))

waveform-type-desc(type-desc) = <WAVEFORM,e,WAVE*,(STANDARD),tt,type-desc>

mk-transport-update(dot-driver)(trans*)
= cons(TRANSPORT_UPDATE,cons(dot-driver,trans*))

mk-preemption(dot-driver)(transaction)
= (PREEMPTION,dot-driver,transaction)

mk-inertial-update(dot-driver)(trans*)
= cons(INERTIAL_UPDATE,cons(dot-driver,trans*))

mk-not(e) = (NOT,e)

= let seq-stat⁺ = cascade-array-signal-assignment
    (delay-type)(ref)(waveform)(t)(p)(c)(v)(stk) in
    SS [ seq-stat⁺ ] (t)(p)(c)(v)(stk)

= let array-refs = mk-array-refs(ref)(t)(p)(c)(v)(stk) and element-waves = mk-element-waves(agg-wave)(t)(p)(c)(v)(stk) in
    mk-scalar-signal-assignments(delay-type)(array-refs)(element-waves)

mk-scalar-signal-assignments(delay-type)(array-refs)(element-waves)
= (null(array-refs)→ e,
    cons((SIGASSN,delay-type,hd(array-refs),hd(element-waves)),
    mk-scalar-signal-assignments
    (delay-type)(tl(array-refs))(tl(element-waves))))

mk-array-refs(ref)(t)(p)(c)(v)(stk)
= let d = T [ ref ] (t)(p) in
    let direction = direction(d) and expr₁ = lb(d) and expr₂ = ub(d) in
    R [ expr₁ ] (t)(p)(k₁)(v₁)(stk₁) where
    k₁ = λ(e₁,f₁),v₁,stk₁.
    R [ expr₂ ] (t)(p)(k₂)(v₁)(stk₁) where
    k₂ = λ(e₂,f₂),v₂,stk₂.
    let sref = hd(second(ref)) and indices = (direction = TO
    → gen-ascending-indices(e₁)(e₂),
    gen-descending-indices(e₁)(e₂)) in
    mk-array-refs-aux(sref)(indices)
gen-ascending-indices(min)(max)  
= (min > max → ε, cons(min,gen-ascending-indices(min+1)(max)))

gen-descending-indices(min)(max)  
= (max < min → ε, cons(max,gen-descending-indices(min)(max-1)))

mk-array-refs-aux(sref)(indices)  
= (null(indices) → ε,  
cons((REF, (sref, (INDEX, (NUM, hd(indices)))),  
mk-array-refs-aux(sref)(tl(indices)))))

mk-element-waves(agg-wave)(t)(p)(c)(v)(stk)  
= let aggregate-transactions = second(agg-wave) in  
  let element-transaction-lists = mk-element-transaction-lists  
    aggregate-transactions(t)(p)(c)(v)(stk) in  
    mk-element-waves-aux(element-transaction-lists)

mk-element-transaction-lists(aggregate-transactions)(t)(p)(c)(v)(stk)  
= (null(aggregate-transactions) → ε,  
cons(mk-transaction-list(hd(aggregate-transactions))(t)(p)(c)(v)(stk),  
mk-element-transaction-lists(tl(aggregate-transactions))(t)(p)(c)(v)(stk)))

mk-transaction-list(agg-trans) (t)(p)(c)(v)(stk)  
= let agg-value-expr = second(agg-trans)  
  and time-expr = third(agg-trans) in  
  let element-value-exprs = (case hd(agg-value-expr)  
    REF → mk-array-refs(agg-value-expr)(t)(p)(c)(v)(stk),  
    (BITSTR, STR, PAGGR) → hd(tl(agg-value-expr)),  
    OTHERWISE  
    → impl-error  
    (“Illegal aggregate in transaction: “,  
    agg-value-expr)) in  
  mk-simultaneous-transactions(element-value-exprs)(time-expr)

mk-simultaneous-transactions(expr*)(time-expr)  
= (null(expr*) → ε,  
cons((TRANS, hd(expr*),time-expr),  
mk-simultaneous-transactions(tl(expr*))(time-expr)))

(SS5) SS \[ IF \] cond-part+ else-part \[ \] (t)(p)(c)(v)(stk)  
= let seq-stat* = else-part in  
  gen-if(cond-part+)(seq-stat*)(seq-stat)(t)(p)(c)(v)(stk)

gen-if(cond-part*)(seq-stat*)(ifclause)(t)(p)(c)(v)(stk)  
= (null(cond-part*) → SS \[ seq-stat* \] \[ (t)(p)(c)(v)(stk)],  
  let (expr,seq-stat*) = hd(cond-part*) in  
  R \[ expr \] \[ (t)(p)(k)(v)(stk)  

where  
k = \lambda(e,f,v_1,stk_1).  
  (mk-sd  
    (hd(p))(cons(e,f))(v_1)(stk_1)  
    (let c_1 = \lambda v_2,stk_2. SS \[ seq-stat* \] (t)(p)(c)(v_2)(stk_2) in  
    c_1(v_1)(stk_1)),  
  mk-sd  
    (hd(p))(cons(mk-not(e,f))(v_1)(stk_1)  
    (let c_2 = \lambda v_3,stk_3.  
    gen-if  
    (tl(cond-part*))(seq-stat*)(v_3)(t)(p)(c)(v_3)(stk_3) in  
    c_2(v_1)(stk_1)))))

136
The abstract syntax of a Stage 2 VHDL IF statement consists of a finite, nonempty list of cond-parts followed by a (possibly empty) else-part. Each cond-part corresponds to an IF expr THEN seq-stats or an ELSIF expr THEN seq-stats construct in the concrete syntax. Thus each cond-part must be translated into two SDs: one for the case where expr evaluates to true and the other where it evaluates to false. The translation is performed by auxiliary semantic function gen-if, which takes as arguments (among others): the cond-part list and the seq-stats comprising the else-part. Successive recursive calls of gen-if process the first element of their cond-part list, reducing it to empty. When the cond-part list is empty, gen-if produces the translation of the else-part. The function mk-not constructs the logical negation of its argument.

(SS6) \( \text{CASE expr case-alt}^+ \) (t)(p)(c)(v)(stk) = R [ expr ] (t)(p)(k)(v)(stk)

where

\( k = \lambda(e,f),v_1,\text{stk}_1. \)

let \( d = \exists \ [\text{expr]} (t)(p) \) in

gen-case(e)(d)((e,f))(case-alt^+)(t)(p)(c)(v)(stk)

\( \text{gen-case(g)}(d)((e,f))(\text{case-alt}^+)(t)(p)(c)(v)(stk) \)

= (null(case-alt^+) \rightarrow e, 

let \( (h,\text{sd}) = \text{gen-alt(g)}(d)((e,f))(\text{hd(case-alt^+)})(t)(p)(c)(v)(stk) \) in

cons(sd,gen-case(append(g,h))(d)((e,f))(tl(case-alt^+))(t)(p)(c)(v)(stk)))

\( \text{gen-alt(g)}(d)((e,f))(\text{case-alt}^+)(t)(p)(c)(v)(stk) \)

= let \( \text{case-alt-tag} = \text{hd(case-alt)} \) in

(case-alt-tag = CASEOTHERS 

\rightarrow \text{let seq-stat^* = hd(tl(case-alt)) in} 

let \( c_1 = \lambda v_1,\text{stk}_1,SS [\text{seq-stat^*}] (t)(p)(c)(v)(stk) \) in

(e,

mk-sd

(hd(p))(append(f,(mk-not(mk-ors(g))))(e)(e)

(c_1)(v)(stk)))),

let \( (\text{case-set},\text{seq-stat}^*) = \text{tl(case-alt)} \) in

let \( c_1 = \lambda v_1,\text{stk}_1,SS [\text{seq-stat}^*] (t)(p)(c)(v)(stk) \) in

let \( h = \text{append}(t,\text{gen-guard(case-set)}(d)(e)(t)(p)) \) in

(h,mk-sd(hd(p))(h)(e)(e)(c_1)(v)(stk))))

\( \text{mk-ors(disjs)} \) = (case length(disjs)

1 \rightarrow \text{hd(disjs)},

2 \rightarrow \text{mk-or(hd(disjs))(hd(tl(disjs))),

OTHERWISE \rightarrow \text{mk-or(hd(disjs))(mk-ors(tl(disjs))))}

\( \text{mk-or(e1,e2)} \)

= (null(e1) \rightarrow e2, 

null(e2) \rightarrow e1, 

ccons(e1)\land ccons(e2) 

\rightarrow (hd(e1) = \text{OR} 

\rightarrow (hd(e2) = \text{OR} \rightarrow \text{cons(OR ,append(tl(e1),tl(e2))), append(e1,(e2)))}, 

hd(e2) = \text{OR} \rightarrow \text{nconc((OR ,e1),tl(e2))}, 

(OR ,e1,e2)), 

(OR ,e1,e2))

137
The abstract syntax of a CASE statement consists of a selector expression followed by a finite, nonempty list of case alternatives. Each case alternative consists of a list of sequential statements, preceded either by a nonempty list of discrete ranges (indicated by CASECHOICE) or (for the last alternative only) by CASEOTHERS. Each of these discrete range lists represents a set of values, called a case selection set. If the selector expression evaluates to one of these values, then the corresponding sequential statement list is executed, after which control passes to the successor of the CASE statement. CASEOTHERS represents a case selection set that is the complement of the union of all of the other case selection sets relative to the set of values in the selector expression’s type. Phase 1 has ensured that no case selection sets intersect.

The Phase 2 translation of a CASE statement first processes its selector expression, obtaining a translated expression and a guard formula. The translation is completed by the function gen-case, which takes the following arguments:

- a formula, initially empty, that is the disjunction of formulas representing the case selection sets of case alternatives translated so far in this CASE statement — this formula’s negation represents the case selection set indicated by CASEOTHERS (if present) in the CASE statement;
- the basic type of the selector expression (and the case selection set elements);
- the selector expression’s translation and guard formula;
- a list of case alternatives.
Each successive recursive call to gen-case processes the first element of its case alternative list, reducing the list to empty, at which time processing terminates normally. Each case alternative is processed by auxiliary semantic function gen-alt, which returns a formula representing the case selection set for that alternative and an SD representing the execution of the corresponding sequential statement list. This formula and SD are collected by gen-case; the final result returned by gen-case is a list of SDs. The function gen-guard converts discrete range lists into formulas representing case selection sets. The function \texttt{mk-or(formula}_1, \texttt{formula}_2)\) constructs the logical disjunction of two formulas; if one of the formulas is empty, then \texttt{mk-or} ignores it and returns the nonempty one.

(SS7) \(SS) SS [ \text{ LOOP } \text{id seq-stat}^* \text{ opt-id }] (t)(p)(c)(v)(stk) = \text{let } lp-desc = \langle \text{*LOOP-EXIT*}, \text{id}, \text{p}, \lambda v, s. c(v)(s) \rangle \text{ in}
\begin{align*}
\text{let } stk_1 = \text{stk-push}(lp-desc)(stk) \text{ in} \\
\text{loop-infinite(seq-stat}(id)(seq-stat^*)(t)(\%p)(id))(c)(v)(stk_1) \end{align*}
\)

(\text{loop-infinite(seq-stat}(id)(seq-stat^*)(t)(p)(c)(v)(stk) = \text{let } c_1 = \lambda v_1.stk_1. \\
\begin{align*}
SS [ seq-stat^* ] (t)(p)(c_2)(v_1)(stk_1) \\
\text{where} \\
c_2 = \lambda v_2.stk_2. \\
\text{loop-infinite(seq-stat}(id)(seq-stat^*)(t)(p)(c)(v_2)(stk_2) \text{ in} \\
(mk-sd(hd(p))(e)(e)(e)(c_1(v)(stk)))
\end{align*}
\)

(SS8) \(SS) SS [ \text{ WHILE } \text{id expr seq-stat}^* \text{ opt-id }] (t)(p)(c)(v)(stk) = \text{let } lp-desc = \langle \text{*LOOP-EXIT*}, \text{id}, \text{p}, \lambda v, s. c(v)(s) \rangle \text{ in}
\begin{align*}
\text{let } stk_1 = \text{stk-push}(lp-desc)(stk) \text{ in} \\
\text{loop-while(seq-stat}(id)(expr)(seq-stat^*)(t)(\%p)(id))(c)(v)(stk_1) \end{align*}
\)

(\text{loop-while(seq-stat}(id)(expr)(seq-stat^*)(t)(p)(c)(v)(stk) = \text{let } c_1 = \lambda v_1.stk_1. \\
\begin{align*}
SS [ seq-stat^* ] (t)(p)(c_2)(v_1)(stk_1) \\
\text{where} \\
c_2 = \lambda v_3.stk_3. \\
\text{loop-while} \\
\begin{align*}
(seq-stat}(id)(expr)(seq-stat^*)(t)(p)(c)(v_2) \\
(stk_2) \text{ in}
\end{align*}
\)
\)

\begin{align*}
\text{mk-sd} \\
(hd(p))(\text{cons}(e,f))(e)(e)(c_1(v_1)(stk_1)), \\
\text{mk-sd} \\
(hd(p))(\text{cons}(\text{mk-not}(e),f))(e)(e) \\
(c(v_1)(\text{stk-pop}(stk_1))))
\end{align*}

(SS9) \(SS) SS [ \text{ FOR } \text{id ref discrete-range seq-stat}^* \text{ opt-id }] (t)(p)(c)(v)(stk) = \text{let } d = T [ \text{ ref }] (t)(p) \text{ in}
\begin{align*}
\text{let } lp-desc = \langle \text{*LOOP-EXIT*}, \text{id}, \text{p}, \lambda v, s. c(v)(s) \rangle \text{ in}
\text{let } stk_2 = \text{stk-push}(lp-desc)(stk) \text{ in} \\
\text{let } (\text{direction}, \text{expr}_1, \text{expr}_2) = \text{discrete-range in} \\
R [ expr_1 ] (t)(p)(k_1)(v)(stk) \\
\text{where}
\end{align*}

139
\[ k_1 = \lambda(c_1,f_1),v_1,stk_1. \]
\[ R \\[ expr_2 \\] (t)(p)(k_2)(v_1)(stk_1) \]
where
\[ k_2 = \lambda(e_2,f_2),v_2,stk_2. \]

let \( bk\)-desc = \*BLOCK-EXIT* ,id,p,\( \lambda v,s.c(v)(s) > \) in

let decl = (DEC,CONST,

(last(hd(hd(tl(ref))))),

(hd(d)),hd(tl(discrete-range))) in

\[ D \{ decl \} (t)(\%(p)(id))(u)(v) \]
\[ (stk-push(bk-desc)(stk_0)) \]
where
\( u = \lambda v_3,stk_3. \)

let \( bg\)-desc = \*BEGIN* ,id,%(p)(id),\( \lambda v,s.c(v)(s) > \) in

\[ \text{let mk-sd} \]
\[ (hd(p))(nconc(f_1,f_2))(e)(e) \]
\[ ((\text{case tag(d)}) \]
\[ ^{\text{*INT*}} \]

\[ \text{let final-iter-val = eval-expr} \]
\[ (e_2) \text{ in} \]

loop-for-int
\[ (seq-stat)(ref)(d) \]
\[ (direction)(final-iter-val) \]
\[ (seq-stat^*)(t)(\%(p)(id))(c_1)(v_3) \]
\[ (stk-push(bg-desc)(stk_3)) \]

\[ ^{\text{*ENUMTYPE*}} \]

\[ \text{let initial-iter-val = eval-expr} \]
\[ (e_1) \]

and final-iter-val = eval-expr
\[ (e_2) \]

and enum-lits = literals(d) in

let parameter-updates = tl(get-loop-enum-param-vals

(initial-iter-val)

(final-iter-val)

(direction)

(enum-lits)) in

loop-for-enum
\[ (seq-stat)(ref)(d) \]
\[ (direction) \]
\[ (parameter-updates) \]
\[ (final-iter-val)(seq-stat^*) \]
\[ (t)(\%(p)(id))(c_1)(v_3) \]
\[ (stk-push(bg-desc)(stk_3)) \]

OTHERWISE

\[ \text{impl-error} \]
\[ ("\text{Illegal FOR loop parameter type: "}, \]
\[ d)) \}) \]

where \( c_2 = \lambda v_4,stk_4. \)

block-exit(\( v_4)(stk_4) \)

\[ \text{loop-for-int}(seq-stat)(ref)(d)(direction)(final-iter-val)(seq-stat^*)(t)(p)(c)(v)(stk) \]
\[ = \text{E} \{ ref \} (t)(p)(k)(v)(stk) \]
where
\[ k = \lambda(e,f),v,stk. \]
\[ \text{E} \{ ref \} (t)(p)(k_1)(v)(stk) \]
where
\[ k_1 = \lambda(e_1,f_1),v_1,stk_1. \]
let \( c_0 = \lambda v_0, stk_0. \)

\[
 \text{SS \[ seq-stat* \] \( (t)(p)(c_1)(v_0)(stk_0) \)
 \text{where}
 c_1 = \lambda v_2, stk_2.
 (mk-sd
 \text{\( (hd(p))(e)(e)((e)) \)\
 \text{\( (cons(mk-rel\}
 \text{\( (d)\}
 \text{\( ((EQ, \text{pound(e)}, \)
 \text{\( (direction = TO \)}
 \text{\( \rightarrow \text{mk-exp2(ADD, e_1, 1}, \)
 \text{\( \text{mk-exp2(SUB, e_1, 1)))}) \}),\)
 \text{loop-for-int
 \text{\( (seq-stat)(ref)(d)(direction)\)\
 \text{\( (final-iter-val)(seq-stat\ast)(t)\)
 \text{\( (p)(c)(v_2)(stk_2)\))) \) in
 (mk-sd\)
 \text{\( (hd(p))\)\
 \text{\( (cons(mk-rel\}
 \text{\( (d)\)
 \text{\( \text{(((direction = TO -> LE, GE, e_1, final-iter-val)), f_1)))(e)(e)(c_0(v)(stk))\),}\)
 \text{mk-sd
 \text{\( (hd(p))\)\
 \text{\( (cons(mk-rel\}
 \text{\( (d)\)
 \text{\( \text{(((direction = TO -> GT, LT, e_1, final-iter-val)), f_1)))(e)(e)\)
 \text{\( (c)(v)(stk-pop(stk))\)))\})\})\}
 \text{where}
 k = \lambda (e,f,v,stk. \)
 R \[ ref \] \( (t)(p)(k_1)(v)(stk) \)
 \text{where}
 k_1 = \lambda (e_1, f_1, v_1, stk_1.
 let \( c_0 = \lambda v_2, stk_0. \)
 \text{SS \[ seq-stat\ast \] \( (t)(p)(c_1)(v_0)(stk_0) \)
 \text{where}
 c_1 = \lambda v_2, stk_2.
 (parameter-updates
 \text{\( \rightarrow (mk-sd\)
 \text{\( (hd(p))(e)(e)((e))\)\
 \text{\( (cons(mk-rel\}
 \text{\( (d)\)
 \text{\( ((EQ, \text{pound(e)}, \)
 \text{\( (hd(\text{parameter-updates})))\}),\)
 \text{loop-for-enum
 \text{\( (seq-stat)(ref)(d)\)\
 \text{\( (direction)\)\
 \text{\( ((parameter-updates)\)\
 \text{\( (final-iter-val)(seq-stat\ast)\)
 \text{\( (t)(p)(c)(v_2)(stk_2)\)))\),\)
 \text{mk-sd\)
 \text{\( (hd(p))(e)(e)(e)\)\
 \text{\( (c)(v)(stk-pop(stk))\))) \) in
 (mk-sd\)
 \text{\( (hd(p))\)\

141
A loop — i.e., a LOOP, WHILE, or FOR statement — has a label (used for leaving that loop by means of an EXIT statement) and a body consisting of sequential statements. When a loop is entered, a new local environment is created (signified by an extended path in the TSE), and a *LOOP-EXIT* descriptor is pushed onto the execution stack, to be used by EXIT statements to leave the loop properly. The continuation in the descriptor is that of the loop statement itself.

In the case of a simple LOOP statement, the loop is nonterminating, and a recursive SD is generated by auxiliary semantic function loop-infinite.

In the case of a WHILE statement, auxiliary semantic function loop-while first processes the control expression, yielding its translation and a guard formula, and then uses these items to generate two SDs, one of which is recursive. The recursive SD represents the situation where the control expression is true and the loop’s body is executed; recursion stems from the appearance of loop-while in the continuation of the loop body’s translation. The execution stack remains unchanged in this case. The other SD represents the case where the loop is exited “naturally” by virtue of its control expression having the value false. The postcondition of this SD is the loop statement’s continuation applied to the result of popping the loop statement’s descriptor from the execution stack.

The case of a FOR statement is analogous to that of the WHILE statement, only more complex technically.
An EXIT statement:

- transfers control from the interior of a loop to the immediate successor of that loop, provided that the EXIT statement's condition (if any) is satisfied; and
- adjusts the state of SDVS to reflect that transfer of control.

The loop being exited can be named in the EXIT statement; Phase 1 has ensured that an appropriate label is used. If a loop is named, then that loop is exited. If no name appears, then the smallest loop enclosing the EXIT statement is exited. The EXIT statement may be enclosed within a system of nested loops. When the loop statement is exited, these other loops must first be exited in the order opposite that in which they were entered. When a FOR loop is exited, the effect of its implicit local declaration of the iteration parameter is reversed by encountering an *UNDECLARE* descriptor on the execution stack.

The translation of an EXIT statement first processes its control expression (which may be empty), resulting in a translated expression and a guard formula. If the control expression is nonempty, two SDs are generated. The first represents the case where the control expression has the value true; in this case the exit process proceeds by invoking the semantic function exit, which appears in the SD's postcondition. The other SD represents the case where the control expression has the value false, whereupon the exit does not occur and control passes to the immediate successor of the EXIT statement. If the control expression is empty, the exit is unconditional; the second SD is not even generated.

(SS11) \[
\begin{align*}
\text{let } & basic-ref = second(ref) \text{ in} \\
& \text{let } expr^* = second(second(basic-ref)) \text{ in} \\
& \text{MR } \left[ expr^* \right] (t)(p)(k)(v)(stk) \\
& \text{where} \\
& k = \lambda (e^*,f^*),v_1,stk_k. \\
& \text{let } ((g,q,\text{id}) = \text{hd(basic-ref)} \text{ in} \\
& \text{let } d = t(q)(\text{id}) \text{ in} \\
& \text{gen-call(ref)(d)(e^*)(f^*)(t)(p)(c)(v)(stk)}
\end{align*}
\]

\[
\begin{align*}
\text{gen-call(ref)(d)(e^*)(f^*)(t)(p)(c)(v)(stk)} \\
& = \text{let } \text{body(d) in} \\
& \text{bind-parameters(ref)(d)(e^*)(f^*)(t)(p)(u)(c)(v)(stk)} \\
& \text{where} \\
& u = \lambda_{v_1,stk_k}. \\
& \text{let } q = \%(\text{path(d))(idf(d))} \text{ in} \\
& \text{let } \text{par-desc} = \text{<UNDECLARE*,collect-pars(extract-pars(d))(ALL ),p,} \\
& \lambda_{v_1,v_k,stk_k}. \\
& \text{unbind-parameters(ref)(d)(e^*)(t)(p)(c)(v)(stk)} \\
& \text{let } sp-desc = \text{<SUBPROGRAM-RETURN*,idf(d),p,\lambda_{v_1,s,c}(v)(s)> in} \\
& \text{let } stk_5 = stk-push(par-desc)(stk-push(sp-desc)(stk_1))
\end{align*}
\]
and \( z = \text{hd}(p) \) in

\[(mk-sd)
\]

\[(z)(e)(e)
\]

\[(\text{cons}((\text{EQ} \cdot \text{pound}(\text{catenate}(z, "\text{pc}")),
\]

\[(\text{AT} \cdot $\text{(path}(d))(\text{idf}(d))))\]

\[u_2(v_1)(\text{stk}_2)\)

where

\[u_2 = \lambda v_2, \text{stk}_2.\]

\[(\text{null}(\text{characterizations}(d))\]

\[\rightarrow D \text{ [ decl ] } (t)(q)(u_2)(v_2)(\text{stk}_2),\]

\[\text{null(seq-stat*)}\]

\[\rightarrow \text{gen-characterizations}\]

\[(e)(p)(\text{characterizations}(d))(c_2)(v_2)(\text{stk}_2)\]

where

\[c_2 = \lambda v_7, \text{stk}_7.\]

\[\text{unbind-parameters}\]

\[\text{(ref}(d)(e^*)(t)(p)(v_7)(\text{stk}_7)\]

where

\[c_3 = \lambda v_8, \text{stk}_8. \text{block-exit}(v_8)(\text{stk}_8),\]

\[\text{impl-error}\]

\[\text{("Offline Characterization not yet implemented")}\]

where

\[u_1 = \lambda v_2, \text{stk}_2.\]

\[
\text{let bg-desc = } <*\text{BEGIN*}, \text{idf}(d), q, \lambda v, s, c_1(vv)(s)> \text{ in } \\
\text{SS [ seq-stat* ] } (t)(q)(c_1)(v_2)(\text{stk-push(bg-desc)(stk}_2)) \\
\text{where } c_1 = \lambda v_8, \text{stk}_8. \text{block-exit}(v_8)(\text{stk}_8) \\
\]

\[\text{bind-parameters(ref}(d)(e^*)(f^*)(t)(p)(u)(v)(\text{stk})\]

\[= \text{(null(extract-pars}(d)) \rightarrow u(v)(\text{stk}),\]

\[\text{let } z = \text{hd}(p) \text{ in} \]

\[\text{let } q = \%\text{(path}(d))(\text{idf}(d)) \text{ in} \]

\[
\text{let all-pars } = \text{get-qids(collect-pars(extract-pars}(d))(\text{ALL })\text{(t)(q) in} \\
\text{let from-pars } = \text{get-qids(collect-pars(extract-pars}(d))(\text{FROM })\text{(t)(q) in} \\
\text{let } v_1 = \text{push-universe}(v)(z)(\text{all-pars) in} \\
\text{let qual-all-pars } = \text{get-qualified-ids(ALL-pars)(v_1) in} \\
\text{let qual-from-pars } = \text{get-qualified-ids(from-pars)(v_1) in} \\
\text{let from-types } = \text{collect-types(extract-pars}(d))(\text{FROM }) \text{ in} \\
\text{let sdcont } = \text{aid, pre, comod, mod, post.} \\
\text{(mk-decl-sd}(d)(\text{pre})(\text{comod})(mod)(post)) \text{ in} \\
\text{mk-exists-already} \\
\text{qual-all-pars)(all-pars)(v) \\
\text{mk-decl-sd} \\
\text{(z)(e)(e)(z)) \\
\text{(nconc} \\
\text{(mk-qual-id-coverings(ALL-pars)(qual-all-pars)(z)(v),} \\
\text{mk-par-decls(q)(extract-pars}(d))(p)(t)(v_2),} \\
\text{(null(qual-from-pars) } \rightarrow u(v_1)(\text{stk)}, \\
\text{sdcont} \\
\text{(z)(f^*)(e)(qual-from-pars)} \\
\text{(nconc} \\
\text{(init-from-pars(qual-from-pars)(from-types)(from-args),} \\
\text{u(v_1)(\text{stk}))}))))) \\
\]

\[\text{extract-pars}(d)\]

\[= \text{let signatures } = \text{signatures}(d) \text{ in} \\
\text{let signature } = \text{hd(signatures) in} \\
\text{(null(tl(signatures)) } \rightarrow \text{pars(signature)}, \\
\text{extract-poly-pars(pars}(signature))))\]
extract-poly-pars(pars)
= (null(pars) → ε,
    let par = hd(pars) in
    cons(((hd(par),(hd(hd(tl(par)))),poly-type-desc())),
         extract-poly-pars(tl(pars))))
collect-pars(par-assoc)(kind)
= (null(par-assoc) → ε,
    let (id,w) = hd(par-assoc) in
    let tm = tmode(w) in
    (kind = ALL V (kind = FROM ∧ ref-mode(tm)∈ (REF VAL)
    → cons(id,collect-pars(tl(par-assoc))(kind)),
    collect-pars(tl(par-assoc))(kind))))
collect-args(actual-args)(par-assoc)(kind)
= (null(actual-args) → ε,
    let arg = hd(actual-args) in
    let (id,w) = hd(par-assoc) in
    let tm = tmode(w) in
    (kind = ALL V (kind = FROM ∧ ref-mode(tm)∈ (REF VAL)
    → cons(arg,collect-args(tl(actual-args))(tl(par-assoc))(kind)),
    collect-args(tl(actual-args))(tl(par-assoc))(kind))))
collect-types(par-assoc)(kind)
= (null(par-assoc) → ε,
    let (id,w) = hd(par-assoc) in
    let tm = tmode(w) in
    (kind = ALL V (kind = FROM ∧ ref-mode(tm)∈ (REF VAL)
    → cons(tdesc(w),collect-types(tl(par-assoc))(kind)),
    collect-types(tl(par-assoc))(kind))))
mk-par-decls(q)(par-assoc)(p)(t)(v)
= (null(par-assoc) → ε,
    let (id,w) = hd(par-assoc) in
    cons((DECLARE ,qualified-id(qid(t(q)(id)))(v),mk-type-spec(tdesc(w))(t)(p)),
         mk-par-decls(q)(tl(par-assoc))(p)(t)(v)))
init-from-pars(from-pars)(from-types)(expr*)
= (null(from-pars) → ε,
    let dst = hd(from-pars) and d = hd(from-types) and src = hd(expr*) in
    nconc
    (assign(d)((dst,src)),
     init-from-pars(t1(from-pars))(tl(from-types))(tl(expr*))))
gen-characterizations(sds)(p)(characterizations)(c)(v)(stk;)
= (null(characterizations) → fix-characterized-sds(sds)(c)(v)(stk;),
    let (q,id,parnames,pre,mod) = hd(characterizations) in
    let post = sixth(hd(characterizations)) in
    gen-characterizations
    (cons(gen-characterization(hd(p))(§(q)(id))(parnames)(pre)(mod)(post)(v),sds)
     (p)(tl(characterizations))(c)(v)(stk;))
gen-characterization(z)(qid)(parnames)(pre)(mod)(post)(v)
= let sd = mk-sd
   (z)(((EQ ,dot(catenate(z,“\pc”))),(AT ,qid)))(ε)(mod)
   (append
    (post,((EQ ,pound.(catenate(z,“\pc”))),(EXITED ,qid)))) in
   subst-vars(parnames)(v)(sd)
unbind-parameters\((\text{ref}(d))((\text{actual-args})(t)(p)(c)(v))(\text{stk})\)

\[
\begin{align*}
\text{let } z &= \text{hd}(p) \text{ in} \\
&\text{let } q = (\%((\text{path}(d))((\text{idf}(d)))) \text{ in} \\
&\text{let } \text{all-pars} = \text{get-qids}((\text{collect-pars})(\text{extract-pars}(d))(\text{ALL}))(t)(q) \text{ in} \\
&\text{let } \text{to-pars} = \text{get-qids}((\text{collect-pars})(\text{extract-pars}(d))(\text{TO}))(t)(q) \text{ in} \\
&\text{let } \text{qual-all-pars} = \text{get-qualified-ids}(\text{all-pars})(v) \\
&\text{and } \text{qual-to-pars} = \text{get-qualified-ids}(\text{to-pars})(v) \text{ in} \\
&\text{let } \text{actual-names} = \text{collect-args}(\text{actual-args})(\text{extract-pars}(d))(\text{TO}) \text{ in} \\
&\text{let } \text{to-args} = \text{underef}(\text{actual-names}) \text{ in} \\
&(\text{mk-sd}) \\
&(z)(e)(e)(e) \\
&(\text{let } u = \lambda a,b,c,d,e,f,g,h,i,j. \\
&\quad (\text{unbind-parameters-sds}(a)(b)(c)(d)(e)(f)(g)(h)(i)(j)) \text{ in} \\
&\quad \text{cons}(\text{EQ},\text{pound}(\text{catenate}(z,\text{"pc"})), \\
&\quad \text{(EXITED},$((\text{path}(d))(\text{idf}(d))))), \\
&\quad u(z)(\text{ref}(d)(\text{all-pars})(\text{qual-all-pars})(\text{qual-to-pars})(\text{to-args})(c)(v)(\text{stk}))) \\
\end{align*}
\]

\text{underef}(\text{actual-args})
\[
= (\text{null}(\text{actual-args}) \rightarrow z, \\
\text{let } \text{actarg} = \text{hd}(\text{actual-args}) \text{ in} \\
\text{cons}(\text{second}(\text{actarg}),\text{underef}(\text{tl}(\text{actual-args}))))
\]

unbind-parameters-sds\((z)(\text{ref}(d)(\text{all-pars})(\text{qual-all-pars})(\text{qual-to-pars})(\text{to-args})(c)(v)(\text{stk})\)
\[
= (\text{null}(\text{qual-all-pars}) \\
\rightarrow \text{mk-sd} \\
&\quad (z)(e)(e)(e) \\
&\quad (\text{cons}(\text{pop-universe}(v)(\text{all-pars}))(\text{stk-pop}(\text{stk})))), \\
(\text{null}(\text{to-args}) \\
\rightarrow \text{mk-sd} \\
&\quad (z)(e)(e)(\text{cons}(z,\text{qual-all-pars})) \\
&\quad (\text{cons}(\text{mk-cover-already}(\text{dot}(z),\text{cons}(\text{pound}(z),\text{qual-all-pars})), \\
&\quad \text{cons}(\text{mk-undeclare}(\text{qual-all-pars}), \\
&\quad \text{c}(\text{pop-universe}(v)(\text{all-pars}))(\text{stk-pop}(\text{stk})))))), \\
\text{let } u = \lambda id,\text{pre},\text{comod},\text{mod},\text{post}. (\text{mk-sd}(id)(\text{pre})(\text{comod})(\text{mod})(\text{post})) \text{ in} \\
\text{mk-sd} \\
&\quad (z)(e)(e)(\text{to-args}) \\
&\quad (\text{let } \text{to-types} = \text{collect-types}(\text{extract-pars}(d))(\text{TO}) \text{ in} \\
&\quad \text{nconc} \\
&\quad \quad (\text{assign-to-args}(\text{to-args})(\text{to-types})(\text{qual-to-pars}), \\
&\quad \quad u(z)(e)(e)(\text{cons}(z,\text{qual-all-pars})) \\
&\quad \quad (\text{cons}(\text{mk-cover-already}(\text{dot}(z),\text{cons}(\text{pound}(z),\text{qual-all-pars})), \\
&\quad \quad \text{cons}(\text{mk-undeclare}(\text{qual-all-pars}), \\
&\quad \quad \text{c}(\text{pop-universe}(v)(\text{all-pars}))(\text{stk-pop}(\text{stk})))))), \\
&\quad \text{let } u = \lambda id,\text{pre},\text{comod},\text{mod},\text{post}. (\text{mk-sd}(id)(\text{pre})(\text{comod})(\text{mod})(\text{post})) \text{ in} \\
&\quad \text{mk-sd} \\
&\quad (z)(e)(e)(\text{to-args}) \\
&\quad (\text{let } \text{to-types} = \text{collect-types}(\text{extract-pars}(d))(\text{TO}) \text{ in} \\
&\quad \text{nconc} \\
&\quad \quad (\text{assign}(d)(((\text{dst},\text{dot}(\text{src}))), \\
&\quad \quad \text{assign-to-args}(\text{tl}(\text{to-args}))(\text{tl}(\text{to-pars}))))(\text{tl}(\text{to-pars}))))
\]

\text{mk-cover-already}(id,lst)
\[
= (\text{new-declarations()} \rightarrow \text{mk-rel}(\text{int-type-desc()})(\text{EQ},\text{hd}(\text{lst}),\text{id}), \\
\text{mk-cover}(\text{id},\text{lst}))
\]

\text{mk-undeclare}(\text{lst}) = \text{cons}(\text{UNDECLARE},\text{lst})

\text{assign-to-args}(\text{to-args})(\text{to-types})(\text{to-pars})
\[
= (\text{null}(\text{to-args}) \rightarrow z, \\
\text{let } \text{dst} = \text{hd}(\text{to-args}) \\
\text{and } d = \text{hd}(\text{to-types}) \\
\text{and } \text{src} = \text{hd}(\text{to-pars}) \text{ in} \\
\text{nconc} \\
\quad (\text{assign}(d)(((\text{dst},\text{dot}(\text{src}))), \\
\quad \text{assign-to-args}(\text{tl}(\text{to-args}))(\text{tl}(\text{to-pars}))))(\text{tl}(\text{to-pars})))
\]
Procedure calls in Stage 2 VHDL use call by value-result semantics. The translation of a procedure call consists of the following steps:

- The actual parameters are translated and then gen-call pushes a subprogram return descriptor and then a (single) undeclaration descriptor for all of the formal parameters onto the execution stack.

- SDVS declarations of all of the formal parameters are emitted (in bind-parameters).

- The IN and INOUT formal parameters are bound to their corresponding actual parameters by first translating the actual parameters and then in effect assigning them to their corresponding formals by emitting appropriate equality relations (as in the translation of assignment). This is done by auxiliary semantic function bind-parameters. In these equality relations, the qualified names of the formal parameters must refer to the procedure's declaration TSE, whereas the qualified names in the actual parameters refer to the procedure's calling environment. This implements the semantics of static binding required by VHDL.

- The subprogram may have either a specific body or a set of state delta characterizations, but not both. Different actions are performed in each case.

1. If the procedure has a body, the procedure's local declarations and statements are translated in the procedure's declaration environment after first pushing a *SUBPROGRAM-RETURN* descriptor on the execution stack. This descriptor will be used to perform a return from the procedure, whether that return is explicit via a RETURN statement or implicit via encountering the end of the procedure's body.

2. If the procedure has one or more characterizations, state deltas representing the actions of the procedure are produced by the functions gen-characterizations and gen-characterization. These two functions use the SDVS functions fixed-characterized-sds and subst-vars, part of the implementation of an offline characterization mechanism for SDVS [16, 17].

- Auxiliary semantic function unbind-parameters is invoked to assign the (final) values of the INOUT and OUT formal parameters to their corresponding actual parameters (which must, of course, have reference types).

(SS12) \[ \text{RETURN opt-expr } (t)(p)(c)(v)(stk) \]
\[ = \text{let expr = opt-expr in} \]
\[ R [ expr ] (t)(p)(k)(v)(stk) \]
where
\[ k = \lambda(e,f,v_1,stk_1). \]
\[ (\text{null}(e) \rightarrow \text{return}(v_1)(stk_1)), \]
\[ \text{let } d = \text{context}(t)(p) \text{ in} \]
\[ (\text{mk-sd} \]
\[ (\text{hd}(p))(f)(e)((\text{qid}(d))) \]
\[ (\text{nconc} \]

\text{3None such are allowed, as yet, by Stage 2 VHDL.}
RETURN statements come in two varieties: with an expression, to effect a return from a function, and without an expression, to effect a return from a procedure. If the RETURN is from a function, then the expression must first be translated and an assignment of its value to the function’s (statically and dynamically uniquely qualified) name must be asserted via an equality relation. Then (no matter whether the RETURN is from a procedure or a function), the function return (similar to exit) is invoked to use the topmost *SUBPROGRAM-RETURN* descriptor on the execution stack to return from the subprogram, after first effecting exits from intervening loops and effecting necessary undeclarations. The function context determines the qualified name of the subprogram from which the return is being made.

(SS13) \[ \text{RETURN statements come in two varieties: with an expression, to effect a return from a function, and without an expression, to effect a return from a procedure. If the RETURN is from a function, then the expression must first be translated and an assignment of its value to the function’s (statically and dynamically uniquely qualified) name must be asserted via an equality relation. Then (no matter whether the RETURN is from a procedure or a function), the function return (similar to exit) is invoked to use the topmost *SUBPROGRAM-RETURN* descriptor on the execution stack to return from the subprogram, after first effecting exits from intervening loops and effecting necessary undeclarations. The function context determines the qualified name of the subprogram from which the return is being made.} \]
(t)(p)) in

(t)(p)) in

(find-process-env(t)(p)
= (null(p)v tag(t(p)(*UNIT* ))) = *PROCESS* → p, find-process-env(t)(rest(p)))

get-signals(signal-names)
= (null(signal-names)→ ε,
  cons(find-signal-structure(hd(signal-names)),get-signals(tl(signal-names)))))

8.4.7 Waveforms and Transactions

(W1) W[ WAVE transaction+ ] (t)(p)(wave-cont)(v)(stk)
     = TRM [ transaction+ ] (t)(p)(wave-cont)(v)(stk)

(TRM0) TRM [ ε ] (t)(p)(wave-cont)(v)(stk) = wave-cont((ε,ε))(v)(stk)

(TRM1) TRM [ transaction transaction* ] (t)(p)(wave-cont)(v)(stk)
     = TR [ transaction ] (t)(p)(trans-cont)(v)(stk)
       where
       trans-cont = λ(trans,guard),v,stk.
       TRM [ transaction* ] (t)(p)(wave-cont1)(v)(stk)
       where
       wave-cont1 = λ(trans*,guard*),v,stk.
       wave-cont
       (cons(trans,trans*),
        nconc(guard,guard*)))(v)(stk)

The transactions in a waveform are translated in order, from left to right.

(TR1) TR [ TRANS expr opt-expr ] (t)(p)(trans-cont)(v)(stk)
     = R [ expr ] (t)(p)(k)(v)(stk)
       where
       k = λ(e1,f1),v,stk.
       let expr2 = opt-expr in
       R [ expr ] (t)(p)(k1)(v)(stk)
       where
       k1 = λ(e2,f2),v,stk.
       trans-cont
       ((mk-transaction-for-update(e1)(e2),nconc(f1,f2)))
       (v)(stk)

mk-transaction-for-update(transaction-value)(delay-time)
= let transaction-time = (null(delay-time)→ mk-add-delay-time(0)(1),
                        mk-add-delay-time(delay-time)(0)) in
     mk-transaction(transaction-time)(transaction-value)

mk-add-delay-time(global)(delta)
= (TIMEPLUS , dot(VHDLTIME ), mk-vhdltime(global)(delta))
mk-vhdltime(global)(delta) = (VHDLTIME , global, delta)
8.4.8 Expressions

Two semantic functions, \( E \) and \( R \), translate expressions. \( E \) obtains the (qualified) place name corresponding to a scalar or array. \( R \) yields an expression that represents a value rather than a reference.

\[ ME \ll e \rr (t)(p)(h)(v)(stk) = h((e,e))(v)(stk) \]

\[ ME \ll ref ref^* \rr (t)(p)(h)(v)(stk) = E \ll ref \rr (t)(p)(k)(v)(stk) \]

where
\[ k = \lambda(e,f),v_1,stk_1. \]

\[ ME \ll ref \rr (t)(p)(h_1)(v_1)(stk_1) \]

where \( h_1 = \lambda(e^*,f^*),v_2,stk_2.h((cons(e,e^*),nconc(f,f^*))) \)

\[ R \ll dj \rr expr\rr (t)(p)(k)(v)(stk) \]

where
\[ k = \lambda(e,f),v_1,stk_1. \]

\[ R \ll expr \rr (t)(p)(h_1)(v_1)(stk_1) \]

where \( h_1 = \lambda(e^*,f^*),v_2,stk_2.h((cons(e,e^*),nconc(f,f^*))) \)

The translation of a (possibly empty) multiple expression list yields a list of translated expressions and a corresponding list of guard formulas.

\[ ME \ll REF modifier^+ \rr (t)(p)(k)(v)(stk) \]

\[ R \ll REF \rr expr\rr (t)(p)(k)(v)(stk) \]

where
\[ k = \lambda(e,f),v_1,stk_1. \]

\[ R \ll expr \rr (t)(p)(h_1)(v_1)(stk_1) \]

where \( h_1 = \lambda(e^*,f^*),v_2,stk_2.h((cons(e,e^*),nconc(f,f^*))) \)

The translation of a multiple expression list yields a list of translated expressions and a corresponding list of guard formulas.
= R \[ expr \] (t)(p)(k)(v)(stk)
where
k = \lambda(e_0,f_0),v_0,stk_0.
cont(((ELEMENT,e,e_0),
nconc
(f,f_0,
(null(ub(d))
→ (mk-rel(int-type-desc())((GE,e_0,(ORIGIN,e))),
(mk-rel(int-type-desc())((GE,e_0,(ORIGIN,e))),
mk-rel
(int-type-desc()))
((LE,e_0,
 mk-exp2
 (SUB,mk-exp2(ADD,(ORIGIN,e),(RANGE,e)),1))))),
 elty(d)))(v_0)(stk)
gen-record-ref(id)(e)(f)(d)(cont)(v)(stk)
= cont((mk-recelt(e,id),f~lookup-record-desc(components(d))(id)))(v)(stk)
mk-recelt(e)(id) = (RECORD,e,id)
lookup-record-desc(comp')(id)
= (null(comp')→ *UNBOUND*,
 let (x,d) = hd(comp') in
 (x = id → d, lookup-record-desc(tl(comp'))(id)))
gen-function-call(ref)(expr*)(d)(t)(p)(cont)(v)(stk)
= declare-function-name(d)(t)(p)(u)(v)(stk)
where
u = AV3,stk3.
MR \[ expr \] (t)(p)(h)(v_3)(stk_3)
where
h = \lambda(e^*,f^*),v_1,stk_1.
gen-call(ref)(d)(e^*)(f^*)(t)(p)(c)(v_1)(stk_1)
where
c = \lambda v_2,stk_2.
cont((qualified-id(qid(d))(v_2),c,
tdesc(extract-rtype(d)))(v_2)(stk_2))
declare-function-name(d)(t)(p)(u)(v)(stk)
= let dd = tdesc(extract-rtype(d)) in
 let q = path(d) in
 let z = hd(q) in
 let suqn+ = get-unique((tidf(d)))(t)(q) in
 let v_1 = push-universe(v)(z)(suqn+) in
 let duqn+ = get-qualified-ids(suqn+)(v_1) in
 let dc-desc = <*UNDECLARE*,idf(d),q,
 λv_1,v_2,stk_2.
 undeclare-function-name(suqn+)(duqn+)(z)(u_1)(v_2)(stk_2)> in
 (mk-exists-already
 (duqn+)(suqn+)(v)
 (mk-decl-sd
 (z)(c)(e)(z))
 (nconc
 (mk-qual-id-coverings(suqn+)(duqn+)(z)(v),
 mk-scalar-monsignal-dec-post
 (c)((duqn+,_dd))(t)(q)(u_1)(v_1)(stk-push(dc-desc)(stk))))))
undeclare-function-name(suqns)(duqns)(z)(u)(v)(stk)
= (mk-sd
  (z)(e)(cons(z,duqns))
  (cons(mk-cover-already(dot(z))(cons(pound(z),duqns)),
   cons(mk-undeclare(duqns),
    u(pop-universe(v)(suqns))(stk-pop(stk))))))

A reference must begin with at least a basic reference, which contains its root identifier
and access path. Following its basic reference, a reference has zero or more array index,
record field selection, or actual parameter list modifiers. The reference itself is translated
by gen-name; the basic reference is translated by gen-basic-name. The array index and
record field selection modifiers are translated by gen-array-ref and gen-record-ref. The
translation of a reference is complicated by the appearance of a parameter list modifier,
which represents a function call; these are translated by gen-function-call.

Whenever a function is called (as part of an expression), the name of that function is used
in the expression to name the value returned by that particular invocation. Because the
same function can be invoked more than once in the same expression, each corresponding
instance of the function’s name must be uniquely dynamically qualified, and each of those
DUQNs must be declared (and later undeclared when they should no longer exist) to SDVS.
The declaration is performed by function declare-function-name and the undeclaration
by undeclare-function-name; the invocation of the latter function is encapsulated in an
undeclaration (*UNDECLARE*) descriptor pushed onto the execution stack. After a
new dynamic instance of the function’s name is declared, gen-function-call evaluates the
actual parameters and then invokes gen-call to finish the translation of this function call.

(R0) R [ ε ] (t)(p)(k)(v)(stk) = k((ε,ε))(v)(stk)

For technical convenience, expressions can be empty; the translation of an empty expression
yields empty results.

(R1) R [ FALSE ] (t)(p)(k)(v)(stk) = k((FALSE,ε))(v)(stk)

(R2) R [ TRUE ] (t)(p)(k)(v)(stk) = k((TRUE,ε))(v)(stk)

(R3) R [ BIT ] bitlit (t)(p)(k)(v)(stk) = k((BIT [ bitlit ] ,ε))(v)(stk)

(R4) R [ NUM constant ] (t)(p)(k)(v)(stk) = k((NUM [ constant ] ,ε))(v)(stk)

(R5) R [ TIME constant FS ] (t)(p)(k)(v)(stk) = k((TIME [ constant ] ,ε))(v)(stk)

(R6) R [ CHAR constant ] (t)(p)(k)(v)(stk) = k((expr,ε))(v)(stk)

(R7) R [ ENUMLIT id ] (t)(p)(k)(v)(stk) = k((id,ε))(v)(stk)
Scalar and array references are first E-translated, yielding an expression and a guard formula. The corresponding R-translation is obtained by applying the dot operation to the translated expression.

(R8) $R \text{ [ } \text{BITSTR} \text{ ] bit-lit } * \text{ [ } t \text{][(p)(k)(v)(stk)}$

$= \text{ let expr* = bit-lit* in}$

$MR \text{ [ } \text{expr* } \text{ ] (t)(p)(k)(v)(stk)}$

(R9) $R \text{ [ } \text{STR} \text{ char-lit } * \text{ [ } t \text{][(p)(k)(v)(stk)}$

$= \text{ let expr* = char-lit* in}$

$MR \text{ [ } \text{expr* } \text{ ] (t)(p)(k)(v)(stk)}$

(R10) $R \text{ [ } \text{REF modifier} + \text{ [ ] (t)(p)(k)(v)(stk)}$

$= \text{ let ref = expr in}$

$E \text{ [ ref ] (t)(p)(k)(v)(stk)}$

$\text{ where } k_1 = \lambda(c,f,v_1,stk_1). k((\text{dot}(c,f))(v_1))(stk_1)$

(R11) $R \text{ [ } \text{PAGRRI expr* } \text{ ] (t)(p)(k)(v)(stk)} = MR \text{ [ } \text{expr* } \text{ ] (t)(p)(k)(v)(stk)}$

(R12) $R \text{ [ } \text{unary-op expr } \text{ ] (t)(p)(k)(v)(stk)}$

$= R \text{ [ } \text{expr } \text{ ] (t)(p)(k)(v)(stk)}$

$\text{ where } k_1 = \lambda(e,f,v_1,stk_1). k((\text{mk-exp1}(\text{unary-op},e))(v_1))(stk_1)$

$\text{ mk-exp1(unary-op,e) }$

$= \text{ (case unary-op}$

$\text{ NOT } \rightarrow (\text{ NOT },e),$

$\text{ BNOT } \rightarrow (\text{ USNOT },e),$

$\text{ PLUS } \rightarrow e,$

$\text{ NEG } \rightarrow (\text{ MINUS },e),$

$\text{ ABS } \rightarrow (\text{ ABS },e),$

$(\text{ RNEG, RABS }) \rightarrow (\text{ unary-op},e),$

$\text{ OTHERWISE}$

$\rightarrow \text{ impl-error("Unrecognized Stage 2 VHDL unary operator: "a",unary-op)}$

(R13) $R \text{ [ } \text{binary-op expr1 expr2 } \text{ ] (t)(p)(k)(v)(stk)}$

$= R \text{ [ } \text{expr1 } \text{ ] (t)(p)(k)(v)(stk)}$

$\text{ where }$

$k_1 = \lambda(e_1,f_1,v_1,stk_1).$

$R \text{ [ } \text{expr2 } \text{ ] (t)(p)(k)(v)(stk)}$

$\text{ where }$

$k_2 = \lambda(e_2,f_2,v_2,stk_2).$

$k((\text{mk-exp2(binary-op,e_1,e_2),nconc(f_1,f_2)}))(v_2)(stk_2)$

(R14) $R \text{ [ } \text{relational-op expr1 expr2 } \text{ ] (t)(p)(k)(v)(stk)}$

$= R \text{ [ } \text{expr1 } \text{ ] (t)(p)(k)(v)(stk)}$

$\text{ where }$

$k_1 = \lambda(e_1,f_1,v_1,stk_1).$

$R \text{ [ } \text{expr2 } \text{ ] (t)(p)(k)(v)(stk)}$

$\text{ where }$

$k_2 = \lambda(e_2,f_2,v_2,stk_2).$

$\text{ let } d = T \text{ [ } \text{expr1 } \text{ ] (t)(p) in}$

$k((\text{mk-rel}(d)((\text{relational-op},e_1,e_2)),nconc(f_1,f_2)))(v_2)(stk_2)$
8.4.9 Expression Types

The function \texttt{mk-rel} (described earlier) requires a type descriptor as its first argument; application of the semantic function \texttt{T} determines the type descriptor of an expression as follows:

- if the expression is a constant, its type descriptor is the basic type of that constant;
- if the expression is a reference, its type descriptor is the basic type of that reference, obtained by the function \texttt{get-type-desc}; and
- if the expression contains operators, its type descriptor is the basic result type of its top-level operator (if there is one);

\[
\begin{align*}
(T0) & \quad \texttt{T [ ε ] } (t)(p) = \text{void-type-desc()} \\
(T1) & \quad \texttt{T [ FALSE ] } (t)(p) = \text{bool-type-desc()} \\
(T2) & \quad \texttt{T [ TRUE ] } (t)(p) = \text{bool-type-desc()} \\
(T3) & \quad \texttt{T [ BIT bitlit ] } (t)(p) = \text{bit-type-desc()} \\
(T4) & \quad \texttt{T [ NUM constant ] } (t)(p) = \text{int-type-desc()} \\
(T5) & \quad \texttt{T [ TIME constant FS ] } (t)(p) = \text{time-type-desc()} \\
(T6) & \quad \texttt{T [ CHAR constant ] } (t)(p) = \text{char-type-desc(t)} \\
(T7) & \quad \texttt{T [ ENUMLIT id ] } (t)(p) \\
& \quad \quad = \text{let } d = \text{lookup-desc(t)(p)(id) in} \\
& \quad \quad \quad \text{tdesc(type(d))} \\
(T8) & \quad \texttt{T [ BITSTR bit-lit* ] } (t)(p) = \text{bitvector-type-desc()}
\end{align*}
\]

\[
\begin{align*}
(T9) & \quad \texttt{T [ STR char-lit* ] } (t)(p) = \text{string-type-desc(t)} \\
(T10) & \quad \texttt{T [ REF modifier+ ] } (t)(p) \\
& \quad \quad = \text{let } \text{basic-ref} = \text{modifier+ in} \\
& \quad \quad \quad \text{get-type-desc(basic-ref)(t)(p)} \\
\end{align*}
\]

\[
\begin{align*}
\text{get-type-desc(basic-ref)(t)(p)} \\
& \quad = \text{let } (t,g,q,id) = \text{hd(basic-ref) in} \\
& \quad \quad \quad \text{let } d = t(q)(id) \text{ in} \\
& \quad \quad \quad \quad \text{(case tag(d)} \\
& \quad \quad \quad \quad \quad \text{(**PROCEDURE*, **FUNCTION*, **PROCESS**)} \\
& \quad \quad \quad \quad \text{← process-ref-tail(d)(tl(basic-ref))(t)(p),} \\
& \quad \quad \quad \quad \text{OTHERWISE ← process-ref-tail(tdesc(type(d)))(tl(basic-ref))(t)(p))}
\end{align*}
\]
process-ref-tail(d)(ref-tail)(t)(p) = (null(ref-tail) → d, let modifier = hd(ref-tail) in (case hd(modifier) INDEX → process-ref-tail(elt(d))(tl(ref-tail))(t)(p), SELECTOR → process-ref-tail (lookup-record-desc(components(d))(second(modifier)))(tl(ref-tail))(t)(p), PARLIST → process-ref-tail(tdesc(extract-rtype(d)))(tl(ref-tail))(t)(p), OTHERWISE → impl-error ("Unrecognized Stage 2 VHDL reference modifier tag: "a", hd(modifier)))(T11)

T [[ PAGGR expr* ]] (t)(p) = void-type-desc()

(T12) T [[ unary-op expr ]] (t)(p) = tdesc(restype1(unary-op)) restype1(unary-op) = (case unary-op NOT → (VAL ,bool-type-desc()), BNOT → (VAL ,bit-type-desc()), (PLUS ,NEG ,ABS ) → (VAL ,int-type-desc()), (RNEG ,RABS ) → (VAL ,real-type-desc()), OTHERWISE → impl-error("Unrecognized Stage 2 VHDL unary operator: "a",unary-op))

(T13) T [[ binary-op expr1, expr2 ]] (t)(p) = tdesc(restype2(binary-op)((expr1 ,expr2 ))(t)(p)) restype2(binary-op)(expr1 ,expr2 )(t)(p) = (case binary-op (AND ,NAND ,OR ,NOR ,XOR ) → mk-type((DUMMY VAL ))(bool-type-desc()), (BAND ,BNAND ,BOR ,BNOR ,BXOR ) → mk-type((DUMMY VAL ))(bit-type-desc()), (ADD ,SUB ,MUL ,DIV ,MOD ,REM ,EXP ) → mk-type((DUMMY VAL ))(int-type-desc()), (RPLUS ,RMINUS ,RTIMES ,RDIV ,REXPT ) → mk-type((DUMMY VAL ))(real-type-desc()), CONCAT → let d1 = T [[ expr1 ]] (t)(p) and d2 = T [[ expr2 ]] (t)(p) in mk-type((DUMMY VAL ))(mk-concat-tdesc(d1 )(d2 )), OTHERWISE → impl-error("Unrecognized Stage 2 VHDL binary operator: "a",binary-op))

mk-concat-tdesc(d1 )(d2 ) = (is-bit-tdesc?(d1 )V is-bitvector-tdesc?(d1 ) → array-type-desc (new-array-type-name(BIT VECTOR ))(e)(e)(tt)(direction(d1 ))(lb(d1 ))(e) (bit-type-desc()), let idf1 = idf(d1 ) in array-type-desc (new-array-type-name((consp(idf1 )→ hd(idf1 ), idf1 ))(e)(e)(tt) (direction(d1 ))(lb(d1 ))(e)(eltty(d1 )))

(T14) T [[ relational-op expr1 expr2 ]] (t)(p) = bool-type-desc()
8.4.10 Primitive Semantic Equations

The following semantic functions are primitive.

(N1) \( N \ [\text{constant}] = \text{constant} \)

(B1) \( B \ [\text{bitlit}] = \text{mk-bit-simp-symbol}(\text{bitlit}) \)

\[ \text{mk-bit-simp-symbol}(\text{bitlit}) = \text{(case bitlit} \]
\[ 0 \rightarrow (\text{BS 0 1}), \]
\[ 1 \rightarrow (\text{BS 1 1}), \]
\[ \text{OTHERWISE} \rightarrow \text{impl-error("Can't construct simp symbol for bit: "}, \text{bitlit})) \]
9 Conclusion

A precise and well-documented formal specification of the Stage 2 VHDL translator has been presented in this report. We have completed and exercised a Common Lisp implementation of both translation phases described herein.

Stage 2 VHDL represents a robust behavioral subset of the VHSIC Hardware Description Language, extending Stage 1 VHDL with the addition of the following VHDL language features: (restricted) design files, declarative parts in entity declarations, package STANDARD (containing predefined types BOOLEAN, BIT, INTEGER, TIME, CHARACTER, REAL, STRING, and BIT_VECTOR), user-defined packages, USE clauses, array type declarations, certain predefined attributes, enumeration types, subprograms (procedures and functions, excluding parameters of object class SIGNAL), concurrent signal assignment statements, FOR loops, octal and hexadecimal representations of bitstrings, ports of default object class SIGNAL, and general expressions of type TIME in AFTER clauses.

As the SDVS interface to VHDL continues to expand and mature, our confidence grows in our language translator semantic specification and implementation paradigm. In 1993, we will be applying this paradigm to implement a translator for the Stage 3 VHDL language subset. Stage 3 VHDL is expected to include constructs for structural descriptions (e.g., component declarations, component instantiation statements, and configuration declarations).

Furthermore, SDVS will be enhanced with proof capabilities enabling both more general specifications and more tractable proofs of VHDL hardware descriptions. Two enhancements of particular importance will be the ability to translate structural descriptions, and the ability to reason about symbolic representations of VHDL time.
References


Index

access 68
access 116
already-qualified-id 117
arl 114
argtypes 82
argtypes-error 83
argtypes-error 84
array-signal-assignment 135
array-size 56
array-type 81
array-type-desc 41
arl 51
arl 89
assign 119, 133
assign-array-downto 120
assign-array-to 120
assign-record 120
assign-to-args 146
at 77
at 77
at2 77
at3 78
attributes-low-high 60
attributes-low-high 130
ax0 93
ax1 93
ax2 93
ax3 93
b1 86
b1 155
bind-parameters 144
bit-type-desc 40
bits-op 96
bitvector-type-desc 41
block-exit 113
body 43
bool-type-desc 40
cascade-array-signal-assignment 135
case-coverage 72
case-overlap 78
case-type-ok 72
case-union 78
char-type-desc 41
characterizations 43
check-array-aggregate 58
check-enum-lits 59
check-exprs 59
check-if 71
check-pkg-names 65
check-wait-ref 77
check-wait-refs 76
chk-array-type 81
collect-args 144
collect-expressions-from-conditional-waveforms 69
collect-expressions-from-selected-waveforms 67
collect-fids 62
collect-formal-pars 63
collect-pars 144
collect-signals-from-expr 68
collect-signals-from-expr-list 68
collect-transaction-expressions 68
collect-types 144
compatible-par-types 55
compatible-signatures 55
components 43
concatenate-bits 125
concatenate-characters 125
construct-case-alternatives 68
construct-cond-parts 69
context 76, 147
cs 131
cs 131
cs 131
cst 67
cst 67
cst 67
cst 67
csx 91
csx 91
csx 91
csx 91
d0 115
d1 115
d10 131
d11 131
d12 131
d13 131
d14 131
d2 115
d3 115
d4 115
d5 115
d6 116
d7 130
d8 130
d9 131

declare-function-name 151
dfl 109
dftl 49
dx1 89
direction 43
dollar 12
dot 107
drt0 78
drt1 78
drt2 78
drx0 93
drx1 93
drx2 93
dt0 52
dt1 52
dt10 61
dt11 61
dt12 62
dt13 63
dt14 64
dt2 52
dt3 52
dt4 52
dt5 52
dt6 57
dt7 59
dt8 60
dt9 60
dx0 90
dx1 90
dx10 90

dx11 90
dx12 90
dx13 90
dx14 90
dx2 90
dx3 90
dx4 90
dx5 90
dx6 90
dx7 90
dx8 90
dx9 90
e1 150
e1ty 43
e11 114
ent1 50
enter-array-objects 58
enter-characters 50
enter-formal-pars 62
enter-objects 50
enter-package 49
enter-predefined 49
enter-standard 49
enter-string 50
enum-le 86
enum-lt 86
enx1 89
et0 80
et1 80
et10 82
et11 82
et12 82
et13 82
et2 80
et3 80
et4 80
et5 81
et6 81
et7 81
et8 81
et9 81
eval-expr 107
ex0 93
ex1 93
ex10 95
ex11 95

162
ex12 95
ex13 95
ex2 93
ex3 93
ex4 93
ex5 94
ex6 94
ex7 94
ex8 94
ex9 94
exit 142
export-qualified-names 66
exported 43
extract-par-types 55
extract-pars 144
extract-poly-pars 144
extract-rtype 55, 148
extract-rtypes 76
fifth 14
filter-components 56
find-architecture-env 67
find-looplabel-env 74
find-process-env 70, 148
find-progunit-env 53
find-signal-structure 106
fixed-characterized-sds 107
fourth 14
gen-alt 137
gen-array-decl 122
gen-array-decl-id* 123
gen-array-decl-id+ 122
gen-array-nonsignal-decl-id+ 123
gen-array-ref 150
gen-array-signal-decl-id+ 126
gen-ascending-indices 135
gen-basic-name 150
gen-call 143
gen-case 137
gen-characterization 145
gen-characterizations 145
gen-characters 50
gen-descending-indices 135
gen-function-call 151
gen-guard 138
gen-if 136
gen-name 150

get-loop-enum-param-vals 107
get-qids 116
get-qualified-ids 103, 116
get-signals 148
get-type-desc 154
idf 43
import-legal 65
import-qualified-names 65
init-array-signal-down-to 107
init-array-signal-to 107
init-from-pars 145
init-scalar-signal 106
init-var 103
int-type-desc 40
invert-bit 83
is-array-tdesc? 42
is-array? 42
is-binary-op? 43
is-bit-tdesc? 41
is-bit? 41
is-bitvector-tdesc? 42
is-bitvector? 42
is-boolean-tdesc? 41
is-boolean? 41
is-character-tdesc? 42
is-character? 42
is-const? 42, 43
is-constant-bitvector? 111, 119
is-constant-string? 111, 119
is-constant-string? 119
is-integer-tdesc? 41
is-integer? 41
is-paggr? 43
is-poly-tdesc? 42
is-poly? 42
is-readable? 43
is-real-tdesc? 41
is-real? 41
is-record-tdesc? 42
is-record? 42

163
r4 152
r5 152
r6 152
r7 152
r8 152
r9 152
real-lb 122
real-op 96
real-type-desc 40
real-ub 122
record-to-type 118
ref-mode 43
remove-enclosing-pkgs 65
rest 15
restype1 83
restype1 155
restype2 85
restype2 155
resval1 83
resval2 86
return 147
reverse 79
reverse-aux 79
rt1 82
rtype 43
rx1 96
scalar-op 96
scalar-signal-assignment 134
second 14
set-card 72
signatures 43
simple-name-match 65
simple-term 113
sixth 15
slt0 69
slt1 69
slt2 69
slx0 91
slx1 91
slx2 91
ss0 132
ss1 132
ss10 142
ss11 143
ss12 147
ss13 148
ss2 132
ss3 132
ss4 133
ss5 136
ss6 136
ss7 138
ss8 139
ss9 140
sst0 70
sst1 70
sst10 75
sst11 75
sst12 76
sst13 76
sst2 70
sst3 70
sst4 71
sst5 71
sst6 72
sst7 73
sst8 73
sst9 74
ssx1 92
ssx10 92
ssx11 92
ssx12 92
ssx13 92
ssx2 92
ssx3 92
ssx4 92
ssx5 92
ssx6 92
ssx7 92
ssx8 92
ssx9 92
string-type-desc 41
subst-vars 107
t0 154
t1 154
t10 154
t11 154
t12 154
t13 155
t14 155
t2 154
t3 154
166